1.1 Overview of the Memory Pool System

The Memory Pool System is a very general, adaptable, flexible, reliable, and efficient memory management system. It permits the flexible combination of memory management techniques, supporting manual and automatic memory management, inline allocation, finalization, weakness, and multiple concurrent co-operating incremental generational garbage collections. It also includes a library of memory pool classes implementing specialized memory management policies.

The MPS has been in development since 1994 and deployed in successful commercial products since 1997. Bugs are almost unknown in production. It is under continuous development and support by Ravenbrook.

The MPS is distributed under an open source license. The license is designed to make it possible for you to use the MPS in your own projects, provided that you either don’t distribute your product, or your product is open source too.

If the licensing terms aren’t suitable for you (for example, you’re developing a closed-source commercial product or a compiler run-time system) you can easily license the MPS under different terms from Ravenbrook by arrangement. Please contact us at mps-questions@ravenbrook.com for details.

1.1.1 Supported target platforms

The MPS is currently supported for deployment on:

- Windows XP or later, on IA-32 and x86-64, using Microsoft Visual C/C++;
- Linux 2.4 or later, on IA-32 using GCC and on x86-64 using GCC or Clang/LLVM;
- FreeBSD 7 or later, on IA-32 and x86-64, using GCC;
- OS X 10.4 or later, on IA-32 and x86-64, using Clang/LLVM.

The MPS is highly portable and has run on many other processors and operating systems in the past (see Building the Memory Pool System). Most of the MPS is written in very pure ANSI C and compiles without warnings on anything.

**Warning:** If you are running a multi-threaded 32-bit application on 64-bit Windows 7 via the WOW64 emulator, then you must install this hotfix from Microsoft. See WOW64 bug: GetThreadContext() may return stale contents for a description of the problem.
1.1.2 Technical introduction

The figure below gives a simplified picture of a program’s memory from the point of view of the Memory Pool System.

![Fig. 1.1: Overview of the Memory Pool System.](image)

The **arena** is the top-level data structure in the MPS. An **arena** is responsible for requesting **memory** from the operating system (and returning it), for making memory available to **pools**, and for **garbage collection**. Multiple arenas are supported, but it’s usually best to have only one arena in your program, because the MPS can’t collect cyclic structures that span multiple arenas. See **Arenas**.

The MPS is designed to co-operate with other memory managers (for example **malloc** and **free** in **C**, or operators **new** and **delete** in **C++**), so you need not move all your memory management to the MPS at once, and you can co-operate with libraries that use other allocation mechanisms.

Within the arena you create one or more **pools**. A **pool** is responsible for requesting memory from the **arena** and making it available to your program. See **Pools**.

Pools belong to **pool classes** that specify policies for how their memory is managed. Some pools are **manually managed** (you must explicitly return memory to the pool, for example by calling **mps_free()**) and others are **automatically managed** (the garbage collector reclaims unreachable blocks). See **Pool reference**.

**Formatted** pools need you to tell them how to **scan** for **references** to allocated blocks. See **Scanning**.

The arena needs you to tell it how to find your **roots**: references to allocated blocks that are stored in static data, in memory not managed by the MPS, in your program’s **registers**, or on its **control stack**. See **Roots**.

The MPS is designed to work with multi-threaded programs. Functions in the C interface are thread safe, except in a few documented cases. See **Threads**. The **allocation point protocol** provides fast lock-free allocation on multiple threads simultaneously. See **Allocation**.

The garbage collector is **incremental**: it proceeds in small steps interleaved with the execution of your program, so there are no long waits. The garbage collector is designed to work efficiently with multiple pools, and in cases where there are many references between objects in different pools. See **Garbage collection**.

1.1.3 What next?

For a much more detailed technical overview of the MPS, see **Brooksby (2002)**.

If you’re going to try it out, see **Building the Memory Pool System**.

If you have a program in need of memory management, then you’ll want to learn how to integrate it with the Memory Pool System. See **Garbage collecting a language with the Memory Pool System**.

If you want to know more technical details, they appear in the **Reference**.

1.2 Building the Memory Pool System

1.2.1 Introduction

This document describes the various ways in which you can build the MPS, its libraries, and the tests and tools that come with it.

You may be building the MPS for a number of different purposes.
1.2.2 Getting hold of the MPS Kit

Download the latest MPS Kit release from http://www.ravenbrook.com/project/mps/release/.

1.2.3 Compiling the MPS for your project

It is easy to compile the MPS. You can do it separately, or include the source in your own project’s build system. This section describes compilation in terms of command lines, but you can equally add the files to a project in an IDE.

The MPS also comes with Makefiles and IDE project files for building libraries, tools, and tests. See “Building the MPS for development”.

Compiling for production

In the simplest case, you can compile the MPS to an object file with just:

<table>
<thead>
<tr>
<th>Command Line</th>
<th>Description</th>
<th>Environment</th>
</tr>
</thead>
<tbody>
<tr>
<td>cc -c mps.c</td>
<td>Compile MPS to object file</td>
<td>(Unix/Mac OS X)</td>
</tr>
<tr>
<td>cl /c mps.c</td>
<td></td>
<td>(Windows)</td>
</tr>
</tbody>
</table>

This will build a “hot” variety (for production) object file for use with `mps.h`. You can greatly improve performance by allowing global optimization, for example:

<table>
<thead>
<tr>
<th>Command Line</th>
<th>Description</th>
<th>Environment</th>
</tr>
</thead>
<tbody>
<tr>
<td>cc -O2 -c mps.c</td>
<td>Compile MPS with optimization</td>
<td>(Unix/Mac OS X)</td>
</tr>
<tr>
<td>cl /O2 /c mps.c</td>
<td></td>
<td>(Windows)</td>
</tr>
</tbody>
</table>

Compiling for debugging

You can get a “cool” variety MPS (with more internal checking, for debugging and development) with:

<table>
<thead>
<tr>
<th>Command Line</th>
<th>Description</th>
<th>Environment</th>
</tr>
</thead>
<tbody>
<tr>
<td>cc -g -DCONFIG_VAR_COOL -c mps.c</td>
<td>Compile MPS with debugging</td>
<td>(Unix/Mac OS X)</td>
</tr>
<tr>
<td>cl /Zi /DCONFIG_VAR_COOL /c mps.c</td>
<td></td>
<td>(Windows)</td>
</tr>
</tbody>
</table>

Optimizing for your object format

If you are using your own object format, you will also get improved performance by allowing the compiler to do global optimizations between it and the MPS. So if your format implementation is in, say, `myformat.c`, then you could make a file `mymps.c` containing:

```
#include "mps.c"
#include "myformat.c"
```

then:

<table>
<thead>
<tr>
<th>Command Line</th>
<th>Description</th>
<th>Environment</th>
</tr>
</thead>
<tbody>
<tr>
<td>cc -O2 -c mymps.c</td>
<td>Compile MPS with optimization</td>
<td>(Unix/Mac OS X)</td>
</tr>
<tr>
<td>cl /O2 /c mymps.c</td>
<td></td>
<td>(Windows)</td>
</tr>
</tbody>
</table>

This will get your format code inlined with the MPS garbage collector.

Compiling without the C library

If you’re building the MPS for an environment without the standard C library, you can exclude the plinth component of the MPS with:
but you must then provide your own implementation of `mpslib.h`. You can base this on the ANSI plinth in `mpsliban.c`.

If you want to do anything beyond these simple cases, use the MPS build as described in the section “Building the MPS for development” below.

### 1.2.4 Building the MPS for development

If you’re making modifications to the MPS itself, want to build MPS libraries for linking, or want to build MPS tests and tools, you should use the MPS build. This uses makefiles or Xcode projects.

#### Prerequisites

For Unix-like platforms you will need the GNU Make tool. Some platforms (such as Linux) have GNU Make as their default make tool. For others you will need to get and install it. (It’s available free from ftp://ftp.gnu.org/gnu/make/.)

On FreeBSD this can be done as root with `pkg_add -r gmake`.

On Windows platforms the NMAKE tool is used. This comes with Microsoft Visual Studio C++ or the Microsoft Windows SDK.

On Mac OS X the MPS is built using Xcode, either by opening `mps.xcodeproj` with the Xcode app, or using the command-line “xcodebuild” tool, installed from Xcode → Preferences → Downloads → Components → Command Line Tools.

#### Platforms

The MPS uses a six-character platform code to express a combination of operating system, CPU architecture, and compiler toolchain. Each six-character code breaks down into three pairs of characters, like this:

```
OSARCT
```

Where **OS** denotes the operating system, **AR** the CPU architecture, and **CT** the compiler toolchain. Here are the platforms that we have regular access to and on which the MPS works well:

<table>
<thead>
<tr>
<th>Platform</th>
<th>OS</th>
<th>Architecture</th>
<th>Compiler</th>
<th>Makefile</th>
</tr>
</thead>
<tbody>
<tr>
<td>fri3gc</td>
<td>FreeBSD</td>
<td>IA-32</td>
<td>GCC</td>
<td>fri3gc.gmk</td>
</tr>
<tr>
<td>fri6gc</td>
<td>FreeBSD</td>
<td>x86_64</td>
<td>GCC</td>
<td>fri6gc.gmk</td>
</tr>
<tr>
<td>lii3gc</td>
<td>Linux</td>
<td>IA-32</td>
<td>GCC</td>
<td>lii3gc.gmk</td>
</tr>
<tr>
<td>lii6gc</td>
<td>Linux</td>
<td>x86_64</td>
<td>GCC</td>
<td>lii6gc.gmk</td>
</tr>
<tr>
<td>lii6ll</td>
<td>Linux</td>
<td>x86_64</td>
<td>Clang</td>
<td>lii6ll.gmk</td>
</tr>
<tr>
<td>xci3ll</td>
<td>Mac OS X</td>
<td>IA-32</td>
<td>Clang</td>
<td>mps.xcodeproj</td>
</tr>
<tr>
<td>xci6ll</td>
<td>Mac OS X</td>
<td>x86_64</td>
<td>Clang</td>
<td>mps.xcodeproj</td>
</tr>
<tr>
<td>xci3gc</td>
<td>Mac OS X</td>
<td>IA-32</td>
<td>GCC (legacy)</td>
<td>xci3gc.gmk</td>
</tr>
<tr>
<td>w3i3mv</td>
<td>Windows</td>
<td>IA-32</td>
<td>Microsoft C</td>
<td>w3i3mv.nmk</td>
</tr>
<tr>
<td>w3i6mv</td>
<td>Windows</td>
<td>x86_64</td>
<td>Microsoft C</td>
<td>w3i6mv.nmk</td>
</tr>
</tbody>
</table>

Historically, the MPS worked on a much wider variety of platforms, and still could: IRIX, OSF/1 (Tru64), Solaris, SunOS, Classic Mac OS; MIPS, PowerPC, ALPHASPARC v8, SPARC v9; Metrowerks Codewarrior, SunPro C, Digital C, EGCS, Pelles C. If you are interested in support on any of these platforms or any new platforms, please contact Ravenbrook at mps-questions@ravenbrook.com.
**Running make**

To build all MPS targets on Unix-like platforms, change to the code directory and run the command:

```
make -f <makefile>
```

where `make` is the command for GNU Make. (Sometimes this will be `gmake` or `gnumake`.)

To build just one target, run:

```
make -f <makefile> <target>
```

To build a restricted set of targets for just one variety, run:

```
make -f <makefile> 'VARIETY=<variety>' <target>
```

For example, to build just the “cool” variety of the `amcss` test on FreeBSD:

```
gmake -f fri3gc.gmk VARIETY=cool amcss
```

On Windows platforms you need to run the “Visual Studio Command Prompt” from the Start menu. Then run one of these commands:

```
nmake /f w3i3mv.nmk (32-bit)
nmake /f w3i6mv.nmk (64-bit)
```

You will need to switch your build environment between 32-bit and 64-bit using Microsoft’s `setenv` command, for example, `setenv /x86` or `setenv /x64`.

To build just one target, run one of these commands:

```
nmake /f w3i3mv.nmk <target> (32-bit)
nmake /f w3i6mv.nmk <target> (64-bit)
```

On Mac OS X, you can build from the command line with:

```
xcodebuild
```

On most platforms, the output of the build goes to a directory named after the platform (e.g. `fri3gc`) so that you can share the source tree across platforms. On Mac OS X the output goes in a directory called `xc`. Building generates `mps.a` or `mps.lib` or equivalent, a library of object code which you can link with your application, subject to the *MPS licensing conditions*. It also generates a number of test programs, such as `amcss` (a stress test for the Automatic Mostly-Copying pool class) and tools such as `mpseventcnv` (for decoding telemetry logs).

### 1.2.5 Installing the Memory Pool System

Unix-like platforms can use the GNU Autoconf `configure` script in the root directory of the MPS Kit to generate a Makefile that can build and install the MPS. For example:

```
./configure --prefix=/opt/mps
make install
```

will install the MPS public headers in `/opt/mps/include`, the libraries in `/opt/mps/lib` etc.

There is currently no automatic way to “install” the MPS on Windows.

On any platform, you can install by copying the libraries built by the make to, for example, `/usr/local/lib`, and all the headers beginning with `mps` to `/usr/local/include`.

Note, however, that you may get better performance by using the method described in the section “Optimizing for your object format” above.
mpseventsql

The MPS Kit can build a command-line program mpseventsql that loads a diagnostic stream of events into a SQLite3 database for processing. In order to build this program, you need to install the SQLite3 development resources.

- On Mac OS X, SQLite3 is pre-installed, so this tool builds by default.
- On Linux, you need to install the libsqlite3-dev package:

  ```bash
  apt-get install libsqlite3-dev
  ```

  and then re-run ./configure and make as described above.
- On FreeBSD, you need to build and install the databases/sqlite3 port from the ports collection:

  ```bash
  cd /usr/ports/databases/sqlite3
  make install clean
  ```

  and then re-run ./configure and make as described above.
- On Windows, you should visit the SQLite Download Page and download the sqlite-amalgamation ZIP archive. (At time of writing this is the first download on the page.) When you unzip the archive, you’ll find it contains files named sqlite3.c and sqlite3.h. Copy these two files into the code directory in the MPS Kit. Then in the “Visual Studio Command Prompt”, visit the code directory and run one of these commands:

  ```bash
  nmake /f w3i3mv.nmk mpseventsql.exe (32-bit)
  nmake /f w3i6mv.nmk mpseventsql.exe (64-bit)
  ```

1.3 Garbage collecting a language with the Memory Pool System

Have you written the lexer, parser, code generator and the runtime system for your programming language, and come to the realization that you are going to need a memory manager too? If so, you’ve come to the right place.

In this guide, I’ll explain how to use the MPS to add incremental, moving, generational garbage collection to the runtime system for a programming language.

I’m assuming that you are familiar with the overall architecture of the MPS (see the chapter Overview of the Memory Pool System) and that you’ve downloaded and built the MPS (see the chapter Building the Memory Pool System).

1.3.1 The Scheme interpreter

As a running example throughout this guide, I’ll be using a small interpreter for a subset of the Scheme programming language. I’ll be quoting the relevant sections of code as needed, but you may find it helpful to experiment with this interpreter yourself, in either of its versions:

scheme-malloc.c

  The toy Scheme interpreter before integration with the MPS, using malloc and free (2) for memory management.

scheme.c

  The toy Scheme interpreter after integration with the MPS.

This simple interpreter allocates two kinds of objects on the heap:

1. All Scheme objects (there are no unboxed objects).
2. The global symbol table: a hash table consisting of a vector of pointers to strings.

A Scheme object (whose type is not necessarily known) is represented by an obj_t, which is a pointer to a union of every type in the language:

```c
typedef union obj_u *obj_t;
typedef union obj_u {
    type_s type;
    pair_s pair;
    symbol_s symbol;
    integer_s integer;
    special_s special;
    operator_s operator;
    string_s string;
    port_s port;
    character_s character;
    vector_s vector;
    table_s table;
    buckets_s buckets;
} obj_s;
```

Each of these types is a structure whose first word is a number specifying the type of the object (TYPE_PAIR for pairs, TYPE_SYMBOL for symbols, and so on). For example, pairs are represented by a pointer to the structure pair_s defined as follows:

```c
typedef struct pair_s {
    type_t type;  /* TYPE_PAIR */
    obj_t car, cdr;  /* first and second projections */
} pair_s;
```

Because the first word of every object is its type, functions can operate on objects generically, testing TYPE(obj) as necessary (which is a macro for obj->type.type). For example, the print() function is implemented like this:

```c
static void print(obj_t obj, unsigned depth, FILE *stream) {
    switch (TYPE(obj)) {
    case TYPE_INTEGER:
        fprintf(stream, "%ld", obj->integer.integer);
        break;
    case TYPE_SYMBOL:
        fputs(obj->symbol.string, stream);
        break;
    /* ... and so on for the other types ... */
    }
}
```

Each constructor allocates memory for the new object by calling malloc. For example, make_pair is the constructor for pairs:

```c
static obj_t make_pair(obj_t car, obj_t cdr) {
    obj_t obj = (obj_t)malloc(sizeof(pair_s));
    if (obj == NULL) error("out of memory");
    obj->pair.type = TYPE_PAIR;
    CAR(obj) = car;
    CDR(obj) = cdr;
    return obj;
}
```

1.3. Garbage collecting a language with the Memory Pool System
Objects are never freed, because it is necessary to prove that they are *dead* before their memory can be *reclaimed*. To prove that they are dead, we need a *tracing garbage collector*, which the MPS will provide.

### 1.3.2 Choosing an arena class

You’ll recall from the *Overview of the Memory Pool System* that the functionality of the MPS is divided between the *arenas*, which request memory from (and return it to) the operating system, and *pools*, which allocate blocks of memory for your program.

There are two main classes of arena: the *client arena*, `mps_arena_class_cl()`, which gets its memory from your program, and the *virtual memory arena*, `mps_arena_class_vm()`, which gets its memory from the operating system’s *virtual memory* interface.

The client arena is intended for use on embedded systems where there is no virtual memory, and has a couple of disadvantages (you have to decide how much memory you are going to use; and the MPS can’t return memory to the operating system for use by other processes) so for general-purpose programs you’ll want to use the virtual memory arena.

You’ll need a couple of headers: `mps.h` for the MPS interface, and `mpsavm.h` for the virtual memory arena class:

```c
#include "mps.h"
#include "mpsavm.h"
```

There’s only one arena, and many MPS functions take an arena as an argument, so it makes sense for the arena to be a global variable rather than having to pass it around everywhere:

```c
static mps_arena_t arena;
```

Create an arena by calling `mps_arena_create_k()`. This function takes a *keyword argument* when creating a virtual memory arena: the size of virtual *address space* (*not* RAM), in bytes, that the arena will reserve initially. The MPS will ask for more address space if it runs out, but the more times it has to extend its address space, the less efficient garbage collection will become. The MPS works best if you reserve an address space that is several times larger than your peak memory usage.

**Note**

Functions in the MPS interface take *keyword arguments* for arguments that are optional, or are only required in some circumstances. These argument are passed in the form of an array of structures of type `mps_arg_s`. See *Keyword arguments* for the full details.

Let’s reserve 32 megabytes:

```c
mps_res_t res;
MPS_ARGS_BEGIN(args) {
    MPS_ARGS_ADD(args, MPS_KEYARENASIZE, 32 * 1024 * 1024);
    res = mps_arena_create_k(&arena, mps_arena_class_vm(), args);
} MPS_ARGS_END(args);
if (res != MPS_RES_OK) error("Couldn't create arena");
```

`mps_arena_create_k()` is typical of functions in the MPS interface in that it stores its result in a location pointed to by an *out parameter* (here, `&arena`) and returns a *result code*, which is `MPS_RES_OK` if the function succeeded, or some other value if it failed.

**Note**

The MPS is designed to co-operate with other memory managers, so when integrating your language with the MPS you need not feel obliged to move all your memory management to the MPS: you can continue to use `malloc` and `free` to manage some of your memory, for example, while using the MPS for the rest.
The toy Scheme interpreter illustrates this by continuing to use malloc and free to manage its global symbol table.

Topics
Arenas, Error handing.

1.3.3 Choosing a pool class

Pool classes come with a policy for how their memory will be managed: some pool classes use automatic memory management and others use manual; some use moving collection and others non-moving.

The section Choosing a pool class in the Pool reference contains a procedure for choosing a pool class. In the case of the toy Scheme interpreter, the answers to the questions are (1) yes, the MPS needs to automatically reclaim unreachable blocks; (2) yes, it’s acceptable for the MPS to move blocks in memory and protect them with barriers; and (3) the Scheme objects will contain exact references to other Scheme objects in the same pool.

The recommended class is AMC (Automatic Mostly-Copying). This pool class uses automatic memory management, moving garbage collection, allocation points and formatted objects, so it will provide an introduction to these features of the MPS.

Note
The MPS is designed for pools of different classes to co-exist in the same arena, so that objects requiring different memory management policies can be segregated into pools of suitable classes.

Topic
Pools.

1.3.4 Describing your objects

In order for the MPS to be able to automatically manage your objects, you need to tell it how to perform various operations on an object (scan it for references; replace it with a forwarding or padding object, and so on). You do this by creating an object format. Here’s the code for creating the object format for the toy Scheme interpreter:

```c
MPS_ARGS_BEGIN(args) {
    MPS_ARGS_ADD(args, MPS_KEY_FMT_ALIGN, ALIGNMENT);
    MPS_ARGS_ADD(args, MPS_KEY_FMT_SCAN, obj_scan);
    MPS_ARGS_ADD(args, MPS_KEY_FMT_SKIP, obj_skip);
    MPS_ARGS_ADD(args, MPS_KEY_FMT_FWD, obj_fwd);
    MPS_ARGS_ADD(args, MPS_KEY_FMT_ISFWD, obj_isfwd);
    MPS_ARGS_ADD(args, MPS_KEY_FMT_PAD, obj_pad);
    res = mps_fmt_create_k(&obj_fmt, arena, args);
} MPS_ARGS_END(args);
if (res != MPS_RES_OK) error("Couldn't create obj format");
```

The keyword arguments specify the alignment and the format methods required by the AMC pool class. These are described in the following sections.

Topic
Object formats.
Alignment

The argument for the keyword MPS_KEY_FMT_ALIGN is the alignment of objects belonging to this format. Determining the alignment is hard to do portably, because it depends on the target architecture and on the way the compiler lays out its structures in memory. Here are some things you might try:

1. Some modern compilers support the `alignof` operator:

   ```
   #define ALIGNMENT alignof(obj_s)
   ```

2. On older compilers you may be able to use this trick:

   ```
   #define ALIGNMENT offsetof(struct {char c; obj_s obj;}, obj)
   ```

   but this is not reliable because some compilers pack structures more tightly than their alignment requirements in some circumstances (for example, GCC if the `-fstruct-pack` option is specified).

3. The MPS interface provides the type `mps_word_t`, which is an unsigned integral type that is the same size as the platform’s object pointer types.

   So if you know that all your objects can be word-aligned, you can use:

   ```
   #define ALIGNMENT sizeof(mps_word_t)
   ```

4. The MPS interface provides the type `MPS_PF_ALIGN`, which is the natural alignment of the platform: the largest alignment that might be required. So as a last resort, you can use:

   ```
   #define ALIGNMENT MPS_PF_ALIGN
   ```

   But this may be larger than necessary and so waste space. For example, on Windows on x86-64, `MPS_PF_ALIGN` is 16 bytes, but this is only necessary for SSE types; ordinary types on this platform require no more than 8-byte alignment.

The scan method

The scan method is a function of type `mps_fmt_scan_t`. It is called by the MPS to scan a block of memory. Its task is to identify all references within the objects in the block of memory, and “fix” them, by calling the macros `MPS_FIX1()` and `MPS_FIX2()` on each reference (possibly via the convenience macro `MPS_FIX12()`).

“Fixing” is a generic operation whose effect depends on the context in which the scan method was called. The scan method is called to discover references and so determine which objects are alive and which are dead, and also to update references after objects have been moved.

Here’s the scan method for the toy Scheme interpreter:

```c
static mps_res_t obj_scan(mps_ss_t ss, mps_addr_t base, mps_addr_t limit) {
    MPS_SCAN_BEGIN(ss) {
        while (base < limit) {
            obj_t obj = base;
            switch (TYPE(obj)) {
            case TYPE_PAIR:
                FIX(CAR(obj));
                FIX(CDR(obj));
                base = (char *)base + ALIGN_OBJ(sizeof(pair_s));
                break;
            case TYPE_INTEGER:
                base = (char *)base + ALIGN_OBJ(sizeof(integer_s));
                break;
            ```
The scan method receives a \textit{scan state} (ss) argument, and the block of memory to scan, from \texttt{base} (inclusive) to \texttt{limit} (exclusive). This block of memory is known to be packed with objects belonging to the object format, and so the scan method loops over the objects in the block, dispatching on the type of each object, and then updating \texttt{base} to point to the next object in the block.

For each reference in an object \texttt{obj\_scan} fixes it by calling \texttt{MPS\_FIX12()} via the macro \texttt{FIX}, which is defined as follows:

```c
#define FIX(ref) do { 
    mps_addr_t _addr = (ref); /* copy to local to avoid type pun */ 
    mps_res_t res = MPS\_FIX12(ss, &_addr); 
    if (res != MPS\_RES\_OK) return res; 
    (ref) = _addr; 
} while (0)
```

Each call to \texttt{MPS\_FIX12()} must appear between calls to the macros \texttt{MPS\_SCAN\_BEGIN()} and \texttt{MPS\_SCAN\_END()}. It’s usually most convenient to call \texttt{MPS\_SCAN\_BEGIN()} at the start of the function and \texttt{MPS\_SCAN\_END()} at the end, as here.

Notes

1. When the MPS calls your scan method, it may be part-way through moving your objects. It is therefore essential that the scan method only examine objects in the range of addresses it is given. Objects in other ranges of addresses are not guaranteed to be in a consistent state.

2. Scanning is an operation on the \textit{critical path} of the MPS, which means that it is important that it runs as quickly as possible.

3. If your reference is \textit{tagged}, you must remove the tag before fixing it. (This is not quite true, but see \textit{Tagged references} for the full story.)

4. The “fix” operation may update the reference. So if your reference is tagged, you must make sure that the tag is restored after the reference is updated.

5. The “fix” operation may fail by returning a \textit{result code} other than \texttt{MPS\_RES\_OK}. A scan function must propagate such a result code to the caller, and should do so as soon as practicable.

Topics

\textit{Object formats}, \textit{Scanning}.

The skip method

The \textit{skip method} is a function of type \texttt{mps\_fmt\_skip\_t}. It is called by the MPS to skip over an object belonging to the format, and also to determine its size.
Here’s the skip method for the toy Scheme interpreter:

```c
static mps_addr_t
obj_skip(mps_addr_t base)
{
    obj_t obj = base;
    switch (TYPE(obj)) {
    case TYPE_PAIR:
        base = (char *)base + ALIGN_OBJ(sizeof(pair_s));
        break;
    case TYPE_INTEGER:
        base = (char *)base + ALIGN_OBJ(sizeof(integer_s));
        break;
    /* ... and so on for the other types ... */
    default:
        assert(0);
        fprintf(stderr, "Unexpected object on the heap
        \n");
        abort();
    }
    return base;
}
```

The argument `base` is the address to the base of the object. The skip method must return the address of the base of the “next object”: in formats of variant A like this one, this is the address just past the end of the object, rounded up to the object format’s alignment.

---

**Topic

Object formats.**

---

**The forward method**

The forward method is a function of type `mps_fmt_fwd_t`. It is called by the MPS after it has moved an object, and its task is to replace the old object with a forwarding object pointing to the new location of the object.

![Fig. 1.2: Copying garbage collection.](image)

The forwarding object must satisfy these properties:

1. It must be scannable and skippable, and so it will need to have a type field to distinguish it from other Scheme objects.
2. It must contain a pointer to the new location of the object (a forwarding pointer).
3. It must be the same size as the old object. This means that the scan method and the skip method will both need to know the length of the forwarding object. This can be arbitrarily long (in the case of string objects, for example) so it must contain a length field.

This poses a problem, because the above analysis suggests that forwarding objects need to contain at least three words, but Scheme objects might be as small as two words (for example, integers).

This conundrum can be solved by having two types of forwarding object. The first type is suitable for forwarding objects of three words or longer:

```c
typedef struct
fwd_s {
    type_t type; /* TYPE_FWD */
    obj_t fwd; /* forwarded object */
} fwd_s;
```
while the second type is suitable for forwarding objects of two words:

```c
typedef struct fwd2_s {
    type_t type; /* TYPE_FWD2 */
    obj_t fwd; /* forwarded object */
} fwd2_s;
```

Here's the forward method for the toy Scheme interpreter:

```c
static void obj_fwd(mps_addr_t old, mps_addr_t new)
{
    obj_t obj = old;
    mps_addr_t limit = obj_skip(old);
    size_t size = (char *)limit - (char *)old;
    assert(size >= ALIGN_WORD(sizeof(fwd2_s)));
    if (size == ALIGN_WORD(sizeof(fwd2_s))) {
        TYPE(obj) = TYPE_FWD2;
        obj->fwd2.fwd = new;
    } else {
        TYPE(obj) = TYPE_FWD;
        obj->fwd.fwd = new;
        obj->fwd.size = size;
    }
}
```

The argument `old` is the old address of the object, and `new` is the location to which it has been moved.

The forwarding objects must be scannable and skippable, so the following code must be added to `obj_scan` and `obj_skip`:

```c
case TYPE_FWD:
    base = (char *)base + ALIGN_WORD(obj->fwd.size);
    break;
```

```c
case TYPE_FWD2:
    base = (char *)base + ALIGN_WORD(sizeof(fwd2_s));
    break;
```

Note

Objects that consist of a single word present a problem for the design of the forwarding object. In the toy Scheme interpreter, this happens on some 64-bit platforms, where a pointer is 8 bytes long, and a character_s object (which consists of a 4-byte int and a 1-byte char) is also 8 bytes long.

There are a couple of solutions to this problem:

1. Allocate the small objects with enough padding so that they can be forwarded. (This is how the problem is solved in the toy Scheme interpreter.)

2. Use a tag to distinguish between the client object and a forwarding object that replaces it. It might help to allocate the small objects in their own pool so that the number of types that the scan method has to distinguish is minimized. Since these objects do not contain references, they could be allocated from the AMCZ (Automatic Mostly-Copying Zero-rank) pool, and so the cost of scanning them could be avoided.
The is-forwarded method

The is-forwarded method is a function of type mps_fmt_isfwd_t. It is called by the MPS to determine if an object is a forwarding object, and if it is, to determine the location where that object was moved.

Here’s the is-forwarded method for the toy Scheme interpreter:

```c
static mps_addr_t obj_isfwd(mps_addr_t addr)
{
    obj_t obj = addr;
    switch (TYPE(obj)) {
    case TYPE_FWD2:
        return obj->fwd2.fwd;
    case TYPE_FWD:
        return obj->fwd.fwd;
    }
    return NULL;
}
```

It receives the address of an object, and returns the address to which that object was moved, or NULL if the object was not moved.

The padding method

The padding method is a function of type mps_fmt_pad_t. It is called by the MPS to fill a block of memory with a padding object: this is an object that fills gaps in a block of formatted objects, for example to enable the MPS to pack objects into fixed-size units (such as operating system pages).

A padding object must be scannable and skippable, and not confusable with a forwarding object. This means they need a type and a size. However, padding objects might need to be as small as the alignment of the object format, which was specified to be a single word. As with forwarding objects, this can be solved by having two types of padding object. The first type is suitable for padding objects of two words or longer:

```c
typedef struct pad_s {
    type_t type;       /* TYPE_PAD */
    size_t size;       /* total size of this object */
} pad_s;
```

while the second type is suitable for padding objects consisting of a single word:

```c
typedef struct pad1_s {
    type_t type;       /* TYPE_PAD1 */
} pad1_s;
```

Here’s the padding method:

```c
static void obj_pad(mps_addr_t addr, size_t size)
{
    obj_t obj = addr;
    assert(size >= ALIGN_OBJ(sizeof(pad1_s)));
    if (size == ALIGN_OBJ(sizeof(pad1_s))) {
        TYPE(obj) = TYPE_PAD1;
    } else {
        TYPE(obj) = TYPE_PAD;
    }
}```
The argument `addr` is the address at which the padding object must be created, and `size` is its size in bytes: this will always be a multiple of the alignment of the object format.

The padding objects must be scannable and skippable, so the following code must be added to `obj_scan` and `obj_skip`:

```c
    case TYPE_PAD:
        base = (char *)base + ALIGN_OBJ(obj->pad.size);
        break;
    case TYPE_PAD1:
        base = (char *)base + ALIGN_OBJ(sizeof(pad1_s));
        break;
```

**Topic**

*Object formats.*

### 1.3.5 Creating the pool

Now you know enough to create an *AMC (Automatic Mostly-Copying)* pool! Let’s review the pool creation code. First, the header for the AMC pool class:

```c
#include "mpscamc.h"
```

Second, the *object format*:

```c
MPS_ARGS_BEGIN(args) {
    MPS_ARGS_ADD(args, MPS_KEY_FMT_ALIGN, ALIGNMENT);
    MPS_ARGS_ADD(args, MPS_KEY_FMT_SCAN, obj_scan);
    MPS_ARGS_ADD(args, MPS_KEY_FMT_SKIP, obj_skip);
    MPS_ARGS_ADD(args, MPS_KEY_FMT_FWD, obj_fwd);
    MPS_ARGS_ADD(args, MPS_KEY_FMT_ISFWD, obj_isfwd);
    MPS_ARGS_ADD(args, MPS_KEY_FMT_PAD, obj_pad);
    res = mps_fmt_create_k(&obj_fmt, arena, args);
} MPS_ARGS_END(args);
if (res != MPS_RES_OK) error("Couldn't create obj format");
```

And finally the *pool*:

```c
mps_pool_t obj_pool;
MPS_ARGS_BEGIN(args) {
    MPS_ARGS_ADD(args, MPS_KEY_FORMAT, obj_fmt);
    res = mps_pool_create_k(&obj_pool, arena, mps_class_amc(), args);
} MPS_ARGS_END(args);
if (res != MPS_RES_OK) error("Couldn't create obj pool");
```

### 1.3.6 Roots

The *object format* tells the MPS how to find *references* from one object to another. This allows the MPS to extrapolate the reachability property: if object `A` is reachable, and the *scan method* fixes a reference from `A` to another object `B`, then `B` is reachable too.
But how does this process get started? How does the MPS know which objects are reachable \textit{a priori}? Such objects are known as \textit{roots}, and you must register them with the MPS, creating root descriptions of type \texttt{mps_root_t}.

The most important root consists of the contents of the \texttt{registers} and the \texttt{control stack} of each \texttt{thread} in your program: this is covered in \texttt{Threads}, below.

Other roots may be found in static variables in your program, or in memory allocated by other memory managers. For these roots you must describe to the MPS how to \texttt{scan} them for references.

The toy Scheme interpreter has a number of static variables that point to heap-allocated objects. First, the special objects, including:

```c
static obj_t obj_empty;    /* (), the empty list */
```

Second, the predefined symbols, including:

```c
static obj_t obj_quote;    /* "quote" symbol */
```

And third, the global symbol table:

```c
static obj_t *symtab;
static size_t symtab_size;
```

You tell the MPS how to scan these by writing root scanning functions of type \texttt{mps_root_scan_t}. These functions are similar to the \texttt{scan method} in an \texttt{object format}, described above.

In the case of the toy Scheme interpreter, the root scanning function for the special objects and the predefined symbols could be written like this:

```c
static mps_res_t globals_scan(mps_ss_t ss, void *p, size_t s)
{
    MPS_SCAN_BEGIN(ss) {
        FIX(obj_empty);
    /* ... and so on for the special objects ... */
        FIX(obj_quote);
    /* ... and so on for the predefined symbols ... */
    } MPS_SCAN_END(ss);
    return MPS_RES_OK;
}
```

but in fact the interpreter already has tables of these global objects, so it’s simpler and more extensible for the root scanning function to iterate over them:

```c
static mps_res_t globals_scan(mps_ss_t ss, void *p, size_t s)
{
    MPS_SCAN_BEGIN(ss) {
        size_t i;
        for (i = 0; i < LENGTH(sptab); ++i)
            FIX(*sptab[i].varp);
        for (i = 0; i < LENGTH(isymtab); ++i)
            FIX(*isymtab[i].varp);
    } MPS_SCAN_END(ss);
    return MPS_RES_OK;
}
```

Each root scanning function must be registered with the MPS by calling \texttt{mps_root_create()}, like this:

```c
mps_root_t globals_root;
res = mps_root_create(&globals_root, arena, mps_rank_exact(), 0,
globals_scan, NULL, 0);
if (res != MPS_RES_OK) error("Couldn't register globals root");
```
The third argument (here `mps_rank_exact()`) is the rank of references in the root. “Exact” means that:

1. each reference in the root is a genuine pointer to another object managed by the MPS, or else a null pointer (unlike ambiguous references); and
2. each reference keeps the target of the reference alive (unlike weak references (1)).

The fourth argument is the root mode, which tells the MPS whether it is allowed to place a barrier (1) on the root. The root mode 0 means that it is not allowed.

The sixth and seventh arguments (here `NULL` and 0) are passed to the root scanning function where they are received as the parameters `p` and `s` respectively. In this case there was no need to use them.

What about the global symbol table? This is trickier, because it gets rehashed from time to time, and during the rehashing process there are two copies of the symbol table in existence. Because the MPS is asynchronous, it might be scanning, moving, or collecting, at any point in time, and if it is doing so during the rehashing of the symbol table it had better scan both the old and new copies of the table. This is most conveniently done by registering a new root to refer to the new copy, and then after the rehash has completed, de-registering the old root by calling `mps_root_destroy()`.

It would be possible to write a root scanning function of type `mps_root_scan_t`, as described above, to fix the references in the global symbol table, but the case of a table of references is sufficiently common that the MPS provides a convenient (and optimized) function, `mps_root_create_table()`, for registering it:

```c
static mps_root_t symtab_root;
/* ... */

mps_addr_t ref = symtab;
res = mps_root_create_table(&symtab_root, arena, mps_rank_exact(), 0,
                            ref, symtab_size);
if (res != MPS_RES_OK) error("Couldn't register new symtab root");
```

The root must be re-registered whenever the global symbol table changes size:

```c
static void rehash(void) {
    obj_t *old_symtab = symtab;
    unsigned old_symtab_size = symtab_size;
    mps_root_t old_symtab_root = symtab_root;
    unsigned i;
    mps_addr_t ref;
    mps_res_t res;

    symtab_size *= 2;
    symtab = malloc(sizeof(obj_t) * symtab_size);
    if (symtab == NULL) error("out of memory");
    /* Initialize the new table to NULL so that "find" will work. */
    for (i = 0; i < symtab_size; ++i)
        symtab[i] = NULL;

    ref = symtab;
    res = mps_root_create_table(&symtab_root, arena, mps_rank_exact(), 0,
                                ref, symtab_size);
    if (res != MPS_RES_OK) error("Couldn't register new symtab root");
    for (i = 0; i < old_symtab_size; ++i)
        if (old_symtab[i] != NULL) {
            obj_t *where = find(old_symtab[i]->symbol.string);
            assert(where != NULL);
        } /* new table shouldn't be full */
}
```
assert(*where == NULL); /* shouldn’t be in new table */
*where = old_symtab[i];
}
mps_root_destroy(old_symtab_root);
free(old_symtab);
}

Notes
1. The old root description (referring to the old copy of the symbol table) is not destroyed until after the new root
description has been registered. This is because the MPS is asynchronous: it might be scanning, moving, or
collecting, at any point in time. If the old root description were destroyed before the new root description was
registered, there would be a period during which:
   (a) the symbol table was not reachable (at least as far as the MPS was concerned) and so all the objects
       referenced by it (and all the objects reachable from those objects) might be dead; and
   (b) if the MPS moved an object, it would not know that the object was referenced by the symbol table, and
       so would not update the reference there to point to the new location of the object. This would result in
       out-of-date references in the old symbol table, and these would be copied into the new symbol table.
2. The root might be scanned as soon as it is registered, so it is important to fill it with scannable references (NULL
   in this case) before registering it.
3. The order of operations at the end is important: the old root must be de-registered before its memory is freed.
4. When calling mps_root_create_table(), take care to avoid undefined behaviour due to type punning.
   See the warning.

Topic
Roots.

1.3.7 Threads

In a multi-threaded environment where incremental garbage collection is used, you must register each of your threads
with the MPS so that the MPS can examine their state.

Even in a single-threaded environment (like the toy Scheme interpreter) it may also be necessary to register the (only)
thread if either of these conditions apply:
   1. you are using moving garbage collection (as with the AMC (Automatic Mostly-Copying) pool);
   2. the thread’s registers and control stack constitute a root (that is, objects may be kept alive via references in local
      variables: this is almost always the case for programs written in C).

You register a thread with an arena by calling mps_thread_reg():

```c
mps_thr_t thread;
res = mps_thread_reg(&thread, arena);
if (res != MPS_RES_OK) error("Couldn't register thread");
```

You register the thread’s registers and control stack as a root by calling mps_root_create_reg() and passing
mps_stack_scan_ambig():

```c
void *marker = &marker;
mps_root_t reg_root;
```
res = mps_root_create_reg(&reg_root,
                     arena,
                     mps_rank_ambig(),
                     0,
                     thread,
                     mps_stack_scan_ambig,
                     marker,
                     0);
if (res != MPS_RES_OK) error("couldn't create root");

In order to scan the control stack, the MPS needs to know where the bottom of the stack is, and that’s the role of the
marker variable: the compiler places it on the stack, so its address is a position within the stack. As long as you
don’t exit from this function while the MPS is running, your program’s active local variables will always be higher up
on the stack than marker, and so will be scanned for references by the MPS.

Topic

Threads.

### 1.3.8 Allocation

It probably seemed a long journey to get here, but at last we’re ready to start allocating.

Manual pools typically support malloc-like allocation using the function mps_alloc(). But automatic pools cannot,
because of the following problem:

```c
static obj_t make_pair(obj_t car, obj_t cdr)
{
    obj_t obj;
    mps_addr_t addr;
    mps_res_t res;
    res = mps_alloc(&addr, pool, sizeof(pair_s));
    if (res != MPS_RES_OK) error("out of memory in make_pair");
    obj = addr;
    /* What happens if the MPS scans obj just now? */
    obj->pair.type = TYPE_PAIR;
    CAR(obj) = car;
    CDR(obj) = cdr;
    return obj;
}
```

Because the MPS is asynchronous, it might scan any reachable object at any time, including immediately after the
object has been allocated. In this case, if the MPS attempts to scan obj at the indicated point, the object’s type field
will be uninitialized, and so the scan method may abort.

The MPS solves this problem via the fast, nearly lock-free Allocation point protocol. This needs an additional structure,
an allocation point, to be attached to the pool by calling mps_ap_create_k():

```c
static mps_ap_t obj_ap;
/* ... */
res = mps_ap_create_k(&obj_ap, obj_pool, mps_args_none);
if (res != MPS_RES_OK) error("Couldn't create obj allocation point");
```
And then the constructor can be implemented like this:

```c
static obj_t make_pair(obj_t car, obj_t cdr)
{
    obj_t obj;
    mps_addr_t addr;
    size_t size = ALIGN_OBJ(sizeof(pair_s));
    do {
        mps_res_t res = mps_reserve(&addr, obj_ap, size);
        if (res != MPS_RES_OK) error("out of memory in make_pair");
        obj = addr;
        obj->pair.type = TYPE_PAIR;
        CAR(obj) = car;
        CDR(obj) = cdr;
    } while (!mps_commit(obj_ap, addr, size));
    return obj;
}
```

The function `mps_reserve()` allocates a block of memory that the MPS knows is uninitialized: the MPS promises not to scan this block or move it until after it is committed by calling `mps_commit()`. So the new object can be initialized safely.

However, there’s a second problem:

```c
CAR(obj) = car;
CDR(obj) = cdr;
/* What if the MPS moves car or cdr just now? */
while (!mps_commit(obj_ap, addr, size));
```

Because `obj` is not yet committed, the MPS won’t scan it, and that means that it won’t discover that it contains references to `car` and `cdr`, and so won’t update these references to point to their new locations.

In such a circumstance (that is, when objects have moved since you called `mps_reserve()`), `mps_commit()` returns false, and we have to initialize the object again (most conveniently done via a `while` loop, as here).

### Notes

1. When using the Allocation point protocol it is up to you to ensure that the requested size is aligned, because `mps_reserve()` is on the MPS’s critical path, and so it is highly optimized: in nearly all cases it is just an increment to a pointer and a test.

2. It is very rare for `mps_commit()` to return false, but in the course of millions of allocations even very rare events occur, so it is important not to do anything you don’t want to repeat between calling `mps_reserve()` and `mps_commit()`. Also, the shorter the interval, the less likely `mps_commit()` is to return false.

---

### 1.3.9 Maintaining consistency

The MPS is asynchronous: this means that it might be scanning, moving, or collecting, at any point in time (potentially, between any pair of instructions in your program). So you must make sure that your data structures always obey these rules:
1. A root must be scannable by its root scanning function as soon as it has been registered.

   See the discussion of the global symbol table in the toy Scheme interpreter.

2. A formatted object must be scannable by the scan method as soon as it has been committed (2) by calling mps_commit().

   See the discussion of the pair constructor in the toy Scheme interpreter.

3. All objects in automatically managed pools that are reachable by your code must always be provably reachable from a root via a chain of references that are fixed by a scanning function.

   See the discussion of the global symbol table in the toy Scheme interpreter.

4. Formatted objects must remain scannable throughout their lifetime.

Examples of code that breaks these rules, together with tactics for tracking down the causes, appear in the chapter Debugging with the Memory Pool System.

1.3.10 Tidying up

When your program is done with the MPS, you should park the arena (by calling mps_arena_park()) to ensure that no incremental garbage collection is in progress, and then tear down all the MPS data structures. This causes the MPS to check the consistency of its data structures and report any problems it detects. It also causes the MPS to flush its telemetry stream.

MPS data structures must be destroyed or deregistered in the reverse order to that in which they were registered or created. So you must destroy all allocation points created in a pool before destroying the pool; destroy all roots and pools, and deregister all threads, that were created in an arena before destroying the arena, and so on.

For example:

```c
mps_arena_park(arena);  /* ensure no collection is running */
mps_ap_destroy(obj_ap); /* destroy ap before pool */
mps_pool_destroy(obj_pool); /* destroy pool before fmt */
mps_root_destroy(reg_root); /* destroy root before thread */
mps_thread_dereg(thread); /* deregister thread before arena */
mps_fmt_destroy(obj_fmt); /* destroy fmt before arena */
mps_arena_destroy(arena); /* last of all */
```

1.3.11 What next?

This article has covered the basic knowledge needed to add incremental, moving, generational garbage collection to the runtime system for a programming language.

If everything is working for your language, then the next step is the chapter Tuning the Memory Pool System for performance.

But in the more likely event that things don’t work out quite as smoothly for your language as they did in the toy Scheme interpreter, then you’ll be more interested in the chapter Debugging with the Memory Pool System.

1.4 The stretchy vector problem

The previous chapter pointed out that:

Because the MPS is asynchronous, it might be scanning, moving, or collecting, at any point in time.
The consequences of this can take a while to sink in, so this chapter discusses a particular instance that catches people out: the *stretchy vector* problem (named after the `<stretchy-vector>` abstract class in Dylan).

A *stretchy vector* is a vector that can change length dynamically. Such a vector is often implemented using two objects: an array, and a header object that stores the length and a pointer to an array. Stretching (or shrinking) such a vector involves five steps:

1. allocate a new array;
2. copy elements from the old array to the new array;
3. clear unused elements in the new array (if stretching);
4. update the pointer to the array in the header;
5. update the length in the header.

For example:

```c
typedef struct vector_s {
    type_t type; /* TYPE_VECTOR */
    size_t length; /* number of elements */
    obj_t *array; /* array of elements */
} vector_s, *vector_t;

void resize_vector(vector_t vector, size_t new_length) {
    obj_t *new_array = realloc(vector->array, new_length * sizeof(obj_t));
    if (new_array == NULL)
        error("out of memory in resize_vector");
    if (vector->length < new_length) {
        memset(&vector->array<vector->length], 0,
               (new_length - vector->length) * sizeof(obj_t));
    }
    vector->array = new_array;
    vector->length = new_length;
}
```

When adapting this code to the MPS, the following problems must be solved:

1. During step 2, the new array must be *reachable* from the roots, and *scannable*. (If it’s not reachable, then it may be collected; if it’s not scannable, then references it contains will not be updated when they are moved by the collector.)

   This can be solved by storing the new array in a *root* until the header has been updated. If the thread’s stack has been registered as a root by calling `mps_root_create_reg()` then any local variable will do.

2. References in the new array must not be scanned until they have been copied or cleared. (Otherwise they will be invalid.)

   This can be solved by clearing the new array before calling `mps_commit()`.

3. The old array must be scanned at the old length (otherwise the scan may run off the end of the old array when the vector grows), and the new array must be scanned at the new length (otherwise the scan may run off the end of the old array when the vector shrinks).

4. The array object must be scannable without referring to the header object. (Because the header object may have been protected by the MPS: see *Cautions*.)

Problems 3 and 4 can be solved by storing the length in the array. The revised data structures and resizing code might look like this:

```c
typedef struct vector_s {
    type_t type; /* TYPE_VECTOR */
    size_t length;
    obj_t *array;
} vector_s, *vector_t;
```

```c
void resize_vector(vector_t vector, size_t new_length) {
    obj_t *new_array = realloc(vector->array, new_length * sizeof(obj_t));
    if (new_array == NULL)
        error("out of memory in resize_vector");
    if (vector->length < new_length) {
        memset(&vector->array<vector->length], 0,
               (new_length - vector->length) * sizeof(obj_t));
    }
    vector->array = new_array;
    vector->length = new_length;
    mps_commit();
}
```
typedef struct array_s {
    type_t type;        /* TYPE_ARRAY */
    size_t length;      /* number of elements */
    obj_t array[0];     /* array of elements */
} array_s, *array_t;

void resize_vector(vector_t vector, size_t new_length) {
    size_t size = ALIGN_OBJ(offsetof(array_s, array) + new_length * sizeof(obj_t));
    mps_addr_t addr;
    array_t array;

    do {
        mps_res_t res = mps_reserve(&addr, ap, size);
        if (res != MPS_RES_OK) error("out of memory in resize_vector");
        array = addr;
        array->type = TYPE_ARRAY;
        array->length = new_length;
        memset(array->array, 0, new_length * sizeof(obj_t));
        /* Now the new array is scannable, and it is reachable via the
         * local variable 'array', so it is safe to commit it. */
    } while (!mps_commit(ap, addr, size));
    /* Copy elements after committing, so that the collector will
     * update them if they move. */
    memcpy(array->array, vector->array->array,
            min(vector->array->length, new_length) * sizeof(obj_t));
    vector->array = array;
}

Similar difficulties can arise even when adapting code written for other garbage collectors. For example, here’s the
function setarrayvector() from Lua:

static void setarrayvector (lua_State *L, Table *t, int size) {
    int i;
    luaM_reallocvector(L, t->array, t->sizearray, size, TValue);
    for (i=t->sizearray; i<size; i++)
        setnilvalue(&t->array[i]);
    t->sizearray = size;
}

Lua’s garbage collector is synchronous, so it can be assumed that there cannot be a garbage collection between the
assignment to t->array (resulting from the expansion of the luaM_reallocvector() macro) and the assign-
ment to t->sizearray, and so the collector will always consistently see either the old array or the new array, with
the correct size. This assumption will no longer be correct if this code is adapted to the MPS.

1.5 Debugging with the Memory Pool System

Memory management errors are some of the most stubborn and difficult to track down, because the effect so often
appears at a distant point in the program that is seemingly unrelated to the cause, and by the time the error is revealed,
the information needed to reconstruct the cause has long vanished. Immediately after an overwriting error, the block
that overran its bounds is fine, and the block that was overwritten may not be visited for a long time. A failure to fix
a reference does not necessarily cause the object pointed to by the missed reference to die immediately: there may be
other references to that object, or a garbage collection may be delayed. And even if it does die, the space it occupies may not be re-allocated for some time.

### 1.5.1 General debugging advice

1. Compile with debugging information turned on (`-g` on the GCC or Clang command line).
2. Build the cool variety of the MPS (by defining the preprocessor constant `CONFIG_VAR_COOL`, for example by setting `-DCONFIG_VAR_COOL` on the GCC or Clang command line). This variety contains many internal consistency checks (including such checks on the critical path, which make it too slow for use in production), and can generate profiling output in the form of the telemetry stream.
3. If your program triggers an assertion failure in the MPS, consult Common assertions and their causes for suggestions as to the possible cause.
4. Prepare a reproducible test case if possible. The MPS may be asynchronous, but it is deterministic, so in single-threaded applications you should be able to get consistent results.

However, you need to beware of address space layout randomization: if you perform computation based on the addresses of objects, for example, hashing objects by their address, then ASLR will cause your hash tables to be laid out differently on each run, which may affect the order of memory management operations. See Address space layout randomization below.

A fact that assists with reproducibility is that the more frequently the collector runs, the sooner and more reliably errors are discovered. So if you have a bug that’s hard to reproduce, or which manifests itself in different ways on different runs, you may be able to provoke it more reliably, or get a more consistent result, by having a mode for testing in which you run frequent collections (by calling `mps_arena_collect()` followed by `mps_arena_release()`), perhaps as frequently as every allocation. (This will of course make the system run very slowly, but it ensures that if there are roots or references that are not being scanned then the failure will occur close in time to the cause, making it easier to diagnose.)

5. Run your test case inside the debugger. Use `assert` and `abort` in your error handler (rather than `exit`) so that you can enter the debugger with the contents of the control stack available for inspection.

You may need to make sure that the debugger isn’t entered on barrier (1) hits (because the MPS uses barriers to protect parts of memory, and barrier hits are common and expected).

If you are using GDB on Linux or FreeBSD, run this command:

```bash
handle SIGSEGV pass nostop noprint
```

On these operating systems, you can add this command to your `.gdbinit` if you always want it to be run.

On OS X, barrier hits do not use signals and so do not enter the debugger.

### 1.5.2 Address space layout randomization

Address space layout randomization (ASLR) makes it hard to prepare a repeatable test case for a program that performs computation based on the addresses of objects, for example, hashing objects by their address. If this is affecting you, you’ll find it useful to disable ASLR when testing.

Here’s a small program that you can use to check if ASLR is enabled on your system. It outputs addresses from four key memory areas in a program (data segment, text segment, stack and heap):

```c
#include <stdio.h>
#include <stdlib.h>

int data;
```
```c
int main() {
    void *heap = malloc(4);
    int stack = 0;
    printf("data: %p text: %p stack: %p heap: %p
",
           &data, (void*)main, &stack, heap);
    return 0;
}
```

When ASLR is turned on, running this program outputs different addresses on each run. For example, here are four runs on OS X 10.9.3:

```
data: 0x10a532020 text: 0x10a531ed0 stack: 0x7fff556ceb1c heap: 0x7f9f80c03980
data: 0x10d781020 text: 0x10d780ed0 stack: 0x7fff5247fb1c heap: 0x7fe498c03980
data: 0x10164b020 text: 0x10164aed0 stack: 0x7fff556beb1c heap: 0x7fb783c03980
data: 0x10c7f8020 text: 0x10c7f7ed0 stack: 0x7fff53408b1c heap: 0x7f9740403980
```

By contrast, here are four runs on FreeBSD 8.3:

```
data: 0x8049728 text: 0x8048470 stack: 0xbfbfebfc heap: 0x28201088
data: 0x8049728 text: 0x8048470 stack: 0xbfbfebfc heap: 0x28201088
data: 0x8049728 text: 0x8048470 stack: 0xbfbfebfc heap: 0x28201088
data: 0x8049728 text: 0x8048470 stack: 0xbfbfebfc heap: 0x28201088
```

Here’s the situation on each of the operating systems supported by the MPS:

- **FreeBSD** (as of version 10.0) does not support ASLR, so there’s nothing to do.

- On **Windows** (Vista or later), ASLR is a property of the executable, and it can be turned off at link time using the `/DYNAMICBASE:NO` linker option.

- On **Linux** (kernel version 2.6.12 or later), ASLR can be turned off for a single process by running `setarch` with the `-R` option:
  ```
  setarch --addr-no-randomize
  Disables randomization of the virtual address space
  ```

  For example:
  ```
  $ setarch $(uname -m) -R ./myprogram
  ```

- On **OS X** (10.7 or later), ASLR can be disabled for a single process by starting the process using `posix_spawn()`, passing the undocumented attribute `0x100`, like this:
  ```c
  #include <spawn.h>

  pid_t pid;
  posix_spawnattr_t attr;

  posix_spawnattr_init(&attr);
  posix_spawnattr_setflags(&attr, 0x100);
  posix_spawn(&pid, argv[0], NULL, &attr, argv, environ);
  ```

  The MPS provides the source code for a command-line tool implementing this (`tool/noaslr.c`). We’ve confirmed that this works on OS X 10.9.3, but since the technique is undocumented, it may well break in future releases. (If you know of a documented way to achieve this, please contact us.)
1.5.3 Example: underscanning

An easy mistake to make is to omit to fix a reference when scanning a formatted object. For example, in the Scheme interpreter’s scan method, I might have forgotten to fix the first element of a pair:

```
case TYPE_PAIR:
    /* oops, forgot: FIX(CAR(obj)); */
    FIX(CDR(obj));
    base = (char *)base + ALIGN_OBJ(sizeof(pair_s));
    break;
```

This means that as far as the MPS is concerned, the first element of the pair is unreachable and so dead, so after collecting the region of memory containing this object, the space will be reused for other objects. So CAR(obj) might end up pointing to the start of a valid object (but the wrong one), or to the middle of a valid object, or to an unused region of memory, or into an MPS internal control structure.

The reproducible test case is simple. Run a garbage collection by calling (gc) and then evaluate any expression:

```
$ gdb ./scheme
GNU gdb 6.3.50-20050815 (Apple version gdb-1820) (Sat Jun 16 02:40:11 UTC 2012)
(gdb) run
Starting program: example/scheme/scheme
Reading symbols for shared libraries +............................. done
MPS Toy Scheme Example
7944, 0> (gc)
Collection started.
    Why: Client requests: immediate full collection.
    Clock: 11357
Collection finished.
    live 1888
    condemned 7968
    not_condemned 0
    clock: 12008
7968, 1> foo
Assertion failed: (TYPE(frame) == TYPE_PAIR), function lookup_in_frame, file scheme.c, line 1065.
```

Program received signal SIGABRT, Aborted.
0x0000000000000000 in __kill ()

What’s going on?

```
(gdb) backtrace
#0 0x0000000000000000 in __kill ()
#1 0x0000000000000000 in abort ()
#2 0x0000000000000000 in __assert_rtn ()
#0 0x0000000000000000 in lookup_in_frame (frame=0x1003fa7d0, symbol=0x1003faf20) at scheme.c:1066
#2 0x0000000000000000 in eval (env=0x1003fb130, exp=0x1003faf20) at scheme.c:1135
#3 0x0000000000000000
#4 0x0000000000000000 in main (argc=1, argv=0x7fff5fbff830) at scheme.c:3314
(gdb) frame 4
#0 0x0000000000000000 in lookup (env=0x1003fb130, symbol=0x1003faf20) at scheme.c:1087
#2 0x0000000000000000 in base (obj) at scheme.c:3314
```

The backtrace shows that the interpreter is in the middle of looking up the symbol foo in the environment. The Scheme interpreter implements the environment as a list of frames, each of which is a list of bindings, each binding
being a pair of a symbol and its value, as shown here:

![Fig. 1.3: The environment data structure in the Scheme interpreter.](image)

In this case, because the evaluation is taking place at top level, there is only one frame in the environment (the global frame). And it’s this frame that’s corrupt:

```
(gdb) frame 3
#3 0x0000000100003f55 in lookup_in_frame (frame=0x1003fa7d0, symbol=0x1003faf20) at scheme.c:1066
1066 assert(TYPE(frame) == TYPE_PAIR);
(gdb) list
1061 */
1062
1063 static obj_t lookup_in_frame(obj_t frame, obj_t symbol)
1064 {
1065 while(frame != obj_empty) {
1066     assert(TYPE(frame) == TYPE_PAIR);
1067     assert(TYPE(CAR(frame)) == TYPE_PAIR);
1068     assert(TYPE(CAAR(frame)) == TYPE_SYMBOL);
1069     if(CAAR(frame) == symbol)
1070         return CAR(frame);
(gdb) print frame->type.type
$2 = 13
```

The number 13 is the value TYPE_PAD. So instead of the expected pair, frame points to a *padding object*.

You might guess at this point that the frame had not been fixed, and since you know that the frame is referenced by the *car* of the first pair in the environment, that’s the suspect reference. But in a more complex situation this might not yet be clear. In such a situation it can be useful to look at the sequence of events leading up to the detection of the error. See *Telemetry*.

### 1.5.4 Example: allocating with wrong size

Here’s another kind of mistake: an off-by-one error in `make_string` leading to the allocation of string objects with the wrong size:

```
static obj_t make_string(size_t length, char *string)
{
    obj_t obj;
    mps_addr_t addr;
    size_t size = ALIGN_OBJ offsetof(string_s, string) + length/* oops, forgot: +1 */;
    do {
        mps_res_t res = mps_reserve(&addr, obj_ap, size);
        if (res != MPS_RES_OK) error("out of memory in make_string");
        obj = addr;
        obj->string.type = TYPE_STRING;
        obj->string.length = length;
        if (string) memcpy(obj->string.string, string, length+1);
        else memset(obj->string.string, 0, length+1);
    } while(!mps_commit(obj_ap, addr, size));
    total += size;
    return obj;
}
```

Here’s a test case that exercises this bug:
Memory Pool System Documentation, Release 1.115.0

```scheme
(define (church n f a) (if (eqv? n 0) a (church (- n 1) f (f a))))
(church 1000 (lambda (s) (string-append s "x")) "")
```

And here's how it shows up in the debugger:

```
$ gdb ./scheme
GNU gdb 6.3.50-20050815 (Apple version gdb-1820) (Sat Jun 16 02:40:11 UTC 2012)
[...]
(gdb) run < test.scm
Starting program: example/scheme/scheme < test.scm
Reading symbols for shared libraries ................................ done
MPS Toy Scheme Example
[...]
9960, 0> church
Assertion failed: (0), function obj_skip, file scheme.c, line 2940.
10816, 0>
Program received signal SIGABRT, Aborted.
0x00007fff91adeed46 in __kill ()
(gdb) backtrace
#0 0x00007fff91adeed46 in __kill ()
#1 0x00007fff90509df0 in abort ()
#2 0x00000001000014e3 in obj_skip (base=0x1003f9b88) at scheme.c:2940
#3 0x0000000100001f2d2 in TracePoll1 (globals=0x10012a000) at trace.c:1981
#4 0x000000010000d75f in ArenaPoll (globals=0x10012a000) at global.c:684
#5 0x0000000100009a40 in mps_ap_fill (p_o=0x7fff5fbff3e0, mps_ap=0x1003fe820, size=208) at mpsi.c:961
#6 0x000000010000447d in make_string (length=190, string=0x0) at scheme.c:468
#7 0x0000000100000497 in list (frame 3)
#8 0x00000001000014e3 in obj_skip (base=0x1003f9b88) at scheme.c:2940
```

The object being skipped is corrupt:
What happened to it? It’s often helpful in these situations to have a look at nearby memory.

You can see that this is a block containing mostly pairs (which have tag 0 and consist of three words), though you can see an operator (with tag 4) near the bottom. But what’s that at the start of the block, where obj’s tag should be? It looks like a pointer. So what’s in the memory just below obj? Let’s look at the previous few words:

Yes: there’s a pair (with tag 0) at 0x1003f9b80. So it looks as though the previous object was allocated with one size, but skipped with a different size. The previous object being the string (with tag 5) at 0x1003f9b70 which has length 0 and so is three words long as far as obj_skip is concerned:

but the next object (the pair) was clearly allocated at 0x1003f9b80 (overwriting the last word of the string), so the string must have been allocated with a size of only two words. This should be enough evidence to track down the cause.

1.5.5 What next?

If you tracked down all your bugs, then the next step is the chapter Tuning the Memory Pool System for performance. But if you’re still struggling, please contact us and see if we can help.

1.6 Tuning the Memory Pool System for performance

Note
When developing a benchmark to profile your program against, bear in mind that the benchmark should allocate several times the amount of physical memory that you expect to be available to the process. If the total allocation fits into the available memory, there’s no point running a garbage collector at all: you might as well just allocate and never collect.
The most important aspect of tuning the MPS is to choose good sizes for the *generations* in your *generation chain*. The ideal size of a generation should be such that when it is collected, most of the blocks allocated in that generation should be found to be *dead* (and so the cost of *scanning* and *copying* them can be avoided entirely). If a generation is collected when its blocks are mostly alive, that is a waste of time.

In the tables below I give the execution time of *test-leaf.scm* in the toy Scheme interpreter under different settings for its generation chain. (This test case allocates hundreds of millions of small short-lived objects.)

First, the effect of varying the capacity of a chain with a single generation.

<table>
<thead>
<tr>
<th>Capacity</th>
<th>Mortality</th>
<th>Execution time (user+sys)</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>0.80</td>
<td>362.6</td>
</tr>
<tr>
<td>200</td>
<td>0.80</td>
<td>354.9</td>
</tr>
<tr>
<td>400</td>
<td>0.80</td>
<td>349.7</td>
</tr>
<tr>
<td>800</td>
<td>0.80</td>
<td>314.4</td>
</tr>
<tr>
<td>1600</td>
<td>0.80</td>
<td>215.7</td>
</tr>
<tr>
<td>3200</td>
<td>0.80</td>
<td>94.0</td>
</tr>
<tr>
<td>6400</td>
<td>0.80</td>
<td>53.5</td>
</tr>
<tr>
<td>12800</td>
<td>0.80</td>
<td>79.6</td>
</tr>
<tr>
<td>25600</td>
<td>0.80</td>
<td>77.6</td>
</tr>
</tbody>
</table>

Second, the effect of varying the mortality of a chain with a single generation.

<table>
<thead>
<tr>
<th>Capacity</th>
<th>Mortality</th>
<th>Execution time (user+sys)</th>
</tr>
</thead>
<tbody>
<tr>
<td>6400</td>
<td>0.20</td>
<td>55.4</td>
</tr>
<tr>
<td>6400</td>
<td>0.40</td>
<td>54.0</td>
</tr>
<tr>
<td>6400</td>
<td>0.60</td>
<td>54.0</td>
</tr>
<tr>
<td>6400</td>
<td>0.80</td>
<td>53.5</td>
</tr>
<tr>
<td>6400</td>
<td>0.99</td>
<td>54.8</td>
</tr>
</tbody>
</table>

Third, the effect of varying the number of generations (all generations being identical).

<table>
<thead>
<tr>
<th>Generations</th>
<th>Capacity</th>
<th>Mortality</th>
<th>Execution time (user+sys)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>6400</td>
<td>0.80</td>
<td>53.5</td>
</tr>
<tr>
<td>2</td>
<td>6400</td>
<td>0.80</td>
<td>42.4</td>
</tr>
<tr>
<td>3</td>
<td>6400</td>
<td>0.80</td>
<td>42.1</td>
</tr>
<tr>
<td>4</td>
<td>6400</td>
<td>0.80</td>
<td>42.2</td>
</tr>
<tr>
<td>5</td>
<td>6400</td>
<td>0.80</td>
<td>42.2</td>
</tr>
</tbody>
</table>

These tables suggest that:

1. The improvement in performance to be gained by getting generation sizes right is dramatic: much bigger than the small improvements to gained from other techniques.
2. The predicted mortality doesn’t make much difference to the overall execution time (it does affect the distribution of pause times, however: see *Scheduling of collections*.)
3. You can make generations too big as well as too small.
4. There are rapidly diminishing returns to be gained from adding generations.

**Note**

*Telemetry* can be used to discover when generations are being collected and what proportion of blocks were found to be alive.

The table below shows the effect of varying the initial allocation of address space to the arena (using three generations each with capacity 6400 kB, mortality 0.80).
### 1.7 Advanced topics

#### 1.7.1 Finalization

In Scheme, an open file is represented by a `port`. In the toy Scheme interpreter, a port is a wrapper around a standard C file handle:

```c
typedef struct port_s {
    type_t type; /* TYPE_PORT */
    obj_t name;  /* name of stream */
    FILE *stream;
} port_s;
```

Operating systems limit the number of files that a process can have open simultaneously, so to avoid running out of file handles, it is necessary to close ports when you are done with them. If a Scheme program fails to call

---

1 With this initial allocation of address space, the test case failed to run to completion after thousands of seconds and tens of thousands of garbage collection cycles.

---

<table>
<thead>
<tr>
<th>Address space</th>
<th>Extensions</th>
<th>Collections</th>
<th>Execution time (user+sys)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>32</td>
<td>371</td>
<td>52.0</td>
</tr>
<tr>
<td>4</td>
<td>21</td>
<td>370</td>
<td>47.0</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>0</td>
<td>2436</td>
<td>160.5</td>
</tr>
<tr>
<td>18</td>
<td>0</td>
<td>1135</td>
<td>89.1</td>
</tr>
<tr>
<td>20</td>
<td>0</td>
<td>673</td>
<td>60.6</td>
</tr>
<tr>
<td>22</td>
<td>0</td>
<td>484</td>
<td>48.7</td>
</tr>
<tr>
<td>24</td>
<td>0</td>
<td>400</td>
<td>43.1</td>
</tr>
<tr>
<td>32</td>
<td>0</td>
<td>368</td>
<td>41.2</td>
</tr>
<tr>
<td>64</td>
<td>0</td>
<td>368</td>
<td>43.1</td>
</tr>
<tr>
<td>128</td>
<td>0</td>
<td>368</td>
<td>46.4</td>
</tr>
<tr>
<td>256</td>
<td>0</td>
<td>368</td>
<td>46.3</td>
</tr>
<tr>
<td>512</td>
<td>0</td>
<td>368</td>
<td>49.3</td>
</tr>
<tr>
<td>1024</td>
<td>0</td>
<td>368</td>
<td>42.0</td>
</tr>
<tr>
<td>2048</td>
<td>0</td>
<td>368</td>
<td>43.2</td>
</tr>
<tr>
<td>4096</td>
<td>0</td>
<td>368</td>
<td>43.5</td>
</tr>
<tr>
<td>8192</td>
<td>0</td>
<td>368</td>
<td>46.1</td>
</tr>
<tr>
<td>16384</td>
<td>0</td>
<td>368</td>
<td>49.2</td>
</tr>
<tr>
<td>32768</td>
<td>0</td>
<td>368</td>
<td>57.1</td>
</tr>
<tr>
<td>65536</td>
<td>0</td>
<td>368</td>
<td>71.1</td>
</tr>
<tr>
<td>131072</td>
<td>0</td>
<td>368</td>
<td>101.3</td>
</tr>
<tr>
<td>262144</td>
<td>0</td>
<td>368</td>
<td>161.3</td>
</tr>
<tr>
<td>524288</td>
<td>0</td>
<td>368</td>
<td>273.0</td>
</tr>
<tr>
<td>1048576</td>
<td>0</td>
<td>368</td>
<td>504.6</td>
</tr>
</tbody>
</table>

---

**Note**

The lesson here is that the allocation of address space has to be comfortably larger than the working set of the program, but that a very large address space is ruinous to performance.
close-input-file, then the underlying file handle should still be closed when the port object dies. This procedure is known as finalization.

Note
It’s generally a bad idea to depend on finalization to release your resources (see the Cautions section in Finalization). Treat it as a last resort when more reliable mechanisms for releasing resources (like Scheme’s with-open-input-file) aren’t available.

Any block in an automatically managed pool can be registered for finalization by calling mps_finalize(). In the toy Scheme interpreter, this can be done in make_port:

```c
static obj_t make_port(obj_t name, FILE *stream)
{
    mps_addr_t port_ref;
    obj_t obj;
    mps_addr_t addr;
    size_t size = ALIGN_OBJ(sizeof(port_s));
    do {
        mps_res_t res = mps_reserve(&addr, obj_ap, size);
        if (res != MPS_RES_OK) error("out of memory in make_port");
        obj = addr;
        obj->port.type = TYPE_PORT;
        obj->port.name = name;
        obj->port.stream = stream;
    } while(!mps_commit(obj_ap, addr, size));
    total += sizeof(port_s);
    port_ref = obj;
    mps_finalize(arena, &port_ref);
    return obj;
}
```

The MPS implements finalization by posting a message to the arena’s message queue when an object that has been registered for finalization is about to die.

If you want to finalize your objects, you must first enable finalization messages by calling mps_message_type_enable():

```c
mp_message_type_enable(arena, mps_message_type_finalization());
```

You must then poll the arena’s message queue at times that are convenient for you, call mps_message_get() to pick up a finalization message from the queue, call mps_message_finalization_ref() to access the finalized object, and finally call mps_message_discard() on the finalization message. The finalized object is then subject to the normal rules of life and death: it continues to live as long as it is strongly reachable.

In the toy Scheme interpreter, the most convenient moment to process the message queue is at the start of the read–eval–print loop. When a finalization message is found, the associated file handle is closed (unless it has been closed already), and the message is discarded.

```c
mp_message_type_t type;
while (mps_message_queue_type(type, arena)) {
    mps_message_t message;
    mps_bool_t b;
    b = mps_message_get(&message, arena, type);
    assert(b); /* we just checked there was one */
```
if (type == mps_message_type_finalization()) {
    mps_addr_t port_ref;
    obj_t port;
    mps_message_finalization_ref(&port_ref, arena, message);
    port = port_ref;
    assert(TYPE(port) == TYPE_PORT);
    if(port->port.stream) {
        printf("Port to file "\%s" is dying. Closing file.\n",
            port->port.name->string.string);
        (void)fclose(port->port.stream);
        port->port.stream = NULL;
    }
    else {
        /* ... handle other message types ... */
    }
    mps_message_discard(arena, message);
}

Here’s an example session showing finalization taking place:

MPS Toy Scheme Example
9960, 0> (open-input-file "scheme.c")
#[port "scheme.c"]
10064, 0> (gc)
Collection started.
Why: Client requests: immediate full collection.
Clock: 3401
Port to file "scheme.c" is dying. Closing file.
Collection finished.
live 10040
condemned 10088
not_condemned 0
clock: 3807

The toy Scheme interpreter definalizes ports by calling mps_definalize() when they are closed. This is purely an optimization: setting stream to NULL ensures that the file handle wouldn’t be closed more than once, even if the port object were later finalized.

static void port_close(obj_t port)
{
    assert(TYPE(port) == TYPE_PORT);
    if(port->port.stream != NULL) {
        mps_addr_t port_ref = port;
        fclose(port->port.stream);
        port->port.stream = NULL;
        mps_definalize(arena, &port_ref);
    }
}

It’s still possible that the toy Scheme interpreter might run out of open file handles despite having some or all of its port objects being finalizable. That’s because the arena’s message queue is only polled after evaluating an expression at top level: if the expression itself opens too many file handles, the finalization messages will queue up and not be processed in time. For example:

MPS Toy Scheme Example
9960, 0> (define (repeat n f _) (if (eqv? n 0) '() (repeat (- n 1) f (f))))
repeat
10840, 0> (repeat 300 (lambda () (open-input-file "scheme.c")) 0)

1.7. Advanced topics
A less naïve interpreter might process finalization messages on a more regular schedule, or might take emergency action in the event of running out of open file handles by carrying out a full garbage collection and processing any finalization messages that are posted as a result.

Topics

Finalization, Messages.

1.7.2 Location dependency

The toy Scheme interpreter contains an address-based (eq?) hash table implementation. It hashes the addresses of its keys, and so needs to take account of the possibility that a moving garbage collector might move the keys. If it fails to take account of this, the hash table might become invalid after a garbage collection.

In the interaction shown below (with a naïve version of the code) you’ll see that although the keys remain present in the table after garbage collection, they cannot be found. This is because their locations (and hence their hashes) have changed, but their positions in the table have not been updated to match.

MPS Toy Scheme Example

```
10240, 0> (define ht (make-eq-hashtable))
ht
10584, 0> (hashtable-set! ht 'one 1)
10768, 0> (hashtable-set! ht 'two 2)
10952, 0> (hashtable-set! ht 'three 3)
11136, 0> ht
#[hashtable (two 2) (three 3) (one 1)]
11136, 0> (hashtable-ref ht 'two #f)
2
11280, 0> (gc)
11304, 1> (hashtable-ref ht 'one #f)
#f
11448, 1> (hashtable-ref ht 'two #f)
#f
11592, 1> (hashtable-ref ht 'three #f)
#f
11736, 1> ht
#[hashtable (two 2) (three 3) (one 1)]
```

The MPS solves this problem with its location dependency feature: a structure of type mps_ld_s encapsulates a set of dependencies on the locations of blocks. You add addresses to the location dependency, and then later test an address to see if it is stale: that is, if the block at that address might have moved since its location was depended upon.

You need to provide space for the mps ld s structure. In the case of a hash table, it is most convenient to inline it in the hash table’s metadata:

```c
typedef struct table_s {
    type_t type; /* TYPE_TABLE */
    hash_t hash; /* hash function */
    cmp_t cmp; /* comparison function */
    mps ld_s ld; /* location dependency */
    obj_t buckets; /* hash buckets */
} table_s;
```

Before being used, the location dependency must be reset to indicate that nothing is depended upon, by calling mps ld reset().
For example:

```c
static obj_t make_table(size_t length, hash_t hashf, cmp_t cmpf)
{
    obj_t obj;
    mps_addr_t addr;
    size_t l, size = ALIGN_OBJ(sizeof(table_s));
    do {
        mps_res_t res = mps_reserve(&addr, obj_ap, size);
        if (res != MPS_RES_OK) error("out of memory in make_table");
        obj = addr;
        obj->table.type = TYPE_TABLE;
        obj->table.buckets = NULL;
    } while (!mps_commit(obj_ap, addr, size));
    total += size;
    obj->table.hash = hashf;
    obj->table.cmp = cmpf;
    /* round up to next power of 2 */
    for (l = 1; l < length; l *= 2);
    obj->table.buckets = make_buckets(l);
    mps_ld_reset(&obj->table.ld, arena);
    return obj;
}
```

Before the hash table becomes dependent on the location of a block, the address of the block must be added to its location dependency by calling `mps_ld_add()`. In particular, you must call `mps_ld_add()` before computing the hash of the address. (If you wait until afterwards, it might be too late: a garbage collection might have taken place after the hash was computed but before you added the dependency.)

In the toy Scheme interpreter, this is done just before the computation of the hash of the address.

```c
static unsigned long eq_hash(obj_t obj, mps_ld_t ld)
{
    union {
        char s[sizeof(obj_t)];
        obj_t addr; } u;
    if (ld) mps_ld_add(ld, arena, obj);
    u.addr = obj;
    return hash(u.s, sizeof(obj_t));
}
```

By adding the dependency at this point in the code, the implementation avoids adding unnecessary dependencies on a location. For example, an `eqv?` hash table does not need to depend on the location of numbers and characters:

```c
static unsigned long eqv_hash(obj_t obj, mps_ld_t ld)
{
    switch (TYPE(obj)) {
        case TYPE_INTEGER:
            return obj->integer.integer;
        case TYPE_CHARACTER:
            return obj->character.c;
        default:
            return eq_hash(obj, ld);
    }
}
```

and a `string=?` hash table does not need to depend on the location of any of its keys.

**Note**
The garbage collector may run at any time, so a key may become stale at any time after calling `mps_ld_add()`, perhaps even before you’ve added it!

1.7. Advanced topics
It’s best to postpone worrying about this until this key is actually looked up, when the staleness will be discovered. After all, it may never be looked up.

If you look up a key in an address-based hash table and fail to find it there, that might be because the table’s dependency on the location of the key is stale: that is, if the garbage collector moved the key. The function `mps_ld_isstale()` tells you if a block whose location you depended upon since the last call to `mps_ld_reset()` might have moved.

In the toy Scheme interpreter this behaviour is encapsulated into `table_find`:

```c
static struct bucket_s *table_find(obj_t tbl, obj_t key, int add)
{
  struct bucket_s *b;
  assert(TYPE(tbl) == TYPE_TABLE);
  b = buckets_find(tbl, tbl->table.buckets, key, add);
  if ((b == NULL || b->key == NULL || b->key == obj_deleted)
      && mps_ld_isstale(&tbl->table.ld, arena, key))
    b = table_rehash(tbl, tbl->table.buckets->buckets.length, key);
  return b;
}
```

It’s important to test `mps_ld_isstale()` only in case of failure. The function may report a false positive (returning true despite the block not having moved). So if `key` has not moved, then if you tested `mps_ld_isstale()` first, it might return true and so you’d end up unnecessarily rehashing the whole table. (It’s crucial, however, to actually test that `key` appears in the table, not just that some key with the same hash does.)

When a table is rehashed, call `mps_ld_reset()` to clear the location dependency, and then `mps_ld_add()` for each key before it is added back to the table.

**Note**

After `mps_ld_isstale()` has returned true, and after rehashing the table, I don’t just repeat the usual lookup by calling `buckets_find`. That’s because the table might have become stale again already.

Instead, `table_rehash` finds and returns the bucket containing `key`. (Since it has to loop over all the entries in the table anyway, it might as well find this bucket too.)

By adding the line:

```c
puts("stale!");
```

in `table_find` after `mps_ld_isstale()` returns true, it’s possible to see when the location dependency becomes stale and the table has to be rehashed:

MPS Toy Scheme Example
10240, 0> (define ht (make-eq-hashtable))
10584, 0> (hashtable-set! ht 'one 1)
10768, 0> ht
10792, 1> (gc)
11080, 1> (hashtable-ref ht 'one #f)
stale! 1
11264, 1> (hashtable-set! ht 'two 2)
11288, 2> (gc) stale!
Note

In case you’re puzzled by the highlighted lines: the symbol ‘one’ must not have been moved by the collection, and so was found in the table at the correct location. Thus `mps_id_isstale()` was not called. The symbol ‘two’ did move in the collection, so it’s not found in the table, and that causes `mps_id_isstale()` to be tested.

Don’t forget to check the location dependency for staleness when setting a value for key in a hash table, and when deleting a key from a hash table. Here’s an interaction with the toy Scheme interpreter showing a key being found to be stale when setting and when deleting it:

MPS Toy Scheme Example
```
13248, 0> (define ht (make-eq-hashtable))
ht
13624, 0> (hashtable-set! ht 'a 1)
13808, 0> (gc)
13832, 1> (hashtable-set! ht 'a 2)
stale!
13832, 1> (hashtable-delete! ht 'one)
stale!
14152, 1> (gc)
14176, 2> (hashtable-delete! ht 'a)
stale!
14456, 2> ht
#|hashtable|
```

Topic

Location dependency.

### 1.7.3 Weak hash tables

A weak-key hash table has weak references (1) to its keys. If the key dies, the value corresponding to that key is automatically deleted from the table too. Similarly, a weak-value hash table has weak references to its values, and a doubly weak hash table has weak references to both.

In this section, I’ll describe how to add all three types of weak hash table to the toy Scheme interpreter. This requires a few far-reaching changes to the code, so in order to keep the basic integration understandable by newcomers to the MPS, I’ve made these changes in a separate version of the code:

```
scheme-advanced.c
```

The Scheme interpreter after a number of “advanced” features, including weak hash tables, have been implemented.

### 1.7. Advanced topics
The MPS supports weak references only in *roots* and in blocks allocated in pools belonging to the *AWL (Automatic Weak Linked)* pool class. Roots aren’t convenient for this use case: it’s necessary for hash tables to be automatically reclaimed when they die. So AWL it is.

**Note**

This isn’t a design limitation of the MPS: it’s just that up until now the only uses our customers have had for weak references are the ones supported by AWL. (In particular, AWL was designed around the requirements of weak hash tables in Open Dylan.) If you need more general handling of weak references, contact us.

All the references in a *formatted object* belong to the same *rank*: that is, they are all *exact*, *weak*, or *ambiguous references*. In AWL, the rank of references is specified when creating an *allocation point*. This has consequences for the design of the hash table data structure: in weak-key strong-value hash tables, the keys need to be in one object and the values in another (and the same is true in the strong-key weak-value case). So instead of having one vector of buckets with alternate keys and values, hash tables must have two vectors, one for the keys and the other for the values, to allow keys and values to have different ranks.

These vectors will be allocated from an AWL pool with two allocation points, one for strong references, and one for weak references:

```c
static mps_pool_t buckets_pool; /* pool for hash table buckets */
static mps_ap_t strong_buckets_ap; /* allocation point for strong buckets */
static mps_ap_t weak_buckets_ap; /* allocation point for weak buckets */
```

**Note**

It’s not necessary to allocate the strong buckets from the same pool as the weak buckets, but we’ll see below that they have to be allocated in a *non-moving* pool such as AWL.

The MPS *splats* a weak reference in a *formatted object* by replacing it with a null pointer when it is *fixed* by the object format’s *scan method*. So the scan method for the buckets is going to have the following structure. (See below for the actual code.)

```c
static mps_res_t buckets_scan(mps_ss_t ss, mps_addr_t base, mps_addr_t limit)
{
    MPS_SCAN_BEGIN(ss) {
        while (base < limit) {
            buckets_t buckets = base;
            size_t length = buckets->length;
            for (i = 0; i < length; ++i) {
                mps_addr_t p = buckets->bucket[i];
                if (MPS_FIX1(ss, p)) {
                    mps_res_t res = MPS_FIX2(ss, &p);
                    if (res != MPS_RES_OK) return res;
                    if (p == NULL) {
                        /* TODO: key/value was splatted: splat value/key too */
                    }
                    buckets->bucket[i] = p;
                }
            }
            base = (char *)base +
            ALIGN_OBJ(offsetof(buckets_s, bucket) +
                length * sizeof(buckets->bucket[0]));
        }
    } MPS_SCAN_END(ss);
    return MPS_RES_OK;
}
```
But how can the corresponding key/value be splatted? A format method is not normally allowed to access memory managed by the MPS in pools that might protect their objects (see the Cautions section in Object formats). The AWL pool class relaxes this constraint by allowing each object in the pool to have a dependent object. When scanning an object in an AWL pool, the MPS ensures that the dependent object is not protected. The dependent object does not have to be in the same pool as the original object, but must be in a non-moving pool. See Dependent objects.

So the value buckets will be the dependent object of the key buckets, and vice versa.

The AWL pool determines an object’s dependent object by calling a function that you supply when creating the pool. This means that each object needs to have a reference to its dependent object:

```c
static mps_addr_t buckets_find_dependent(mps_addr_t addr)
{
    buckets_t buckets = addr;
    return buckets->dependent;
}
```

There’s one final requirement to take into account before revealing the new buckets structure, which is that each word in an object in an AWL pool must either be a valid word-aligned reference, or else the bottom bits of the word must be non-zero so that it does not look like an aligned pointer. So the sizes stored in the buckets structure (the length of the array of buckets, and the counts of used and deleted buckets) must be tagged so that they cannot be mistaken for pointers. See the Caution section in AWL (Automatic Weak Linked).

A one-bit tag suffices here:

```c
#define TAG_COUNT(i) (((i) << 1) + 1)
#define UNTAG_COUNT(i) ((i) >> 1)

typedef struct buckets_s {
    struct buckets_s *dependent; /* the dependent object */
    size_t length; /* number of buckets (tagged) */
    size_t used; /* number of buckets in use (tagged) */
    size_t deleted; /* number of deleted buckets (tagged) */
    obj_t bucket[1]; /* hash buckets */
} buckets_s, *buckets_t;
```

Now the full details of the scan method can be given, with the revised code highlighted:

```c
static mps_res_t buckets_scan(mps_ss_t ss, mps_addr_t base, mps_addr_t limit)
{
    MPS_SCAN_BEGIN(ss) {
        while (base < limit) {
            buckets_t buckets = base; /* see note 1 */
            size_t i, length = UNTAG_COUNT(buckets->length);
            FIX(buckets->dependent);
            if (buckets->dependent != NULL)
                assert(buckets->dependent->length == buckets->length);
            for (i = 0; i < length; ++i) {
                mps_addr_t p = buckets->bucket[i];
                if (MPS_FIX1(ss, p)) {
                    mps_res_t res = MPS_FIX2(ss, &p);
                    if (res != MPS_RES_OK) return res;
                    if (p == NULL) {
                        /* key/value was splatted: splat value/key too */
                        p = obj_deleted; /* see note 3 */
                        buckets->deleted = TAG_COUNT(UNTAG_COUNT(buckets->deleted) + 1);
                        if (buckets->dependent != NULL) { /* see note 2 */
                            buckets->dependent->bucket[i] = p;
                            buckets->dependent->deleted
                                = TAG_COUNT(UNTAG_COUNT(buckets->dependent->deleted) + 1);
                        }
                    }
                }
            }
        }
    }
}
```
buckets->bucket[i] = p;

base = (char *)base + ALIGN_OBJ(offsetof(buckets_s, bucket) +
    length * sizeof(buckets->bucket[0]));

MPS_SCAN_END(ss);
return MPS_RES_OK;

Notes

1. There’s no need to dispatch on the type of the buckets object (or even to store a type at all) because buckets are
the only objects to be stored in this pool.
2. The dependent object must be fixed, and because the reference to it might be weak, it might be splatted. This
means that even if you are confident that you will always initialize this field, you still have to guard access to it,
as here.
3. This hash table implementation uses NULL to mean “never used” and obj_deleted to mean “formerly used
but then deleted”. So when a key is splatted it is necessary to replace it with obj_deleted.

The skip method is straightforward:

static mps_addr_t buckets_skip(mps_addr_t base)
{
    buckets_t buckets = base;
    size_t length = UNTAG_SIZE(buckets->length);
    return (char *)base + ALIGN_OBJ(offsetof(buckets_s, bucket) +
        length * sizeof(buckets->bucket[0]));
}

Now we can create the object format, the pool and the allocation points:

/* Create the buckets format. */
MPS_ARGS_BEGIN(args) {
    MPS_ARGS_ADD(args, MPS_KEY_FMT_ALIGN, ALIGNMENT);
    MPS_ARGS_ADD(args, MPS_KEY_FMT_SCAN, buckets_scan);
    MPS_ARGS_ADD(args, MPS_KEY_FMT_SKIP, buckets_skip);
    res = mps_fmt_create_k(&buckets_fmt, arena, args);
} MPS_ARGS_END(args);
if (res != MPS_RES_OK) error("Couldn’t create buckets format");

/* Create an Automatic Weak Linked (AWL) pool to manage the hash table
buckets. */
MPS_ARGS_BEGIN(args) {
    MPS_ARGS_ADD(args, MPS_KEY_FORMAT, buckets_fmt);
    MPS_ARGS_ADD(args, MPS_KEY_AWL_FIND_DEPENDENT, buckets_find_dependent);
    res = mps_pool_create_k(&buckets_pool, arena, mps_class_awl(), args);
} MPS_ARGS_END(args);
if (res != MPS_RES_OK) error("Couldn’t create buckets pool");

/* Create allocation points for weak and strong buckets. */
MPS_ARGS_BEGIN(args) {
    MPS_ARGS_ADD(args, MPS_KEY_RANK, mps_rank_exact());
    res = mps_ap_create_k(&strong_buckets_ap, buckets_pool, args);
} MPS_ARGS_END(args);
if (res != MPS_RES_OK) error("Couldn’t create strong buckets allocation point");
MPS_ARGS_BEGIN(args) {
    MPS_ARGS_ADD(args, MPS_KEY_RANK, mps_rank_weak());
    res = mps_ap_create_k(&weak_buckets_ap, buckets_pool, args);
} MPS_ARGS_END(args);
if (res != MPS_RES_OK) error("Couldn’t create weak buckets allocation point");

By adding the line:
puts("splat!");

at the point in buckets_scan where the splatting of a weak reference is detected, we can see this happening:

MPS Toy Scheme Example
24624, 0> (define ht (make-doubly-weak-hashtable string-hash string=?))
ht
25264, 0> (hashtable-set! ht "one" 1)
25456, 0> (hashtable-set! ht "two" 2)
25648, 0> (hashtable-set! ht "three" 3)
25840, 0> ht
#\[hashtable ("two" 2) ("one" 1) ("three" 3)]
25864, 0> (gc)
splat!
splat!
splat!
25912, 1> ht
#\[hashtable]

Topics
Weak references, AWL (Automatic Weak Linked).

1.7.4 Global symbol table

In the original (non-MPS) version of the toy Scheme interpreter, the global symbol table was implemented as a key-
only hash table, and each symbol stored its own name.

But now that we have weak hash tables, it makes sense to re-implement the global symbol table as a strong-key weak-
value hash table mapping strings to symbols. Each symbol will now contain a reference to its name as a string object,
instead of containing the name itself.

Fig. 1.4: Global symbol table design (weak references shown as dashed lines).

This design depends on the string object containing the symbol name being immutable. As it happens, all strings are
immutable, because the toy Scheme interpreter doesn’t implement string-set!, but if it did then some care would
need to be taken. (Either by marking these strings as immutable in some way, or by ensuring that these strings are
“private”: that is, that Scheme programs never get hold of references to them.)

When there are no more strong references to a symbol:
1. the reference to the symbol from the “values” array may be splatted;
2. that’s detected by the buckets scan method, which deletes the corresponding entry in the “keys” array;
3. which may in turn cause the symbol name to die, unless there are other strong references keeping it alive.

Here’s the new symbol structure:

1.7. Advanced topics
typedef struct symbol_s {
    type_t type;    /* TYPE_SYMBOL */
    obj_t name;     /* its name (a string) */
} symbol_s;

and the new implementation of intern:

```c
static obj_t intern_string(obj_t name)
{
    obj_t symbol;
    assert(TYPE(name) == TYPE_STRING);
    symbol = table_ref(symtab, name);
    if(symbol == NULL) {
        symbol = make_symbol(name);
        table_set(symtab, name, symbol);
    }
    return symbol;
}

static obj_t intern(char *string)
{
    return intern_string(make_string(strlen(string), string));
}
```

The symbol table now becomes a very simple root, that only has to be registered once (not every time it is rehashed, as previously):

```c
mps_addr_t ref;
symtab = NULL;
ref = &symtab;
res = mps_root_create_table(&symtab_root, arena, mps_rank_exact(), 0,
                           ref, 1);
if(res != MPS_RES_OK) error("Couldn't register symtab root");
symtab = make_table(16, string_hash, string_equalp, 0, 1);
```

**Note**

The order of operations is important here. The global variable symtab must be registered as a root before creating the symbol table, otherwise the symbol table might be collected in the interval between creation and registration. But we must also ensure that symtab is valid (that is, scannable) before registering it (in this case, by setting it to NULL).

By printing splat! when the splatting of a weak reference is detected by the scan method, we can see when symbols are dying:

```scheme
MPS Toy Scheme Example
24624, 0> (define a 1)
a
24832, 0> '(a b c d)
(a b c d)
25144, 0> (gc)
splat! splat! splat!
```

Here, the symbols b, c and d died, but a was kept alive by the reference from the environment.
1.7.5 Segregation of objects

When objects of different types have different properties (different sizes, lifetimes, references, layouts) it makes sense to segregate them into pools of appropriate classes. The garbage collector in the MPS is designed to work efficiently with many pools: it traces references between objects in different pools, and it coordinates the scanning of the registers and control stacks (see Thread roots).

For example, the toy Scheme interpreter has a mixture of object types, some of which contain references to other objects (for example, pairs) that must be scanned, and some of which do not (for example, strings). If the leaf objects are segregated into a pool of an appropriate class, the cost of scanning them can be avoided.

Here the appropriate class is AMCZ (Automatic Mostly-Copying Zero-rank), and the necessary code changes are straightforward. First, global variables for the new pool and its allocation point:

```
static mps_pool_t leaf_pool; /* pool for leaf objects */
static mps_ap_t leaf_ap;    /* allocation point for leaf objects */
```

Second, the leaf objects must be allocated on leaf_ap instead of obj_ap. And third, the pool and its allocation point must be created:

```
/* Create an Automatic Mostly-Copying Zero-rank (AMCZ) pool to manage the leaf objects. */
MPS_ARGS_BEGIN(args) {
    MPS_ARGS_ADD(args, MPS_KEY_CHAIN, obj_chain);
    MPS_ARGS_ADD(args, MPS_KEY_FORMAT, obj_fmt);
    res = mps_pool_create_k(&leaf_pool, arena, mps_class_amcz(), args);
} MPS_ARGS_END(args);
if (res != MPS_RES_OK) error("Couldn't create leaf pool");

/* Create allocation point for leaf objects. */
res = mps_ap_create_k(&leaf_ap, leaf_pool, mps_args_none);
if (res != MPS_RES_OK) error("Couldn't create leaf objects allocation point");
```

Note that the new pool shared a generation chain with the old pool. This is important, because the leaf objects live and die along with the non-leaf objects of similar ages.

As an initial step in making this change, the new pool uses the same object format. However, we normally wouldn’t stop there: we’d take advantage of the segregation to simplify the scanning of the objects that have been left behind.

### Topic

AMCZ (Automatic Mostly-Copying Zero-rank).

1.8 Implementing malloc and free

The MPS function `mps_free()` is unlike the Standard C Library function `free()` in that it takes a size argument. That’s because it’s nearly always the case that either the size of a block is known statically based on its type (for example, a structure), or else the size of the block is easily computed from information that needs to be stored anyway (for example, a vector), and so memory can be saved by not storing the size separately. It’s also better for virtual memory performance, as a block does not have to be touched in order to free it.

But sometimes you need to interact with foreign code which requires `malloc()` and `free()` (or a pair of functions with the same interface). In this situation you can implement this interface using a global pool variable, and putting the size of each block into its header, like this:

```
```
#include "mps.h"

**static mps_pool_t malloc_pool;**

typedef union {
    size_t size;
    char alignment[MPS_PF_ALIGN]; /* see note below */
} header_u;

**void *malloc(size_t size) {
    mps_res_t res;
    mps_addr_t p;
    header_u *header;
    size += sizeof *header;
    res = mps_alloc(&p, malloc_pool, size);
    if (res != MPS_RES_OK)
        return NULL;
    header = p;
    header->size = size;
    return header + 1;
}

void free(void *p) {
    if (p) {
        header_u *header = ((header_u *)p) - 1;
        mps_free(malloc_pool, header, header->size);
    }
}

The alignment member of union header_u ensures that allocations are aligned to the platform’s natural alignment (see Alignment).

The pool needs to belong to a manually managed pool class, for example MVFF (Manual Variable First Fit) (or its debugging counterpart):

#include "mpscmvff.h"

**void malloc_pool_init(mps_arena_t arena) {
    mps_res_t res;
    res = mps_pool_create_k(&malloc_pool, arena, mps_class_mvff(), mps_args_none);
    if (res != RES_OK)
        abort();
}**
2.1 Interface conventions

This document describes the conventions used in the programming interface to the Memory Pool System. It also contains our policy for support for the public identifiers and definitions of general types that appear throughout the interface.

2.1.1 Support policy

1. We support the documented behaviour of public symbols in the MPS interface. We will only remove these symbols or change their behaviour in a new version, and not in a patch release. Normally we will give one version’s notice before removing a symbol or changing a particular documented behaviour: that is, there will be a version in which the symbol (or reliance on some of its behaviour) is deprecated.

Symbols may be deprecated in their old place in the reference manual, or they may be moved to the Deprecated interfaces chapter.

Note

If you are relying on a feature and you see that it’s deprecated, please contact us. It makes a difference if we know that someone is using a feature.

2. Behaviour that is not documented in the Guide, Reference, or Pool reference is not supported and may change without notice in future releases. In particular, private identifiers may disappear or their behaviour be changed without notice in future releases.

2.1.2 Language


2.1.3 Headers

1. The main interface is in the header mps.h. This header contains all the core MPS interfaces. In practice, you always need at least one arena class and one pool class header file as well.

2. We will always prefix public header file names with mps to avoid clashes. We reserve the right to create new headers with names beginning with mps in future versions.
3. **Pool class** headers have names beginning with `mpsc`. For example, the header for **AMC (Automatic Mostly-Copying)** is `mpscamc.h`.

4. **Arena class** headers have names beginning with `mpsa`. For example, the header for the virtual memory arena class is `mpsavm.h`.

### 2.1.4 Identifiers

1. Identifiers are in lower case, except for preprocessor constants and macros that do not behave like functions, which are in upper case. Words are joined by underscores.

2. All identifiers are either **public** or **private**.

3. The names of public types, functions, variables, and macros start with `mps_` or `MPS_`. The names of public structure members start with any letter.

4. Private identifiers start with an underscore `_`.

5. Type names end with `_t`, except for structure and union types.

6. The names of structure types and tags end with `_s`.

7. The names of union types and tags end with `_u`.

### 2.1.5 Types

There are three kinds of types declared in the MPS interface: **transparent types**, **opaque types**, and **derived types**.

1. A **transparent type** is an alias defined using `typedef`, and this is documented so that the client program can rely on that fact. For example, `mps_addr_t` is a transparent alias for `void *`. Transparent types express intentions in the interface: in the case of `mps_addr_t` it represents a pointer that is under the control of the MPS.

2. An **opaque type** is a pointer to an incomplete structure type. The client program must not rely on details of its implementation. For example, the type `mps_arena_t` is an alias for `struct mps_arena_s *`, but the implementation of `struct mps_arena_s` is not public.

   There are a few structure types that are declared in `mps.h` but whose implementation is not public. These only exist so that code can be inlined using macros. The most important of these is the **scan state** structure `mps_ss_s`, which is accessed by scanning macros such as `MPS_SCAN_BEGIN()` and `MPS_FIX12()`.

3. A **derived type** is a structure or function type based on transparent and opaque types and on built-in C types. The degree to which you may or must depend upon the implementation of a derived type is covered by the documentation for the type. For example, the structure type `mps_ap_s` has a mixture of public and private members.

### 2.1.6 Functions

1. Operations that might fail return a **result code**, rather than a “special value” of the return type. See **Error handling**.

2. A function that needs to return a value as well as a result code returns the value via an **out parameter**, a parameter that points to a location to store the result.

3. A function that stores a result in the location pointed to by an out parameter only does so if the function is successful (that is, if the function returns `MPS_RES_OK`).

4. The value in the location pointed to by an out parameter is not read by the function.
5. Out parameters have names ending with _o.

6. A function that both needs to read a value stored in a location and update the value does so via an in/out parameter, which is the same as an out parameter except that the location it points to is read by the function. See for example MPS_FIX12().

7. In/out parameters have names ending with _io.

8. A function that takes optional arguments does so in the form of an array of keyword argument structures. These functions have names ending with _k. See Keyword arguments.

2.1.7 Type punning

It’s tempting to use a type cast to change the type of an in/out or out parameter, like this:

```c
/* allocate a struct foo */
struct foo *fp;
res = mps_alloc((mps_addr_t *)&fp, pool, sizeof(struct foo));
```

This is known as type punning, and its behaviour is not defined in ANSI/ISO Standard C. See ISO/IEC 9899:1990 §6.3.2.3, which defines the conversion of a pointer from one type to another: the behaviour of this cast is not covered by any of the cases in the standard.

Instead, we recommend this approach:

```c
mps_addr_t p;
struct foo *fp;
res = mps_alloc(&p, pool, sizeof(struct foo));
if (res != MPS_RES_OK) /* handle error case */;
fp = p;
```

This has defined behaviour because conversion from void * to any other object pointer type is defined by ISO/IEC 9899:1990 §6.3.2.3.1.

2.1.8 Macros

1. For function-like macros, the MPS follows the same convention as the Standard C library. To quote ISO/IEC 9899:1990 §7.1.7:

   Any function declared in a header may additionally be implemented as a macro defined in the header, so a library function should not be declared explicitly if its header is included. Any macro definition of a function can be suppressed locally by enclosing the name of the function in parentheses, because the name is then not followed by the left parenthesis that indicates expansion of a macro function name. [...] Any invocation of a library function that is implemented as a macro shall expand to code that evaluates each of its arguments exactly once, fully protected by parentheses where necessary, so it is generally safe to use arbitrary expressions as arguments.

2. Some function-like macros evaluate an argument more than once, so it is not safe to have a side effect in an argument of such a method. These special cases are documented. For example, mps_reserve().

3. If you need the function rather than the macro, there are two approaches. You can undefined the macro:

   ```c
   #undef mps_reserve
   res = mps_reserve(...); /* calls function */
   ```

   Or you can put the name in parentheses:
res = (mps_reserve)(...); /* calls function */

4. Statement-like macros have names in uppercase, for example MPS_reserve_BLOCK(). These macros behave like statements rather than expressions, so that you cannot write:

(MPS_RESERVE_BLOCK(res, p, ap, size), 0)

5. Details of the macro expansion, although visible in the header file, are not part of the MPS interface, and might change between releases. Don’t rely on them, unless they are documented separately.

2.1.9 General types

mps_addr_t
The type of addresses managed by the MPS, and also the type of references.

It is a transparent alias for void *.

It is used in the MPS interface for any pointer that is under the control of the MPS. In accordance with standard C practice, null pointers of type mps_addr_t will never be used to represent a reference to a block.

mps_align_t
The type of an alignment.

It is a transparent alias for size_t.

An alignment must be a positive power of 2.

mps_bool_t
The type of a Boolean value.

It is a transparent alias for int.

When used as an input parameter to the MPS, a value of 0 means “false” and any other value means “true”. As an output parameter or function return from the MPS, 0 means “false”, and 1 means “true”.

mps_clock_t
The type of a processor time.

It is a transparent alias for mps_word_t.

This is the type returned by the plinth function mps_clock().

mps_fun_t
The type of a generic function pointer.

It is a transparent alias for void (*)(void).

mps_label_t
The type of a telemetry label.

It is an unsigned integral type.

mps_word_t
An unsigned integral type that is the same size as an object pointer, so that sizeof(mps_word_t) == sizeof(void *).

The exact identity of this type is platform-dependent. Typical identities are unsigned long and unsigned __int_64.

Topic
Platforms.
2.2 Keyword arguments

Some functions in the MPS interface take keyword arguments in order to pass values that might be optional, or are only required in some circumstances. For example, the function mps_arena_create_k() creates any class of arena, but client arenas require you to specify a base address. These arguments are passed in a keyword argument array, like this:

```c
mps_res_t res;
mps_arena_t arena;
mps_arg_s args[3];
args[0].key = MPS_KEY_ARENA_SIZE;
args[0].val.size = 6553600;
args[1].key = MPS_KEY_ARENA_CL_BASE;
args[1].val.addr = base_address;
args[2].key = MPS_KEY_ARGS_END;
res = mps_arena_create_k(&arena, mps_arena_class_cl(), args);
```

Each keyword argument in the array is a structure of type mps_arg_s.

For convenience and robustness, the MPS interface includes macros to help with forming keyword argument lists:

```c
MPS_ARGS_BEGIN(args) {
    MPS_ARGS_ADD(args, MPS_KEY_ARENA_SIZE, 6553600);
    MPS_ARGS_ADD(args, MPS_KEY_ARENA_CL_BASE, base_address);
    res = mps_arena_create_k(&arena, mps_arena_class_cl(), args);
} MPS_ARGS_END(args);
```

The argument array must not be NULL, and must end with MPS_KEY_ARGS_END. If you don’t want to pass any arguments, you can pass mps_args_none.

When a function that takes keyword arguments returns, the keyword argument array has been modified to remove any arguments that have been used. If all arguments have been used, the first element key is now MPS_KEY_ARGS_END.

mps_arg_s
The type of the structure used to represent a single keyword argument to a function.

```c
typedef struct mps_arg_s {
    mps_key_t key;
    union { /* many fields; see table below */ } val;
} mps_arg_s;
```

key identifies the key. It must be one of the values listed in the documentation for the type mps_key_t.

val is the corresponding value. This union contains many fields: one for each keyword argument type. The table given in the documentation for mps_key_t below indicates which structure field is used by each keyword.

Note
If you use the convenience macro MPS_ARGS_ADD() then you don’t need to know the name of the field.

mps_args_none
An array of mps_arg_s representing the empty list of keyword arguments. Equivalent to:

```c
mps_arg_s mps_args_none[] = {{MPS_KEY_ARGS_END}};
```

mps_key_t
The type of keyword argument keys. Must take one of the following values:
<table>
<thead>
<tr>
<th>Keyword</th>
<th>Type &amp; field in arg.val</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPS_KEY_ARGS_END</td>
<td>none</td>
<td>see above</td>
</tr>
<tr>
<td>MPS_KEY_ALIGN</td>
<td>mps_align_t align</td>
<td>mps_class_mv()</td>
</tr>
<tr>
<td>MPS_KEY_AWS_SUPPORT_AMBIGUOUS</td>
<td>mps_bool_t b</td>
<td>mps_arena_class_cl()</td>
</tr>
<tr>
<td>MPS_KEY_ARENA_CL_BASE</td>
<td>mps_addr_t addr</td>
<td>mps_arena_class_cl()</td>
</tr>
<tr>
<td>MPS_KEY_ARENA_SIZE</td>
<td>size_t size</td>
<td>mps_arena_class_vm()</td>
</tr>
<tr>
<td>MPS_KEY_ARENA_GRAIN_SIZE</td>
<td>size_t size</td>
<td>mps_arena_class_mvt()</td>
</tr>
<tr>
<td>MPS_KEY_AWL_FIND_DEPENDENT</td>
<td>void <em>(</em>)(void *) addr_method</td>
<td>mps_class_awl()</td>
</tr>
<tr>
<td>MPS_KEY_CHAIN</td>
<td>mps_chain_t chain</td>
<td>mps_class_amc()</td>
</tr>
<tr>
<td>MPS_KEY_EXTEND_BY</td>
<td>size_t size</td>
<td>mps_class_amc()</td>
</tr>
<tr>
<td>MPS_KEY_FMT_ALIGN</td>
<td>mps_align_t align</td>
<td>mps_fmt_create_k()</td>
</tr>
<tr>
<td>MPS_KEY_FMT_CLASS</td>
<td>mps_fmt_class_t fmt_class</td>
<td>mps_fmt_create_k()</td>
</tr>
<tr>
<td>MPS_KEY_FMT_FWD</td>
<td>mps_fmt_fwd_t fmt_fwd</td>
<td>mps_fmt_create_k()</td>
</tr>
<tr>
<td>MPS_KEY_FMT_HEADER_SIZE</td>
<td>size_t size</td>
<td>mps_fmt_create_k()</td>
</tr>
<tr>
<td>MPS_KEY_FMT_PAD</td>
<td>mps_fmt_pad_t fmt_pad</td>
<td>mps_fmt_create_k()</td>
</tr>
<tr>
<td>MPS_KEY_FMT_SCAN</td>
<td>mps_fmt_scan_t fmt_scan</td>
<td>mps_fmt_create_k()</td>
</tr>
<tr>
<td>MPS_KEY_FMT_SKIP</td>
<td>mps_fmt_skip_t fmt_skip</td>
<td>mps_fmt_create_k()</td>
</tr>
<tr>
<td>MPS_KEY_FORMAT</td>
<td>mps_fmt_t format</td>
<td>mps_class_amc()</td>
</tr>
<tr>
<td>MPS_KEY_GEN</td>
<td>unsigned u</td>
<td>mps_class_amc()</td>
</tr>
<tr>
<td>MPS_KEY_INTERIOR</td>
<td>mps_bool_t b</td>
<td>mps_class_amc()</td>
</tr>
<tr>
<td>MPS_KEY_MAX_SIZE</td>
<td>size_t size</td>
<td>mps_class_msv()</td>
</tr>
<tr>
<td>MPS_KEY_MEAN_SIZE</td>
<td>size_t size</td>
<td>mps_class_msv()</td>
</tr>
<tr>
<td>MPS_KEY_MFS_UNIT_SIZE</td>
<td>size_t size</td>
<td>mps_class_msv()</td>
</tr>
<tr>
<td>MPS_KEY_MIN_SIZE</td>
<td>size_t size</td>
<td>mps_class_msv()</td>
</tr>
<tr>
<td>MPS_KEY_MVF_FARENA_HIGH</td>
<td>mps_bool_t b</td>
<td>mps_class_mvf()</td>
</tr>
<tr>
<td>MPS_KEY_MVF_FIRST_FIT</td>
<td>mps_bool_t b</td>
<td>mps_class_mvf()</td>
</tr>
<tr>
<td>MPS_KEY_MVF_SLOT_HIGH</td>
<td>mps_bool_t b</td>
<td>mps_class_mvf()</td>
</tr>
<tr>
<td>MPS_KEY_MVT_FRAG_LIMIT</td>
<td>mps_word_t count</td>
<td>mps_class_mvt()</td>
</tr>
<tr>
<td>MPS_KEY_MVT_RESERVED_DEPTH</td>
<td>mps_word_t count</td>
<td>mps_class_mvt()</td>
</tr>
<tr>
<td>MPS_KEY_POOL_DEBUG_OPTIONS</td>
<td>mps_pool_debug_option_s *pool_debug_options</td>
<td>mps_class_ams()</td>
</tr>
<tr>
<td>MPS_KEY_RANK</td>
<td>mps_rank_t rank</td>
<td>mps_class_ams()</td>
</tr>
<tr>
<td>MPS_KEY_SPARE</td>
<td>double d</td>
<td>mps_class_mvf()</td>
</tr>
<tr>
<td>MPS_KEY_VMW3_TOP_DOWN</td>
<td>mps_bool_t b</td>
<td>mps_arena_class_cl()</td>
</tr>
</tbody>
</table>

**MPS_ARGS_BEGIN** (args)

Start construction of a list of keyword arguments. This macro must be used like this:

```c
MPS_ARGS_BEGIN(args) {
    MPS_ARGS_ADD(args, MPS_KEY_ARENA_SIZE, 6553600);
    MPS_ARGS_ADD(args, MPS_KEY_ARENA_CL_BASE, base_address);
    res = mps_arena_create_k(&arena, mps_arena_class_cl(), args);
} MPS_ARGS_END(args);
```

That is, you must call **MPS_ARGS_ADD()** (or **MPS_ARGS_ADD_FIELD()** ) zero or more times, and then pass the arguments to a function.

args is the name of the array that contains the keyword arguments. The array is stack-allocated, and exists between **MPS_ARGS_BEGIN** and **MPS_ARGS_END**.

It is safe to nest blocks created by **MPS_ARGS_BEGIN** and **MPS_ARGS_END**.

**MPS_ARGS_ADD** (mps_arg_s args[], mps_key_t key, value)

Add an argument to a list of keyword arguments. This macro must be used only between **MPS_ARGS_BEGIN** and **MPS_ARGS_END**.
args is the name of array that contains the keyword arguments. It must match the argument to the preceding call to `MPS_ARGS_BEGIN()`.

key is the keyword identifying this argument. It must be one of the key names starting with `MPS_KEY_` that are listed in the table in the documentation for `mps_key_t`.

value is the value for this argument.

**MPS_ARGS_ADD_FIELD** (mps_arg_s args[], mps_key_t key, field, value)

Add an argument to a list of keyword arguments. This macro must be used only between `MPS_ARGS_BEGIN()` and `MPS_ARGS_END()`.

args is the name of array that contains the keyword arguments. It must match the argument to the preceding call to `MPS_ARGS_BEGIN()`.

key is the keyword identifying this argument.

field is the name of the field in the val union in the structure mps_args_s.

value is the value for this argument.

**Note**

You should prefer to use `MPS_ARGS_ADD()` because then you don’t need to look up the name of the field.

**MPS_ARGS_END** (args)

Finish using a list of keyword arguments whose construction was started by `MPS_ARGS_BEGIN()`.

args is the name of array that contains the keyword arguments. It must match the argument to the preceding call to `MPS_ARGS_BEGIN()`.

## 2.3 Error handing

Operations in the Memory Pool System that might fail return a result code of type `mps_res_t`. Success is always indicated by the result code `MPS_RES_OK`, which is defined to be zero. Other result codes indicate failure, and are non-zero. The MPS never uses a “special value” of some other type to indicate failure (such as returning NULL for a pointer result, or 1 for a size result).

**Note**

The MPS does not throw or catch exceptions. (This is necessary for the MPS to be portable to systems that have only a freestanding implementation of the C language.)

The modular nature of the MPS means that it is not usually possible for a function description to list the possible error codes that it might return. A function in the public interface typically calls methods of an arena class and one or more pool classes, any of which might fail. The MPS is extensible with new arena and pool classes, which might fail in new and interesting ways, so the only future-proof behaviour is for a client program to assume that any MPS function that returns a result code can return any result code.

`mps_res_t`

The type of result codes. It is a transparent alias for `int`, provided for convenience and clarity.

A result code indicates the success or failure of an operation, along with the reason for failure. As with error numbers in Unix, the meaning of a result code depends on the call that returned it. Refer to the documentation of the function for the exact meaning of each result code.

The result codes are:

• `MPS_RES_OK`: operation succeeded.
• **MPS_RES_FAIL**: operation failed.
• **MPS_RES_IO**: an input/output error occurred.
• **MPS_RES_LIMIT**: an internal limitation was exceeded.
• **MPS_RES_MEMORY**: needed memory could not be obtained.
• **MPS_RES_RESOURCE**: a needed resource could not be obtained.
• **MPS_RES_UNIMPL**: operation is not implemented.
• **MPS_RES_COMMIT_LIMIT**: the arena’s *commit limit* would be exceeded.
• **MPS_RES_PARAM**: an invalid parameter was passed.

### 2.3.1 Result codes

**MPS_RES_COMMIT_LIMIT**

A *result code* indicating that an operation could not be completed as requested without exceeding the *commit limit*.

You need to deallocate something or allow the *garbage collector* to reclaim something to make more space, or increase the commit limit by calling `mps_arena_commit_limit_set()`.

**MPS_RES_FAIL**

A *result code* indicating that something went wrong that does not fall under the description of any other result code.

**MPS_RES_IO**

A *result code* indicating that an input/output error occurred in the telemetry system.

**MPS_RES_LIMIT**

A *result code* indicating that an operation could not be completed as requested because of an internal limitation of the MPS.

**MPS_RES_MEMORY**

A *result code* indicating that an operation could not be completed because there wasn’t enough memory available.

You need to deallocate something or allow the *garbage collector* to reclaim something to free enough memory, or extend the *arena* (if you’re using an arena for which that does not happen automatically).

---

**Note**

Failing to acquire enough memory because the *commit limit* would have been exceeded is indicated by returning `MPS_RES_COMMIT_LIMIT`, not `MPS_RES_MEMORY`.

Running out of *address space* (as might happen in *virtual memory* systems) is indicated by returning `MPS_RES_RESOURCE`, not `MPS_RES_MEMORY`.

**MPS_RES_OK**

A *result code* indicating that an operation succeeded.

If a function takes an *out parameter* or an *in/out parameter*, this parameter will only be updated if `MPS_RES_OK` is returned. If any other result code is returned, the parameter will be left untouched by the function.

`MPS_RES_OK` is zero.

**MPS_RES_PARAM**

A *result code* indicating that an operation could not be completed as requested because an invalid parameter was passed to the operation.
MPS_RES_RESOURCE

A result code indicating that an operation could not be completed as requested because the MPS could not obtain a needed resource. It can be returned when the MPS runs out of address space. If this happens, you need to reclaim memory within your process (as for the result code MPS_RES_MEMORY).

Two special cases have their own result codes: when the MPS runs out of committed memory, it returns MPS_RES_MEMORY, and when it cannot proceed without exceeding the commit limit, it returns MPS_RES_COMMIT_LIMIT.

MPS_RES_UNIMPL

A result code indicating that an operation, or some vital part of it, is not implemented.

This might be returned by functions that are no longer supported, or by operations that are included for future expansion, but not yet supported.

2.3.2 Assertions

Bugs in the client program may violate the invariants that the MPS relies on. Most functions in the MPS (in most varieties; see below) assert the correctness of their data structures, so these bugs will often be discovered by an assertion failure in the MPS. The section Common assertions and their causes below lists commonly encountered assertions and explains the kinds of client program bugs that can provoke these assertions.

It is very rare for an assertion to indicate a bug in the MPS rather than the client program, but it is not unknown, so if you have made every effort to track down the cause (see Debugging with the Memory Pool System) without luck, get in touch.

Assertion handling

When the MPS detects an assertion failure, it calls the plinth function mps_lib_assert_fail(). Unless you have replaced the plinth, this behaves as follows:

- In the cool variety, print the assertion message to standard error and terminate the program by calling abort().
- In the hot and rash varieties, print the assertion message to standard error and do not terminate the program.

You can change this behaviour by providing your own plinth, or using mps_lib_assert_fail_install().

In many applications, users don’t want their program terminated when the MPS detects an error, no matter how severe. A lot of MPS assertions indicate that the program is going to crash very soon, but there still may be a chance for a user to get some useful results or save their work. This is why the default assertion handler only terminates in the cool variety.

Common assertions and their causes

This section lists some commonly encountered assertions and suggests likely causes. If you encounter an assertion not listed here (or an assertion that is listed here but for which you discovered a different cause), please let us know so that we can improve this documentation.

arg.c: MPS_KEY...

A required keyword argument was omitted from a call to mps_ap_create_k(), mps_arena_create_k(), mps_fmt_create_k(), or mps_pool_create_k().

buffer.c: BufferIsReady(buffer)

The client program called mps_reserve() twice on the same allocation point without calling mps_commit(). See Allocation point protocol.
dbgpool.c: fencepost check on free

The client program wrote to a location after the end, or before the beginning of an allocated block. See Debugging pools.

dbgpool.c: free space corrupted on release

The client program used an object after it was reclaimed. See Debugging pools.

format.c: SigCheck Format: format

The client program called mps_pool_create_k() for a pool class like AMC (Automatic Mostly-Copying) that requires a object format, but passed something other than a mps_fmt_t for this argument.

global.c: RingIsSingle(&arena->chainRing)

The client program called mps_arena_destroy() without destroying all the generation chains belonging to the arena. It is necessary to call mps_chain_destroy() first.

global.c: RingIsSingle(&arena->formatRing)

The client program called mps_arena_destroy() without destroying all the object formats belonging to the arena. It is necessary to call mps_fmt_destroy() first.

global.c: RingIsSingle(&arena->rootRing)

The client program called mps_arena_destroy() without destroying all the roots belonging to the arena. It is necessary to call mps_root_destroy() first.

global.c: RingIsSingle(&arena->threadRing)

The client program called mps_arena_destroy() without deregistering all the threads belonging to the arena. It is necessary to call mps_thread_dereg() first.

global.c: RingLength(&arenaGlobals->poolRing) == 5

The client program called mps_arena_destroy() without destroying all the pools belonging to the arena. It is necessary to call mps_pool_destroy() first.

lockix.c: res == 0

lockw3.c: lock->claims == 0

The client program has made a re-entrant call into the MPS. Look at the backtrace to see what it was. Common culprits are signal handlers, assertion handlers, format methods, and stepper functions.

locus.c: chain->activeTraces == TraceSetEMPTY

The client program called mps_chain_destroy(), but there was a garbage collection in progress on that chain. Park the arena before destroying the chain, by calling mps_arena_park().

mpsi.c: SizeIsAligned(size, BufferPool(buf)->alignment)

The client program reserved a block by calling mps_reserve() but neglected to round the size up to the alignment required by the pool’s object format.

poolams.c: AMS_ALLOCED(seg, i)

The client program tried to fix a reference to a block in an AMS (Automatic Mark and Sweep) pool that died. This may mean that there was a previous collection in which a reference that should have kept the block alive failed to be scanned. Perhaps a formatted object was updated in some way that has a race condition?

poolsnc.c: foundSeg

The client program passed an incorrect frame argument to mps_ap_frame_pop(). This argument must be the result from a previous call to mps_ap_frame_push() on the same allocation point.
The client program destroyed pool without first destroying all the allocation points created on that pool. The allocation points must be destroyed first.

The client program destroyed a pool containing objects registered for finalization, and then continued to run the garbage collector. See Caution under Finalization, which says, “You must destroy these pools by following the ‘safe tear-down’ procedure described under mps_pool_destroy().”

The client program’s scan method failed to update a reference to an object that moved. See Scanning protocol, which says, “If MPS_FIX2() returns MPS_RES_OK, it may have updated the reference. Make sure that the updated reference is stored back to the region being scanned.”

2.3.3 Varieties

The MPS has three varieties which have different levels of internal checking and telemetry. The variety can be selected at compile time, by defining one of the following preprocessor constants. If none is specified then CONFIG_VAR_HOT is the default.

**CONFIG_VAR_COOL**

The cool variety is intended for development and testing.

All functions check the consistency of their data structures and may assert, including functions on the critical path. Furthermore, in the default ANSI Library the default assertion handler will terminate the program. See mps_lib_assert_fail_install().

All events are sent to the telemetry stream, including events on the critical path.

**CONFIG_VAR_HOT**

The hot variety is intended for production and deployment.

Some functions check the consistency of their data structures and may assert, namely those not on the critical path. However, in the default ANSI Library, the default assertion handler will not terminate the program. See mps_lib_assert_fail_install().

Some events are sent to the telemetry stream, namely those not on the critical path.

**CONFIG_VAR_RASH**

The rash variety is intended for mature integrations, or for developers who like living dangerously.

No functions check the consistency of their data structures and consequently there are no assertions.

No events are sent to the telemetry stream.

2.4 Arenas

An arena is an object that encapsulates the state of the Memory Pool System, and tells it where to get the memory it manages. You typically start a session with the MPS by creating an arena with mps_arena_create_k() and end the session by destroying it with mps_arena_destroy(). The only function you might need to call before making an arena is mps_telemetry_control().

Before destroying an arena, you must first destroy all objects and data in it, as usual for abstract data types in the MPS. If you can’t destroy the arena properly (for example, because your program has crashed and you are at the debugger prompt), you can still call mps_telemetry_flush() explicitly.
Other types of objects in the MPS are created “in the arena”. They are part of the world within the arena, and may interact and affect each other.

**Note**

The MPS allows creation of multiple arenas, but you would only do this in unusual circumstances. It might be useful to have two active arenas and to try different things out in them, or you might be in the process of integrating two pieces of software that each independently uses the MPS.

 Arenas do not normally interact, but they compete with each other for resources, and references from one arena to another are not traced, though you can declare roots pointing from one arena to another. It is not efficient to have multiple arenas containing automatically managed pools: if you find yourself in this situation it’s best to find a way to move all the automatically managed pools to one arena.

The open source MPS comes with two classes of arena, *Client arenas* and *Virtual memory arenas*. These differ in the way that they acquire the memory to be managed.

**Note**

The MPS is designed to be extensible with new arena classes. If you need features that are not provided by any of the open source arena classes, contact us.

### mps_arena_t

The type of *arenas*.

An arena is responsible for requesting memory (3) from the operating system, making it available to pools, and for garbage collection.

### mps_arena_class_t

The type of *arena classes*.

```c
mps_res_t mps_arena_create_k (mps_arena_t *arena_o, mps_arena_class_t arena_class,
                          mps_arg_s args[])
```

Create an arena.

*arena_o* points to a location that will hold a pointer to the new arena.

*arena_class* is the arena class.

*args* are keyword arguments specific to the arena class. See the documentation for the arena class.

Returns MPS_RES_OK if the arena is created successfully, or another result code otherwise.

The arena persists until it is destroyed by calling mps_arena_destroy().

### void mps_arena_destroy (mps_arena_t arena)

Destroy an arena.

*arena* is the arena to destroy.

This function checks the consistency of the arena, flushes the telemetry stream and destroys the arena’s internal control structures. Additionally, *virtual memory arenas* return their reserved address space to the operating system if possible.

It is an error to destroy an arena without first destroying all generation chains, object formats, pools and roots created in the arena, and deregistering all threads registered with the arena.
2.4.1 Client arenas

```c
#include "mpsacl.h"

mps_arena_class_t mps_arena_class_cl(void)
Return the arena class for a client arena.

A client arena gets its managed memory from the client program. This memory chunk is passed when the arena is created.

When creating a client arena, `mps_arena_create_k()` requires two keyword arguments:

- `MPS_KEY_ARENA_CL_BASE` (type `mps_addr_t`) is the address of the chunk of memory that will be managed by the arena.
- `MPS_KEY_ARENA_SIZE` (type `size_t`) is its size.

It also accepts one optional keyword argument:

- `MPS_KEY_ARENA_GRAIN_SIZE` (type `size_t`, default 8192) is the granularity with which the arena will manage memory internally. It must be a power of 2. Larger granularity reduces overheads, but increases fragmentation and retention.

For example:

```c
MPS_ARGS_BEGIN(args) {
  MPS_ARGS_ADD(args, MPS_KEY_ARENA_CL_BASE, base);
  MPS_ARGS_ADD(args, MPS_KEY_ARENA_SIZE, size);
  res = mps_arena_create_k(&arena, mps_arena_class_cl(), args);
} MPS_ARGS_END(args);
```

If the chunk is too small to hold the internal arena structures, `mps_arena_create_k()` returns `MPS_RES_MEMORY`. In this case, you need to use a (much) larger chunk.

**Note**
You don’t have to provide all the memory up front: you can call `mps_arena_extend()` later on.

Client arenas have no mechanism for returning unused memory.

```c
mps_res_t mps_arena_extend(mps_arena_t arena, mps_addr_t base, size_t size)
Extend a client arena with another block of memory.
base is the address of the block of memory that will be managed by the arena.
size is its size.

Return MPS_RES_OK if successful, or another result code if it fails.
```

2.4.2 Virtual memory arenas

```c
#include "mpsavm.h"

mps_arena_class_t mps_arena_class_vm(void)
Return the arena class for a virtual memory arena.

A virtual memory arena uses the operating system’s virtual memory interface to allocate memory. The chief consequence of this is that the arena can manage many more virtual addresses than it needs to commit memory to. This gives it flexibility as to where to place blocks, which reduces fragmentation and helps make garbage collection more efficient.
```
When creating a virtual memory arena, `mps_arena_create_k()` accepts two optional *keyword arguments* on all platforms:

- **MPS_KEY_ARENA_SIZE** (type `size_t`, default 256 *megabytes*) is the initial amount of virtual address space, in *bytes* (I), that the arena will reserve (this space is initially reserved so that the arena can subsequently use it without interference from other parts of the program, but most of it is not committed, so it doesn’t require any RAM or backing store). The arena may allocate more virtual address space beyond this initial reservation as and when it deems it necessary. The MPS is most efficient if you reserve an address space that is several times larger than your peak memory usage.

If you specify a value for `MPS_KEY_ARENA_SIZE` that’s too small for the virtual memory arena, then the MPS rounds it up to the minimum and continues. The minimum size for the virtual memory arena is \( MPS_{\text{WORD\_WIDTH}} \times MPS_{\text{KEY\_ARENA\_GRAIN\_SIZE}} \) bytes. For example, on a 64-bit platform with a 4 *kilobyte* page size, this is 256 *kilobytes*.

**Note**
The MPS asks for more address space if it runs out, but the more times it has to extend its address space, the less efficient garbage collection will become.

- **MPS_KEY_ARENA_GRAIN_SIZE** (type `size_t`) is the granularity with which the arena will manage memory internally. It must be a power of 2. If not provided, the operating system’s page size is used. Larger granularity reduces overheads, but increases *fragmentation* and *retention*.

If you specify a value of `MPS_KEY_ARENA_GRAIN_SIZE` that’s smaller than the operating system page size, the MPS rounds it up to the page size and continues.

A third optional *keyword argument* may be passed, but it only has any effect on the Windows operating system:

- **MPS_KEY_VMW3_TOP_DOWN** (type `mps_bool_t`). If true, the arena will allocate address space starting at the highest possible address and working downwards through memory.

**Note**
This causes the arena to pass the `MEM_TOP_DOWN` flag to `VirtualAlloc`.

If the MPS fails to reserve adequate address space to place the arena in, `mps_arena_create_k()` returns `MPS_RES_Resource`. Possibly this means that other parts of the program are reserving too much virtual memory.

If the MPS fails to allocate memory for the internal arena structures, `mps_arena_create_k()` returns `MPS_RES_MEMORY`. Either `MPS_KEY_ARENA_SIZE` was far too small or the operating system refused to provide enough memory.

For example:

```c
MPS_ARGS_BEGIN(args) {
    MPS_ARGS_ADD(args, MPS_KEY_ARENA_SIZE, size);
    res = mps_arena_create_k(&arena, mps_arena_class_vm(), args);
} MPS_ARGS_END(args);
```

### 2.4.3 Arena properties

`mps_word_t mps_collections(mps_arena_t arena)`

Return the number of *flips* that have taken place in an *arena* since it was created.

*arena* is the arena.
size_t mps_arena_commit_limit (mps_arena_t arena)

Return the current commit limit for an arena.

arena is the arena to return the commit limit for.

Returns the commit limit in bytes. The commit limit controls how much memory the MPS can obtain from the operating system, and can be changed by calling mps_arena_commit_limit_set().

mps_res_t mps_arena_commit_limit_set (mps_arena_t arena, size_t limit)

Change the commit limit for an arena.

arena is the arena to change the commit limit for.

limit is the new commit limit in bytes.

Returns MPS_RES_OK if successful, or another result code if not.

If successful, the commit limit for arena is set to limit. The commit limit controls how much memory the MPS will obtain from the operating system. The commit limit cannot be set to a value that is lower than the number of bytes that the MPS is using. If an attempt is made to set the commit limit to a value greater than or equal to that returned by mps_arena_committed() then it will succeed. If an attempt is made to set the commit limit to a value less than that returned by mps_arena_committed() then it will succeed only if the amount committed by the MPS can be reduced by reducing the amount of spare committed memory; in such a case the spare committed memory will be reduced appropriately and the attempt will succeed.

Note
mps_arena_commit_limit_set() puts a limit on all memory committed by the MPS. The spare committed memory can be limited separately with mps_arena_spare_commit_limit_set(). Note that “spare committed” memory is subject to both limits; there cannot be more spare committed memory than the spare commit limit, and there can’t be so much spare committed memory that there is more committed memory than the commit limit.

size_t mps_arena_committed (mps_arena_t arena)

Return the total committed memory for an arena.

arena is the arena.

Returns the total amount of memory that has been committed for use by the MPS, in bytes.

For a virtual memory arena, this is the amount of memory mapped to RAM by the operating system’s virtual memory interface.

For a client arena, this is the amount of memory marked as in use in the arena’s page tables. This is not particularly meaningful by itself, but it corresponds to the amount of mapped memory that the MPS would use if switched to a virtual memory arena.

The committed memory is generally larger than the sum of the sizes of the allocated blocks. The reasons for this are:

• some memory is used internally by the MPS to manage its own data structures and to record information about allocated blocks (such as free lists, page tables, colour tables, statistics, and so on);
• operating systems (and hardware) typically restrict programs to requesting and releasing memory with a certain granularity (for example, pages), so extra memory is committed when this rounding is necessary;
• there might also be spare committed memory: see mps_arena_spare_committed().

The amount of committed memory is a good measure of how much virtual memory resource (“swap space”) the MPS is using from the operating system.

2.4. Arenas
The function `mps_arena_committed()` may be called whatever state the arena is in (unclamped, clamped, or parked). If it is called when the arena is in the unclamped state then the value may change after this function returns. A possible use might be to call it just after `mps_arena_collect()` to estimate the size of the heap.

If you want to know how much memory the MPS is using then you're probably interested in the value `mps_arena_committed()` `mps_arena_spare_committed()`.

The amount of committed memory can be limited with the function `mps_arena_commit_limit()`.

```
size_t mps_arena_reserved (mps_arena_t arena)
Return the total address space reserved by an arena, in bytes (1).
arena is the arena.
```

For a virtual memory arena, this is the total address space reserved via the operating system’s virtual memory interface.

For a client arena, this is the sum of the usable portions of the chunks of memory passed to the arena by the client program via `mps_arena_create_k()` and `mps_arena_extend()`.

### Note

For a client arena, the reserved address space may be lower than the sum of the MPS_KEY_ARENA_SIZE keyword argument passed to `mps_arena_create_k()` and the `size` arguments passed to `mps_arena_extend()`, because the arena may be unable to use the whole of each chunk for reasons of alignment.

```
size_t mps_arena_spare_commit_limit (mps_arena_t arena)
Return the current spare commit limit for an arena.
arena is the arena to return the spare commit limit for.
```

Returns the spare commit limit in bytes (1). The spare commit limit can be changed by calling `mps_arena_spare_commit_limit_set()`.

```
void mps_arena_spare_commit_limit_set (mps_arena_t arena, size_t limit)
Change the spare commit limit for an arena.
arena is the arena to change the spare commit limit for.
limit is the new spare commit limit in bytes (1).
```

The spare commit limit is the maximum amount of spare committed memory the MPS is allowed to have. Setting it to a value lower than the current amount of spare committed memory causes spare committed memory to be uncommitted so as to bring the value under the limit. In particular, setting it to 0 will mean that the MPS will have no spare committed memory.

Non-virtual-memory arena classes (for example, a client arena) do not have spare committed memory. For these arenas, this function sets a value but has no other effect.

Initially the spare commit limit is a configuration-dependent value. The value of the limit can be retrieved by the function `mps_arena_spare_commit_limit()`.

```
size_t mps_arena_spare_committed (mps_arena_t arena)
Return the total spare committed memory for an arena.
arena is the arena.
```

Returns the number of bytes of spare committed memory.

Spare committed memory is memory which the arena is managing as free memory (not in use by any pool and not otherwise in use for internal reasons) but which remains committed (mapped to RAM by the operating sys-
It is used by the arena to (attempt to) avoid calling the operating system to repeatedly map and unmap areas of virtual memory as the amount of memory in use goes up and down. Spare committed memory is counted as committed memory by `mps_arenacommitted()` and is restricted by `mps_areanecommit_limit()`. The amount of “spare committed” memory can be limited by calling `mps_arena_spare_commit_limit_set()`, and the value of that limit can be retrieved with `mps_arena_spare_commit_limit()`. This is analogous to the functions for limiting the amount of committed memory.

**Note**

*Client arenas* do not use spare committed memory, and so this function always returns 0.

### 2.4.4 Arena states

An arena is always in one of three states.

1. In the **unclamped state**, garbage collection may take place, objects may move in memory, references may be updated, *location dependencies* may become stale, virtual memory may be requested from or returned to the operating system, and other kinds of background activity may occur. This is the normal state.

2. In the **clamped state**, objects do not move in memory, references do not change, the staleness of *location dependencies* does not change, and memory occupied by *unreachable* objects is not recycled. However, a *garbage collection* may be in progress and incremental collection may still occur, but it will not be visible to the *client program* and no new collections will begin.

3. The **parked state** is the same as the clamped state, with the additional constraint that no garbage collections are in progress.

Here’s a summary:

<table>
<thead>
<tr>
<th>State</th>
<th>unclamped</th>
<th>clamped</th>
<th>parked</th>
</tr>
</thead>
<tbody>
<tr>
<td>Collections may be running?</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>New collections may start?</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Objects may move?</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Location dependencies may become stale?</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Memory may be returned to the OS?</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Functions that leave the arena in this state</td>
<td><code>mps_arena_create_k()</code>, <code>mps_arena_release()</code>, <code>mps_arena_start_collect()</code>, <code>mps_arena_step()</code></td>
<td><code>mps_arena_clamp()</code>, <code>mps_arena_park()</code>, <code>mps_arena_collect()</code></td>
<td></td>
</tr>
</tbody>
</table>

The clamped and parked states are important when introspecting and debugging. If you are examining the contents of the heap, you don’t want data moving under your feet. So for example, if your program is stopped in GDB you might type:

```
(gdb) print mps_arena_clamp(arena)
```
before inspecting memory, and:

```
(gdb) print mps_arena_release(arena)
```

afterwards.

The results of introspection functions like `mps_arena_has_addr()` only remain valid while the arena remains in the parked state, and functions like `mps_arena_roots_walk()` can only be called in this state.

```c
void mps_arena_clamp (mps_arena_t arena)

Put an arena into the clamped state.

arena is the arena to clamp.

In the clamped state, no object motion will occur and the staleness of location dependencies will not change. All references to objects loaded while the arena is clamped will keep the same binary representation until after it is released by calling `mps_arena_release()`.

In a clamped arena, incremental collection may still occur, but it will not be visible to the mutator and no new collections will begin. Space used by unreachable objects will not be recycled until the arena is unclamped.

```c
void mps_arena_park (mps_arena_t arena)

Put an arena into the parked state.

arena is the arena to park.

While an arena is parked, no object motion will occur and the staleness of location dependencies will not change. All references to objects loaded while the arena is parked will keep the same binary representation until after it is released.

Any current collection is run to completion before the arena is parked, and no new collections will start. When an arena is in the parked state, it is necessarily not in the middle of a collection.

```c
void mps_arena_release (mps_arena_t arena)

Puts an arena into the unclamped state.

arena is the arena to unclamp.

While an arena is unclamped, garbage collection, object motion, and other background activity can take place.

### 2.4.5 Running garbage collections

The Memory Pool System’s garbage collector runs asynchronously and incrementally. This means that it is not normally necessary to tell it when to start garbage collections, or to wait until it has finished collecting. (But if your program has idle time that could be productively spent by the MPS, see Using idle time for collection below.)

However, during development and testing it is useful to be able to request that MPS run a full collection cycle. For example, you might run frequent collections in an attempt to detect bugs in your allocation and scanning code.

```c
void mps_arena_collect (mps_arena_t arena)

Collect an arena and put it into the parked state.

arena is the arena to collect.

The collector attempts to recycle as many unreachable objects as possible and reduce the size of the arena as much as possible (though in some cases it may increase because it becomes more fragmented). Note that the collector may not be able to recycle some objects (such as those near the destination of ambiguous references) even though they are not reachable.

If you do not want the arena to remain in the parked state, you must explicitly call `mps_arena_release()` afterwards.
Note
It is not normally necessary to call this function: in the unclamped state, collections start automatically. However, it may be useful during development and debugging: the more frequently the collector runs, the sooner and more reliably errors are discovered. See General debugging advice.

```c
mps_res_t mps_arena_start_collect(mps_arena_t arena)
```
Request an arena to start a full collection cycle.

arena is the arena.

Returns MPS_RES_OK if a collection is started, or another result code if not.

This function puts arena into the unclamped state and requests that it start a full collection cycle. The call to mps_arena_start_collect() returns quickly, leaving the collection to proceed incrementally (as for a collection that is scheduled automatically).

Note
Contrast with mps_arena_collect(), which does not return until the collection has completed.

### 2.4.6 Using idle time for collection

Some types of program have “idle time” in which they are waiting for an external event such as user input or network activity. The MPS provides a function, mps_arena_step(), for making use of idle time to make memory management progress.

Here’s an example illustrating the use of this function in a program’s event loop.

```c
for (;;) {
    /* event loop */
    for (;;) {
        if (client_is_waiting()) {
            perform_client_action();
        } else if (!mps_arena_step(arena, 0.010, 0.0)) {
            /* no incremental MPS work remaining */
            break;
        }
    }

    if (!block_on_client_with_timeout(2.0)) {
        /* Perhaps the user has gone for a cup of coffee? Allow the
         * MPS to start a big piece of work, but don't actually pause
         * for more than 10 ms. */
        mps_arena_step(arena, 0.010, 100.0);
    }
}
```

When the program is idle (there are no client actions to perform), it requests that the MPS spend up to 10 milliseconds on incremental work, by calling mps_arena_step(arena, 0.010, 0.0). When this returns false to indicate that there is no more work to do, the program blocks on the client for two seconds: if this times out, it predicts that the user will remain idle for at least a further second, so it calls mps_arena_step(arena, 0.010, 100.0) to tell that it’s a good time to start a collection taking up to 10 ms × 100 = 1 second, but not to pause for more than 10 ms.

The program remains responsive: the MPS doesn’t take control for more than a few milliseconds at a time (at most 10). But at the same time, major collection work can get done at times when the program would otherwise be idle. Of course the numbers here are only for illustration; they should be chosen based on the requirements of the application.
**mps_bool_t mps_arena_step (mps_arena_t arena, double interval, double multiplier)**

Request an arena to do some work during a period where the client program is idle.

arena is the arena.

interval is the time, in seconds, the MPS is permitted to take. It must not be negative, but may be 0.0.

multiplier is the number of further similar calls that the client program expects to make during this idle period.

Returns true if there was work for the MPS to do in arena (regardless of whether or not it did any) or false if there was nothing to do.

**mps_arena_step()** allows the client program to make use of idle time to do some garbage collection, for example when it is waiting for interactive input. The MPS makes every effort to return from this function within interval seconds, but cannot guarantee to do so, as it may need to call your own scanning code. It uses multiplier to decide whether to commence long-duration operations that consume CPU (such as a full collection): it will only start such an operation if it is expected to be completed within multiplier * interval seconds.

If the arena was in the parked state or the clamped state before mps_arena_step() was called, it is in the clamped state afterwards. It it was in the unclamped state, it remains there.

### 2.4.7 Arena introspection

**Note**

Introspection functions covered in other chapters are:

- **mps_addr_fmt()**: determine the object format to which an address belongs;
- **mps_arena_formatted_objects_walk()**: visit all formatted objects in an arena;
- **mps_arena_roots_walk()**: visit all references in roots registered with an arena; and
- **mps_addr_pool()**: determine the pool to which an address belongs.

**mps_bool_t mps_arena_has_addr (mps_arena_t arena, mps_addr_t addr)**

Test whether an address is managed by an arena.

arena is an arena.

addr is an address.

Returns true if addr is managed by arena; false otherwise.

An arena manages a portion of address space. No two arenas overlap, so for any particular address this function will return true for at most one arena.

In general, not all addresses are managed by any arena. This is what allows the MPS to cooperate with other memory managers, shared object loaders, memory mapped file input/output, and so on: it does not steal the whole address space.

**Note**

The result from this function is valid only at the instant at which the function returned. In some circumstances the result may immediately become invalidated (for example, a garbage collection may occur, the address in question may become free, the arena may choose to unmap the address and return storage to the operating system). For reliable results call this function and interpret the result while the arena is in the parked state.
2.5 Pools

Within an arena a client program creates one or more pools. A pool is responsible for requesting memory from the arena and making it available for allocation.

mps_pool_t
The type of pools.

A pool is responsible for requesting memory from the arena and making it available to the client program via mps_alloc() or via an allocation point.

mps_res_t mps_pool_create_k (mps_pool_t *pool_o, mps_arena_t arena, mps_pool_class_t pool_class, mps_arg_s args[])

Create a pool in an arena.

pool_o points to a location that will hold a pointer to the new pool.
arena is the arena in which to create the pool.
pool_class is the pool class of the new pool.
args are keyword arguments specific to the pool class. See the documentation for the pool class.

Returns MPS_RES_OK if the pool is created successfully, or another result code otherwise.

The pool persists until it is destroyed by calling mps_pool_destroy().

void mps_pool_destroy (mps_pool_t pool)

Destroy a pool.

pool is the pool to destroy.

This function checks the consistency of the pool, destroys the pool’s internal control structures and causes the pool’s memory to be returned to the arena for reuse by other pools, or to be returned to the operating system. Blocks allocated from the pool may no longer be used.

It is an error to destroy a pool without first destroying all allocation points and segregated allocation caches created in the pool.

Warning: It is not safe to carry on running the garbage collector after destroying an automatically managed pool that contains any objects that are reachable from your roots, or any objects that have been registered for finalization but not yet finalized.

Our recommended approach is to destroy automatically managed pools just before destroying the arena, and then only while the arena is in the parked state. Thus a safe tear-down sequence looks like this:

mps_arena_park(arena);
/* destroy threads and roots belonging to the arena */
/* destroy allocation points and caches belonging to the pool */
mps_pool_destroy(pool);
/* destroy chains and formats belonging to the arena */
mps_arena_destroy(arena);

2.5.1 Pool classes

Pools belong to pool classes that specify policies for how their memory is managed. Some pools are manually managed (you must call mps_free() to return a block of memory to the pool) and others are automatically managed (the garbage collector reclaims unreachable blocks).

See the Pool reference for a list of pool classes.
mps_pool_class_t
The type of pool classes.

2.5.2 Pool introspection

size_t mps_pool_total_size (mps_pool_t pool)
Return the total memory allocated from the arena and managed by the pool.
pool is the pool.
The result includes memory in use by the client program, memory that’s available for use by the client program, and memory that’s lost to fragmentation. It does not include memory used by the pool’s internal control structures.

size_t mps_pool_free_size (mps_pool_t pool)
Return the free memory: memory managed by the pool but not in use by the client program.
pool is the pool.
The result includes memory that’s available for use by the client program, and memory that’s lost to fragmentation. It does not include memory used by the pool’s internal control structures.

mps_bool_t mps_addr_pool (mps_pool_t *pool_o, mps_arena_t arena, mps_addr_t addr)
Determine the pool to which an address belongs.
pool_o points to a location that will hold the address of the pool, if one is found.
arena is the arena whose pools will be considered.
addr is the address.
If addr is the address of a location inside a block allocated from a pool in arena, then update the location pointed to by pool_o with the address of the pool, and return true.
If addr points to a location that is not managed by arena, return false.
If neither of the above conditions is satisfied, mps_addr_pool() may return either true or false.

Note
This function might return a false positive by returning true if you ask about an address that happens to be inside memory managed by a pool, but which is not inside a block allocated by that pool. It never returns a false negative.
The result from this function is valid only at the instant at which the function returned. In some circumstances the result may immediately become invalidated. For reliable results call this function and interpret the result while the arena is in the parked state.

2.6 Allocation

2.6.1 Manual allocation

Note
Not all pool classes support this interface: automatically managed pools typically support none of it, and even manually managed pools may not support the whole interface. Consult the pool class documentation for details. For example, the MVT (Manual Variable Temporal) pool class supports deallocation via mps_free() but allocation must use allocation points, as described below.
mps_res_t mps_alloc (mps_addr_t *p_o, mps_pool_t pool, size_t size)
  Allocate a "block" of memory in a "pool".
  p_o points to a location that will hold the address of the allocated block.
  pool is the pool to allocate in.
  size is the size of the block to allocate. If it is unaligned, it will be rounded up to the pool's "alignment" (unless the pool documentation says otherwise).

void mps_free (mps_pool_t pool, mps_addr_t addr, size_t size)
  Free a "block" of memory to a "pool".
  pool is the pool the block belongs to.
  addr is the address of the block to be freed.
  size is the size of the block to be freed. If it is unaligned, it will be rounded up to the pool's "alignment" (unless the pool documentation says otherwise).

The freed block of memory becomes available for allocation by the pool, or the pool might decide to make it available to other pools, or it may be returned to the operating system.

**Note**

mps_free() takes a size parameter because it is most efficient to do so. In most programs, the type of an object is known at the point in the code that frees it, hence the size is trivially available. In such programs, storing the size on the MPS side would cost time and memory, and make it hard to get good virtual memory behaviour because of the need to touch the object in order to free it. As it is, the deallocation code doesn’t have to touch the dead object at all.

### 2.6.2 Allocation points

**Allocation points** provide fast, *inline*, nearly *lock-free* allocation. They allow code to allocate without calling an allocation function: this is vital for performance in languages or programs that allocate many small objects. They must be used according to the Allocation point protocol.

mps_ap_t
  The type of allocation points. It is a transparent alias for a pointer to mps_ap_s.

mps_res_t mps_ap_create_k (mps_ap_t *ap_o, mps_pool_t pool, mps_arg_s args[])
  Create an allocation point in a pool.
  ap_o points to a location that will hold the address of the allocation point, if successful.
  pool is the pool.
  args are keyword arguments specific to the pool class to which pool belong. See the documentation for that pool class. (Most pool classes don’t take any keyword arguments; in those cases you can pass mps_args_none.)

Returns MPS_RES_OK if successful, or another result code if not.

**Warning:** An allocation point must not be used by more than one thread; each thread must create its own allocation point or points.

void mps_ap_destroy (mps_ap_t ap)
  Destroy an allocation point.
Destroying an allocation point has no effect on blocks that were allocated from it, so long as they were successfully committed by `mps_commit()`.

### 2.6.3 Allocation point protocol

This protocol is designed to work with incremental garbage collection and multiple threads, where between any two instructions in the client program, the MPS may run part of a garbage collection, move blocks in memory, rewrite pointers, and reclaim space. In order to reliably handle this, the allocation point protocol consists of (at least) two steps, a `reserve` followed by a `commit`.

#### Note

The description of the protocol assumes that you have declared your threads' control stacks and registers to be ambiguous roots, by passing `mps_stack_scan_ambig()` to `mps_root_create_reg()`. This is the simplest way to write a client, but other scenarios are possible. Please contact us if your use case is not covered here (for example, if you need an exact collector).

When the client program is initializing a newly allocated object, you can think of it as being “in a race” with the MPS. Until the object is initialized, the MPS cannot manage it in the usual way: in particular, it cannot ensure that the new object remains correct if other objects move during its initialization. So if other objects do move, the MPS tells the client program that it has “lost the race”: the partially-initialized object may be invalid, and the client must initialize it again from scratch.

The allocation point protocol is as follows:

1. Call `mps_reserve()` to reserve a block of memory on an allocation point. The size of the block must be a multiple of the alignment of the pool in which the allocation point was created.

   If `mps_reserve()` returns `MPS_RES_OK`, go to step 2.

   Otherwise, the block cannot be reserved (this might happen if the MPS is out of memory).

2. Initialize the block. During this step the block must not be referenced by an exact reference, and references stored in it must not be followed.

   The block need not be initialized completely, but if the pool has an object format, then by the end of this step, the block must be capable of being passed to the format’s scan method and skip method.

3. Call `mps_commit()` to attempt to commit the object to the care of the MPS.

   If `mps_commit()` returns true, this means that the object is valid, and is now under the management of the MPS. The client program may rely on references stored in the object, and may store references to the new object in its other objects.

   If `mps_commit()` returns false, this means that the block is invalid. It is usual in this case to go back to step 1 and re-reserve and re-initialize it, but other courses of action are permitted.

#### Note

In this case, the reason the block is invalid because a flip took place after the call to `mps_reserve()` and before the call to `mps_commit()`. This means that references in the block may point to the old location of blocks that moved.

The usual implementation of the allocation point protocol in C is thus:
```c
mp_obj_t obj;
size_t aligned_size = ALIGN(size); /* see note 1 */
do {
    mps_res_t res = mps_reserve(p, ap, aligned_size);
    if (res != MPS_RES_OK)
        /* handle the error */;
    /* p is now an ambiguous reference to the reserved block */
    obj = p;
    /* initialize obj */
} while (!mps_commit(ap, p, aligned_size)); /* see note 2 */
/* obj is now valid and managed by the MPS */
```

Notes

1. Here ALIGN() represents a function or macro that rounds size up to the necessary alignment, which should be at least as big as the alignment of the pool. (The reason that the MPS does not do this rounding up for you is to provide more opportunities for optimization: in many cases the required alignment will be a constant that’s known at compilation time.)

2. mps_commit() returns false only if a garbage collection flip occurs after mps_reserve(). This is a very rare event, especially if the object initialization is short.

```c
mp_res_t mps_reserve(mps_addr_t *p, mps_ap_t ap, size_t size)
    Reserve a block of memory on an allocation point.
    p points to a location that will hold the address of the reserved block.
    ap is the allocation point.
    size is the size of the block to allocate. It must be a multiple of the alignment of the pool (or of the pool’s object format if it has one).
    Returns MPS_RES_OK if the block was reserved successfully, or another result code if not.
    The reserved block may be initialized but must not otherwise be used
    Until it has been committed (2) via a successful call to mps_commit(), the reserved block may be:
        • initialized;
        • referenced by an ambiguous reference;
    but:
        • it must not be referenced by an exact reference;
        • references stored in it must not be followed;
        • it is not scanned, moved, or protected (even if it belongs to a pool with these features).
```

Note

mps_reserve() must only be called according to the Allocation point protocol.
mps_reserve() is implemented as a macro for speed. It may evaluate its arguments multiple times.

There is an alternative, MPS_RESERVE_BLOCK(), which may generate faster code on some compilers.

```c
MPS_RESERVE_BLOCK (mp_res_t res_v, mps_addr_t *p_v, mps_ap_t ap, size_t size)
    An alternative to mps_reserve(). On compilers that do not perform common-subexpression elimination, it
may generate faster code than \texttt{mps\_reserve()} (but may not). It may only be used in statement context (not as an expression).

The second argument is an lvalue \texttt{p\_v}, which is assigned the address of the reserved block. It takes an additional first argument, the lvalue \texttt{res\_v}, which is assigned the \texttt{result code}.

\begin{verbatim}
mps\_bool\_t mps\_commit (mps\_ap\_t ap, mps\_addr\_t p, size\_t size)

Commit a reserved block on an allocation point.

ap is an allocation point.
p points to a block that was reserved by \texttt{mps\_reserve()} but has not yet been committed.

size is the size of the block to allocate. It must be the same size that was passed to \texttt{mps\_reserve()}.

If \texttt{mps\_commit()} returns true, the block was successfully committed, which means that the client program may use it, create references to it, and rely on references from it. It also means that the MPS may scan it, move it, protect it, or reclaim it (if ap was attached to a pool with those features).

If \texttt{mps\_commit()} returns false, the block was not committed. This means that the client program must not create references to the block, rely on references from it, or otherwise use it. It is normal to attempt the reserve operation again when this happens.

It is very rare for \texttt{mps\_commit()} to return false: this only happens if there was a \texttt{flip} between the call to \texttt{mps\_reserve()} and the call to \texttt{mps\_commit()}. Nonetheless, it can happen, so it is important not to perform operations with side effects (that you aren’t prepared to repeat) between calling \texttt{mps\_reserve()} and \texttt{mps\_commit()}. Also, the shorter the interval, the less likely \texttt{mps\_commit()} is to return false.

\textbf{Note}

\texttt{mps\_commit()} must only be called according to the Allocation point protocol.

\texttt{mps\_commit()} is implemented as a macro for speed. It may evaluate its arguments multiple times.
\end{verbatim}

### 2.6.4 Example: allocating a symbol

```c
typedef struct symbol_s {
    type_t type;    /* TYPE_SYMBOL */
    size_t length;  /* length of symbol string (excl. NUL) */
    char string[1]; /* symbol string, NUL terminated */
} symbol_s, *symbol_t;

symbol_t make_symbol(size_t length, char string[])
{
    symbol_t symbol;
    mps_addr_t addr;
    size_t size = ALIGN(offsetof(symbol_s, string) + length+1);
    do {
        mps_res_t res = mps_reserve(&addr, ap, size);
        if (res != MPS_RES_OK) error("out of memory in make_symbol");
        symbol = addr;
        symbol->type = TYPE_SYMBOL;
        symbol->length = length;
        memcpy(symbol->string, string, length+1);
    } while (!mps_commit(ap, addr, size));
    return symbol;
}
```
2.6.5 Cautions

While a block is reserved but not yet committed:

1. The client program must not create an *exact reference* to the reserved block (for example, by referring to the reserved block from a *formatted object*). All references to it must be ambiguous (for example, local variables).

2. Similar restrictions apply to a reference that has been stored in the reserved block. Such a reference might be invalid, and must not be copied to an *exact reference* or dereferenced. It is safe to copy such a reference if it remains ambiguous (for example, copying to a local variable or to another part of the new block).

Before calling `mps_commit()`:

1. The new block must be validly formatted. If it belongs to an *object format*, then it must be correctly recognized by the format methods (the *skip method* must return the object’s correct size; the *scan method* must scan it; the *is-forwarded method* must report that it is not a forwarding object, and so on).

2. All exact references in the new block (references that are *fixed* by scanning functions) must contain valid references or null pointers.

3. The new object must be ambiguously *reachable*.

You do not have to initialize the whole block so long as you satisfy these conditions. For example, it is permissible to defer initialization completely (for example, by writing `TYPE_UNINITIALIZED` into a tag field), so long as you handle this correctly in the format methods.

However, if you do not initialize the whole block then you should beware: the uninitialized contents of the block is likely to consist of dead objects. If, due to a bug, you created an exact reference into the middle of the uninitialized block, this might by bad luck point to a dead object, which would be resurrected (and it might well contain further exact references to other dead objects). To ensure detection of such a bug promptly you should consider filling the uninitialized object with dummy values that cannot be mistaken for part of a valid formatted object (at least in the debugging version of your program).

**Note**

Some *pool classes* have debugging counterparts that automatically overwrite free space with a pattern of bytes of your choosing. See *Debugging pools*.

2.6.6 Example: inserting into a doubly linked list

This example contains several mistakes. See the highlighted lines:

```c
typedef struct link_s {
    type_t type;        /* TYPE_LINK */
    /* all three of these pointers are fixed: */
    struct link_s *prev;
    struct link_s *next;
    obj_t obj;
} link_s, *link_t;

/* insert 'obj' into the doubly-linked list after 'head' */
link_t insert_link(link_t head, obj_t obj)
{
    mps_addr_t p;
    link_t link;
    size_t size = ALIGN(sizeof(link_s));
    do {
        mps_res_t res = mps_reserve(sp, ap, size);
```

2.6 Allocation
The mistakes are:

1. Dereferencing a reference (here, `link->prev`) that was stored in the reserved block.
2. Making an exact reference to the reserved block (here, `head->next` becomes an exact reference to `link`). This must be deferred until after a successful commit.
3. This line makes both mistakes made by lines (1) and (2).
4. The `obj` slot contains an exact reference that gets fixed by the scan method, so it must be initialized before the call to commit.

A correct version of `insert_link` looks like this:

```c
link_t insert_link(link_t head, obj_t obj)
{
    mps_addr_t p;
    link_t link;
    size_t size = ALIGN(sizeof(link_s));
    do {
        mps_res_t res = mps_reserve(&p, ap, size);
        if (res != MPS_RES_OK) error("out of memory");
        link = p;
        link->type = TYPE_LINK;
        link->prev = head;
        link->next = head->next;
        link->obj = obj;
    } while (!mps_commit(ap, p, size));
    head->next->prev = link;
    head->next = link;
    return link;
}
```

### 2.6.7 Allocation point implementation

An allocation point consists of a structure of type `mps_ap_s` and an associated buffer.

Fig. 2.1: Allocation point and its associated buffer.

The buffer is structured as shown in the figure, with free space at the end of the buffer, committed blocks at the beginning, and (possibly) one reserved block in the middle. The `mps_ap_s` structure contains three addresses into the associated buffer: limit points to the end of the buffer, alloc points to the beginning of the free space, and init points to the end of the initialized blocks.
Allocation points are fast and nearly lock-free because in order to reserve space for a new block, the client program first checks that \( \text{ap->alloc + size} \leq \text{ap->limit} \) and in the common case that it is, it takes a copy of \( \text{ap->init} \) (which now points to the reserved block) and sets \( \text{ap->alloc += size} \).

What happens when \( \text{ap->alloc + size} > \text{ap->limit} \), that is, when the new block won’t fit in the buffer? Then the buffer needs to be refilled by calling \text{mps_ap_fill()}\), with typical results shown in the diagram below.

Fig. 2.2: Allocation point after refilling.

Refilling is why allocation points are only nearly lock-free: \text{mps_ap_fill()}\) has to take locks on internal MPS data structures.

Note that \text{mps_ap_fill()}\) reserves the requested block as well as refilling the buffer.

The \text{reserve} operation thus looks like this:

```c
if (ap->alloc + size <= ap->limit) {
    ap->alloc += size;
    p = ap->init;
} else {
    res = mps_ap_fill(&p, ap, size);
    if (res != MPS_RES_OK) {
        /* handle error */
    }
}
```

The critical path consists of three loads, an add, two stores, and a branch (and branch prediction should work well since the test usually succeeds).

\[\text{Note}\]

Normally the client program would use the macro \text{mps_reserve()}\) to perform this operation, as described above, rather than directly accessing the fields of the allocation point structure. But there are use cases where direct access is needed to generate the fastest code (for example, in the case of a compiler generating machine code that needs to interface with the MPS), and it is for these use cases that the details of \text{mps_ap_s} are made public and supported.

When the new block has been initialized it must be committed. To do this, set \( \text{ap->init = ap->alloc} \) and then check to see if the allocation point has been trapped: that is, if the garbage collector might have moved some objects since the new block was reserved. The garbage collector traps an allocation point by setting \( \text{ap->limit = 0} \), so if this case is found, then the reserved block may have been invalidated, and must be discarded and re-reserved, and the buffer must be refilled. The function \text{mps_ap_trip()}\) determines whether or not this case applies, returning true if the block is valid, false if not.

The \text{commit} operation thus looks like this:

```c
ap->init = ap->alloc;
if (ap->limit == 0 && !mps_ap_trip(ap, p, size)) {
    /* p is invalid */
} else {
    /* p is valid */
}
```

The critical path here consists of three loads, a store and a branch (and again, branch prediction should work well since the test almost never fails).

\[\text{Note}\]

Normally the client program would use \text{mps_commit()}\) to perform this operation, as described above, rather than directly accessing the fields of the allocation point structure. But direct access is supported by the MPS.

2.6. Allocation
Note

The commit operation relies on atomic ordered access to words in memory to detect a flip that occurs between the assignment `ap->init = ap->alloc` and the test `ap->limit == 0`. A compiler or processor that reordered these two instructions would break the protocol. On some processor architectures and some compilers, it may be necessary to insert a memory barrier instruction at this point.

### mps_ap_s

The type of the structure used to represent allocation points:

```c
typedef struct mps_ap_s {
    mps_addr_t init;
    mps_addr_t alloc;
    mps_addr_t limit;
    /* ... private fields ... */
} mps_ap_s;
```

- `init` is the limit of initialized memory.
- `alloc` is the limit of allocated memory.
- `limit` is the limit of available memory.

An allocation point is an interface to a pool which provides very fast allocation, and defers the need for synchronization in a multi-threaded environment.

Create an allocation point for a pool by calling `mps_ap_create_k()`, and allocate memory via one by calling `mps_reserve()` and `mps_commit()`.

```c
mps_res_t mps_ap_fill (mps_addr_t *p_o, mps_ap_t ap, size_t size)

Reserve a block of memory on an allocation point when the allocation point has insufficient space.

`mps_ap_fill()` has same interface as `mps_reserve()`.
```

Note

`mps_ap_fill()` must only be called according to the Allocation point protocol.

```c
mps_bool_t mps_ap_trip (mps_ap_t ap, mps_addr_t p, size_t size)

Test whether a reserved block was successfully committed (2) when an allocation point was trapped.

`mps_ap_trip()` has the same interface as `mps_commit()`.
```

Note

`mps_ap_trip()` must only be called according to the Allocation point protocol.

### 2.7 Object formats

The need for some means of describing objects in the client program comes from tracing and moving. During tracing, when an object is scanned, all the references in the object must be identified so that the objects they point to can be scanned in their turn. When an object has moved, references to that object must be identified so that they can be updated to point to the new location of the object.
In general, only the client program can say which fields in an object are references, and only the client program knows how references are represented (for example, are they tagged?). Object formats provide the means by which the client program communicates this information to the MPS.

An object format is a collection of format methods and other (usually scalar) values which together describe programmatically the layout of objects belonging to the format. Format methods include the skip method (which calculates an object’s size), the scan method (which fixes references in the object), and the forward method (which replaces an object that has moved with a forwarding object).

Not every pool class supports formatted objects.

### 2.7.1 Interface

**mps_fmt_t**

The type of an object format.

```c
void mps_fmt_create_k (mps_fmt_t *fmt_o, mps_arena_t arena, mps_arg_s args[])
```

Create an object format.

- `fmt_o` points to a location that will hold the address of the new object format.
- `arena` is the arena in which to create the format.
- `args` are keyword arguments describing the format. Each pool class requires a particular subset of these keyword arguments: see the documentation for that pool class.

- `MPS_KEY_FMT_ALIGN` (type `mps_align_t`, default `MPS_PF_ALIGN`) is an integer value specifying the alignment of objects allocated with this format. It should be large enough to satisfy the alignment requirements of any field in the objects, and it must not be larger than the pool alignment.

- `MPS_KEY_FMT_HEADER_SIZE` (type `size_t`, default 0) is an integer value specifying the header size for objects with in-band headers. See In-band headers below.

- `MPS_KEY_FMT_SCAN` (type `mps_fmt_scan_t`) is a scan method that identifies references within objects belonging to this format. See `mps_fmt_scan_t`.

- `MPS_KEY_FMT_SKIP` (type `mps_fmt_skip_t`) is a skip method that skips over objects belonging to this format. See `mps_fmt_skip_t`.

- `MPS_KEY_FMT_FWD` (type `mps_fmt_fwd_t`) is a forward method that stores relocation information for an object belonging to this format that has moved. See `mps_fmt_fwd_t`.

- `MPS_KEY_FMT_ISFWD` (type `mps_fmt_isfwd_t`) is a is-forwarded method that determines if an object belonging to this format has been moved. See `mps_fmt_isfwd_t`.

- `MPS_KEY_FMT_PAD` (type `mps_fmt_pad_t`) is a padding method that creates padding objects belonging to this format. See `mps_fmt_pad_t`.

- `MPS_KEY_FMT_CLASS` (type `mps_fmt_class_t`) is a method that returns an address that is related to the class or type of the object, for inclusion in the telemetry stream for some events relating to the object. See `mps_fmt_class_t`.

`mps_fmt_create_k()` returns `MPS_RES_OK` if successful. The MPS may exhaust some resource in the course of `mps_fmt_create_k()` and will return an appropriate result code if so.

The object format pointed to by `fmt_o` persists until it is destroyed by calling `mps_fmt_destroy()`.

For example:

```c
MPS_ARGS_BEGIN(args) {
    MPS_ARGS_ADD(args, MPS_KEY_FMT_ALIGN, ALIGNMENT);
    MPS_ARGS_ADD(args, MPS_KEY_FMT_SCAN, obj_scan);
}
```
void mps_fmt_destroy (mps_fmt_t fmt)
    Destroy an object format.

    fmt is the object format to destroy.

    It is an error to destroy an object format if there exists a pool using the format. The pool must be destroyed first.

2.7.2 In-band headers

There are use cases in which it is convenient for the client program’s pointers to point some distance into the memory block containing the object. This typically happens when the objects have a common in-band header used for memory management or class system purposes, but this situation also arises when the low bits of a pointer are used for a tag. The MPS does not care what the reason is, only about the offset of the pointer in relation to the memory block.

If you have one of these use cases, you should pass the MPS_KEY_FMT_HEADER_SIZE keyword argument to mps_fmt_create_k(), specifying the size of the header: that is, the offset of a client pointer from the base of the memory block.

There are some cautions to be observed when using in-band headers:

1. The format methods (other than the padding method) receive client pointers (that is, pointers past the header) but all other MPS functions expect to receive and return base pointers (that is, pointers to the base of the block where the header is stored).

   In particular, mps_reserve() and mps_alloc() always hand out base pointers, and mps_free() expects to receive one.

2. Formatted objects must be longer than the header. In other words, objects consisting of only a header are not supported.

3. Even if the header size is larger than or equal to alignment, the padding method must still be able to create padding objects down to the alignment size.

4. Not all pool classes support objects with in-band headers. See the documentation for the pool class.

Note

A client program that allocates objects with in-band headers has to make a choice about how to represent references to those objects. It can represent them using base pointers (which is convenient for allocation, since mps_reserve() returns a base pointer, but requires decoding when scanning) or using client pointers (which is convenient for scanning, since the scan method takes a client pointer, but requires encoding on allocation). Either approach will work, but client pointers are normally the better choice, since scanning is normally more performance-critical than allocation.

2.7.3 Cautions

1. The MPS guarantees that format methods have exclusive access to the object for the duration of the call. This guarantee may entail suspending arbitrary threads. The methods that manipulate the object must not perform any sort of inter-thread locking or communication.
2. The MPS may call format methods in the context of an exception handler or a signal handler. For example, the following sequence of events is common:

(a) the MPS places a read barrier on a block of memory;
(b) the client program attempts to read from this block;
(c) the hardware raises a protection fault;
(d) the MPS signal handler is called;
(e) the MPS ensures that the contents of the block are correct and consistent: this may involve inspection of formatted objects in the block (or indeed, elsewhere), and so
(f) the MPS calls format methods.

Therefore, the format methods must be able to be run at any time, including asynchronously or in parallel with the rest of the program.

3. Format methods must be re-entrant.

4. Format methods must use no more than 64 words of stack space.

This restriction is necessary to avoid stack overflow in the MPS; see design-sp for details. If your application has format methods that need more stack space than this, contact us.

5. Format methods must not:

(a) call library code;
(b) perform a non-local exit (for example, by calling longjmp);
(c) call any functions in the MPS other than the fix functions (mps_fix(), MPS_FIX1(), MPS_FIX12(), and MPS_FIX2()).

It’s permissible to call other functions in the client program, but see MPS_FIX_CALL() for a restriction on passing the scan state.

6. Subject to the above constraints, format methods can freely access:

(a) memory inside the object or block that they have been asked to look at;
(b) memory managed by the MPS that is in pools that do not protect their contents;
(c) memory not managed by the MPS;

They must not access other memory managed by the MPS.

### 2.7.4 Format methods

`mps_addr_t (*mps_fmt_class_t)(mps_addr_t addr)`

The type of the class method of an object format.

*addr* is the address of the object whose class is of interest.

Returns an address that is related to the class or type of the object, or a null pointer if this is not possible.

It is recommended that a null pointer be returned for padding objects and forwarding objects.

`void (*mps_fmt_fwd_t)(mps_addr_t old, mps_addr_t new)`

The type of the forward method of an object format.

*old* is the address of an object.

*new* is the address to where the object has been moved.
The MPS calls the forward method for an object format when it has relocated an object belonging to that format. The forward method must replace the object at old with a forwarding marker that points to the address ‘new’. The forwarding marker must meet the following requirements:

1. It must be possible for the MPS to call other methods in the object format (the scan method, the skip method and so on) with the address of a forwarding marker as the argument.

2. The forwarding marker must be the same size as the old object. That is, when the skip method is called on the forwarding marker, it must return the same address as when it was called on the old object.

3. It must be possible for the is-forwarded method of the object format to distinguish the forwarding marker from ordinary objects, and the is-forwarded method method must return the address new. See mps_fmt_isfwd_t.

Note
This method is never invoked by the garbage collector on an object in a non-moving pool.

mps_addr_t (*mps_fmt_isfwd_t) (mps_addr_t addr)
The type of the is-forwarded method of an object format.
addr is the address of a candidate object.

If the addr is the address of a forwarding object, return the address where the object was moved to. This must be the value of the new argument supplied to the forward method when the object was moved. If not, return a null pointer.

Note
This method is never invoked by the garbage collector on an object in a non-moving pool.

void (*mps_fmt_pad_t) (mps_addr_t addr, size_t size)
The type of the padding method of an object format.
addr is the address at which to create a padding object.

size is the size of the padding object to be created.

The MPS calls a padding method when it wants to create a padding object. Typically the MPS creates padding objects to fill in otherwise unused gaps in memory; they allow the MPS to pack objects into fixed-size units (such as operating system pages).

The padding method must create a padding object of the specified size at the specified address. The size can be any aligned (to the format alignment) size. A padding object must be acceptable to other methods in the format (the scan method, the skip method, and so on).

Note
The padding method always receives a base pointer, even if the object format has a non-zero MPS_KEY_FMT_HEADER_SIZE.

mps_res_t (*mps_fmt_scan_t) (mps_ss_t ss, mps_addr_t base, mps_addr_t limit)
The type of the scan method of an object format.

ss is the scan state. It must be passed to MPS_SCAN_BEGIN() and MPS_SCAN_END() to delimit a sequence of fix operations, and to the functions MPS_FIX1() and MPS_FIX2() when fixing a reference.

base points to the first formatted object in the block of memory to be scanned.

limit points to the location just beyond the end of the block to be scanned. Note that there might not be any object at this location.
Returns a result code. If a fix function returns a value other than \texttt{MPS\_RES\_OK}, the scan method must return that value, and may return without fixing any further references. Generally, it is better if it returns as soon as possible. If the scanning is completed successfully, the function should return \texttt{MPS\_RES\_OK}.

The scan method for an object format is called when the MPS needs to scan objects in a block of memory containing objects belonging to that format. The scan method is called with a scan state and the base and limit of the block of objects to scan. It must then indicate references within the objects by calling \texttt{MPS\_FIX1()} and \texttt{MPS\_FIX2()}.

If the object format is capable of creating forwarding objects or padding objects, the scan method must be able to scan these objects. (In the case of the forwarding object, the scan method should not fix the pointer to the new location.)

See also: Scanning.

\texttt{mps\_addr\_t (\*mps\_fmt\_skip\_t) (mps\_addr\_t addr)}

The type of the skip method of an object format.

\texttt{addr} is the address of the object to be skipped.

Returns the address of the “next object”. In an object format without in-band headers, this is the address just past the end of this object. In an object format with in-band headers, it’s the address just past where the header of next object would be, if there were one.

Note

In either case, the result is the sum of \texttt{addr} and the size of the block containing the object.

If the object format is capable of creating forwarding objects or padding objects, the skip method must be able to skip these objects.

A skip method is not allowed to fail.

Note

The MPS uses this method to determine the size of objects (by subtracting \texttt{addr} from the result) as well as skipping over them.

\subsection{2.7.5 Object format introspection}

\texttt{mps\_bool\_t mps\_addr\_fmt (mps\_fmt\_t *fmt\_o, mps\_arena\_t arena, mps\_addr\_t addr)}

Determine the object format to which an address belongs.

\texttt{fmt\_o} points to a location that will hold the address of the object format, if one is found.

\texttt{arena} is the arena whose object formats will be considered.

\texttt{addr} is the address.

If \texttt{addr} is the address of a location inside a block allocated from a pool in \texttt{arena}, and that pool has an object format, then update the location pointed to by \texttt{fmt\_o} with the address of the object format, and return true.

If \texttt{addr} is the address of a location inside a block allocated from a pool in \texttt{arena}, but that pool has no object format, return false.

If \texttt{addr} points to a location that is not managed by \texttt{arena}, return false.

If none of the above conditions is satisfied, \texttt{mps\_addr\_fmt()} may return either true or false.
Note
This function might return a false positive by returning true if you ask about an address that happens to be inside memory managed by a pool with an object format, but which is not inside a block allocated by that pool. It never returns a false negative.

```c
void mps_arena_formatted_objects_walk (mps_arena_t arena, mps_formatted_objects_stepper_t f, void *p, size_t s)
```

Visit all formatted objects in an arena. 
arena is the arena whose formatted objects you want to visit. 
f is a formatted objects stepper function. It will be called for each formatted object in the arena. See mps_formatted_objects_stepper_t.
p and s are arguments that will be passed to f each time it is called. This is intended to make it easy to pass, for example, an array and its size as parameters. 
Each pool class determines for which objects the stepper function is called. Typically, all validly formatted objects are visited. Padding objects may be visited at the pool class’s discretion: the stepper function must handle this case.

Note
This function is intended for heap analysis, tuning, and debugging, not for frequent use in production.

---

**Warning:** If a garbage collection is currently in progress (that is, if the arena is in the clamped or unclamped state), then only objects that are known to be currently valid are visited. 
For the most reliable results, ensure the arena is in the parked state by calling mps_arena_park() before calling this function (and release it by calling mps_arena_release() afterwards, if desired).

```c
void (*mps_formatted_objects_stepper_t) (mps_addr_t addr, mps_fmt_t fmt, mps_pool_t pool, void *p, size_t s)
```

The type of a formatted objects stepper function.
A function of this type can be passed to mps_arena_formatted_objects_walk(), in which case it will be called for each formatted object in an arena. It receives five arguments:
addr is the address of the object. 
fmt is the object format for that object. 
pool is the pool to which the object belongs. 
p and s are the corresponding values that were passed to mps_arena_formatted_objects_walk().
The function may not call any function in the MPS. It may access:
1. memory inside the object or block pointed to by addr;
2. memory managed by the MPS that is in pools that do not protect their contents;
3. memory not managed by the MPS;
It must not access other memory managed by the MPS.

### 2.8 Scanning

Scanning is the process of identifying the references in a block of memory and “fixing” them. It’s the process at the heart of the Memory Pool System, and the most critical of the memory management functions that have to be
implemented by the client program.

Scanning is performed for two tasks: during tracing, blocks are scanned in order to follow references, and so determine which blocks are reachable and which are not. After objects have been moved in memory, blocks are scanned in order to identify references that need to be updated to point to the new locations of these objects. Both tasks use the same scanning protocol, described here.

### 2.8.1 Scanning protocol

There are several types of scanning functions (the scan method in an object format, of type `mps_fmt_scan_t`, and root scanning functions of various types) but all take a scan state argument of type `mps_ss_t`, and a description of a region to be scanned. They must carry out the following steps:

1. Call the macro `MPS_SCAN_BEGIN()` on the scan state.
2. For each reference in the region:
   1. Call `MPS_FIX1()`, passing the scan state and the reference.
   2. If `MPS_FIX1()` returns false, the reference is not of interest to the MPS. Proceed to the next reference in the region.
   3. If `MPS_FIX1()` returns true, the reference is of interest to the MPS. Call `MPS_FIX2()`, passing the scan state and a pointer to a location containing the reference.
   4. If `MPS_FIX2()` returns a result code other than `MPS_RES_OK`, return this result code from the scanning function as soon as practicable.
   5. If `MPS_FIX2()` returns `MPS_RES_OK`, it may have updated the reference. Make sure that the updated reference is stored back into the region being scanned.
3. Call the macro `MPS_SCAN_END()` on the scan state.
4. Return `MPS_RES_OK`.

This description of the protocol simplifies a number of important details, which are covered in the following sections.

### 2.8.2 Tagged references

If your references are tagged (or otherwise “encrypted”), then you must remove the tag (or decrypt them) before passing them to `MPS_FIX1()` and `MPS_FIX2()`.

The reference passed to `MPS_FIX2()` must be the address of the base of the block referred to (unless the referent belongs to an object format with in-band headers, in which case it must be a reference to the address just after the header).

However, `MPS_FIX1()` allows some leeway: if you pass it a reference to the interior of an allocated block, then `MPS_FIX1()` correctly determines whether a reference to the block is of interest to the MPS.

This means that if your tag is in the low bits of the reference, you may not have to remove it before calling `MPS_FIX1()`. For example, if you use three tag bits, then your reference is at most `base + 7`, and if your objects are at least 8 bytes long, then the reference is within the object and need not be stripped. So your code might look like this:

```c
if (MPS_FIX1(ss, obj->ref)) {
    /* strip the tag */
    mps_addr_t p = obj->ref & ~0x7;
    mps_res_t res = MPS_FIX2(ss, &p);
    if (res != MPS_RES_OK) return res;
    /* restore the tag and update reference */
```
```c
mps_word_t tag = obj->ref & 0x7;
obj->ref = (obj_t)((char*)p + tag);
```

This saves the cost of stripping the tag in the case that `obj->ref` is not of interest to the MPS.

Similarly, if you use interior pointers, you do not need to convert them to base pointers before calling `MPS_FIX1()` (or, indeed, before calling `MPS_FIX2()`, if the target of the referent belongs to an object format with in-band headers).

### 2.8.3 Critical path

Scanning is an operation on the critical path of the MPS and so it is vital that it runs fast. The scanning protocol is designed to ensure that as much of the scanning code can be run inline in the client program as possible. In particular, the macro `MPS_FIX1()` does not need to call into the MPS.

The purpose of `MPS_FIX1()` is to provide a fast check as to whether a reference is “of interest” to the MPS. It is legitimate to call this on any word: it does not even have to be an address. So if you have a mixture of references and non-references, it might turn out to be faster to call `MPS_FIX1()` on each word before you even determine whether or not the word is a reference.

Whether this is in fact an optimization depends on the proportion of references to non-references, on how often genuine references turn out to be “of interest”, and what kind of code the compiler has generated. There is no substitute for measurement.

See design-critical-path.

**Note**

In one application with a high proportion of unboxed values, it turned out to be fastest to check the tag and reject non-references before calling `MPS_FIX1()`.

**Warning:** If you passed a word that might not be a reference to `MPS_FIX1()`, and it returned true, this might be a false positive. You must be certain that the alleged reference is genuine as well as “of interest” before passing it to `MPS_FIX2()`.

Another technique that can speed up scanning is to segregate objects into pools whose object formats contain different scan methods. In particular, if you can segregate objects that do not contain any references into leaf object pools like AMCZ (Automatic Mostly-Copying Zero-rank), these objects do not need to be scanned at all.

### 2.8.4 Ambiguous references

If the references in the object being scanned are ambiguous then `MPS_FIX2()` does not update the reference (because it can’t know if it’s a genuine reference). The MPS handles an ambiguous reference by pinning the block pointed to so that it cannot move.

You could use this fact to optimize the scan by avoiding the need to reassemble and store the updated reference after calling `MPS_FIX2()`.

**Note**

The MPS currently has no pools that support ambiguous references, so this cannot arise for the scan method in an object format, but root scanning functions may encounter this case.
2.8.5 Unfixed references

The MPS does not require you to fix all your references. But if a reference is not fixed:

1. it does not keep its target alive (this might be acceptable if you know that the target is being kept alive for another reason, for example if it is in a manually managed pool, or if there is always another reference to the target that is fixed);

2. it does not get updated if the target moves (this might be acceptable if you know that the target cannot move, for example if it is in a non-moving pool, or if it is pinned by an ambiguous reference).

These optimizations can be tricky to make correct, and can make the system fragile (for example, it may break if you start using a different pool class), so it is usually safest to fix all references.

2.8.6 Example: Scheme objects

Scanning tends to be a repetitive procedure and so you’ll find it is usually helpful to define macros to reduce the size of the source code. The MPS provides a convenience macro MPS_FIX12() for the common case of calling MPS_FIX1() and then immediately calling MPS_FIX2() if the reference is “of interest”.

Note

Some compilers generate better code if you use MPS_FIX12(), and some if you use MPS_FIX1() and MPS_FIX2(). There’s no substitute for measurement.

Here’s the macro FIX defined by the toy Scheme interpreter:

```
#define FIX(ref) \ 
    do { \ 
      mps_addr_t _addr = (ref); /* copy to local to avoid type pun */ \ 
      mps_res_t res = MPS_FIX12(ss, &_addr); \ 
      if (res != MPS_RES_OK) return res; \ 
      (ref) = _addr; \ 
    } while(0)
```

Note

The comment refers to a temptation to write non-portable code that presents itself here. MPS_FIX2() takes a pointer to a location containing the reference (an argument of type mps_addr_t *). It is tempting to take the address of the reference and cast it to this type. The behaviour of such a cast is not defined by the C standard. See Type punning.

Here’s the Scheme scanner:

```
static mps_res_t obj_scan(mps_ss_t ss, mps_addr_t base, mps_addr_t limit)
{
    MPS_SCAN_BEGIN(ss) {
        while (base < limit) {
            obj_t obj = base;
            switch (obj->type.type) {
                case TYPE_PAIR:
                    FIX(obj->pair.car);
                    FIX(obj->pair.cdr);
                    base = (char *)base + ALIGN(sizeof(pair_s));
                    break;
                case TYPE_VECTOR:
                    {
                        size_t i;
                        for (i = 0; i < obj->vector.length; ++i)
```
```c
FIX(obj->vector.vector[i]);
base = (char *)base +
    ALIGN(offsetof(vector_s, vector) +
          obj->vector.length * sizeof(obj->vector.vector[0]));
break;
} /* ... and so on for the other types ... */
default:
    assert(0);
    fprintf(stderr, "Unexpected object on the heap\n"); abort();
    return MPS_RES_FAIL;
}
} MPS_SCAN_END(ss);
return MPS_RES_OK;
```

**Note**
This scanner is a simple example intended to make the process clear to the reader. The scanning code and the object layout are not at all optimized.

### 2.8.7 Scanning interface

**mps_ss_t**
The type of *scan states*.

A scan state represents the state of the current *scan*. The MPS passes a scan state to the *scan method* of an *object format* when it needs to *scan* for references within a region of memory. The scan method must pass the scan state to `MPS_SCAN_BEGIN()` and `MPS_SCAN_END()` to delimit a sequence of fix operations, and to the functions `MPS_FIX1()`, `MPS_FIX2()` and `MPS_FIX12()` when fixing a *reference*.

**MPS_SCAN_BEGIN** *(mps_ss_t ss)*
Within a *scan method*, set up local information required by `MPS_FIX1()`, `MPS_FIX2()` and `MPS_FIX12()`.
The local information persists until `MPS_SCAN_END()`.

*ss* is the *scan state* that was passed to the scan method.

**Note**
Between `MPS_SCAN_BEGIN()` and `MPS_SCAN_END()`, the scan state is in a special state, and must not be passed to a function. If you really need to do so, for example because you have an embedded structure shared between two scan methods, you must wrap the call with `MPS_FIX_CALL()` to ensure that the scan state is passed correctly.

**MPS_SCAN_END** *(mps_ss_t ss)*
Within a *scan method*, terminate a block started by `MPS_SCAN_BEGIN()`.

*ss* is the *scan state* that was passed to the scan method.

**Note**
`MPS_SCAN_END()` ensures that the scan is completed, so successful termination of a scan must invoke it. However, in case of an error it is allowed to return from the scan method without invoking `MPS_SCAN_END()`.
**Note**

Between `MPS_SCAN_BEGIN()` and `MPS_SCAN_END()`, the scan state is in a special state, and must not be passed to a function. If you really need to do so, for example because you have an embedded structure shared between two scan methods, you must wrap the call with `MPS_FIX_CALL()` to ensure that the scan state is passed correctly.

```c
MPS_FIX_CALL(ss, call)
```

Call a function to do some scanning, from within a scan method, between `MPS_SCAN_BEGIN()` and `MPS_SCAN_END()`, passing the scan state correctly.

- `ss` is the scan state that was passed to the scan method.
- `call` is an expression containing a function call where `ss` is one of the arguments.

Between `MPS_SCAN_BEGIN()` and `MPS_SCAN_END()`, the scan state is in a special state, and must not be passed to a function. If you really need to do so, for example because you have a structure shared between two object formats, you must wrap the call with `MPS_FIX_CALL()` to ensure that the scan state is passed correctly.

The function being called must use `MPS_SCAN_BEGIN()` and `MPS_SCAN_END()` appropriately.

In example below, the scan method `obj_scan` fixes the object's left and right references, but delegates the scanning of references inside the object's data member to the function `data_scan`. In order to ensure that the scan state is passed correctly to `data_scan`, the call must be wrapped in `MPS_FIX_CALL()`.

```c
mps_res_t obj_scan(mps_ss_t ss, mps_addr_t base, mps_addr_t limit) {
    obj_t obj;
    mps_res_t res;
    MPS_SCAN_BEGIN(ss) {
        for (obj = base; obj < limit; obj++) {
            res = MPS_FIX12(ss, &obj->left);
            if (res != MPS_RES_OK)
                return res;
            MPS_FIX_CALL(ss, res = data_scan(ss, &obj->data));
            if (res != MPS_RES_OK)
                return res;
            res = MPS_FIX12(ss, &obj->right);
            if (res != MPS_RES_OK)
                return res;
        }
    } MPS_SCAN_END(ss);
    return MPS_RES_OK;
}
```

**Warning:** Use of `MPS_FIX_CALL()` is best avoided, as it may force values out of registers (depending on compiler optimisations such as inlining). The gains in simplicity of the code ought to be measured against the loss in performance.

### 2.8.8 Fixing interface

```c
mps_bool_t MPS_FIX1(mps_ss_t ss, mps_addr_t ref)
```

Determine whether a reference needs to be passed to `MPS_FIX2()`.

- `ss` is the scan state that was passed to the scan method.
- `ref` is the reference.
Returns a truth value (mps_bool_t) indicating whether ref is “interesting” to the MPS. If it returns true, the scan method must invoke MPS_FIX2() to fix ref.

This macro must only be used within a scan method, between MPS_SCAN_BEGIN() and MPS_SCAN_END().

**Note**
If your reference is tagged or otherwise “encrypted”, you must ensure that it points to a location within the target block before calling MPS_FIX1(). (Therefore, a small tag in the low bits need not be stripped.)

This macro is a convenience for the case where MPS_FIX1() is immediately followed by MPS_FIX2(). The interface is the same as MPS_FIX2().

**Note**
In the case where the scan method does not need to do anything between MPS_FIX1() and MPS_FIX2(), you can use the convenience macro MPS_FIX12().

```c
mps_res_t MPS_FIX12 (mps_ss_t ss, mps_addr_t *ref_io)
Fix a reference.
This macro is a convenience for the case where MPS_FIX1() is immediately followed by MPS_FIX2(). The interface is the same as MPS_FIX2().
```

```c
mps_res_t MPS_FIX2 (mps_ss_t ss, mps_addr_t *ref_io)
Fix a reference.
ss is the scan state that was passed to the scan method.
ref_io points to the reference.
Returns MPS_RES_OK if successful. In this case the reference may have been updated, and so the scan method must store the updated reference back to the region being scanned. The scan method must continue to scan the block.
If it returns any other result, the scan method must return that result as soon as possible, without fixing any further references.
This macro must only be used within a scan method, between MPS_SCAN_BEGIN() and MPS_SCAN_END().
```

**Note**
If your reference is tagged (or otherwise “encrypted”), you must remove the tag (or otherwise decrypt the reference) before calling MPS_FIX2(), and restore the tag to the (possibly updated) reference afterwards.
The only exception is for references to objects belonging to a format with in-band headers: the header size must not be subtracted from these references.

**Note**
In the case where the scan method does not need to do anything between MPS_FIX1() and MPS_FIX2(), you can use the convenience macro MPS_FIX12().

---

### 2.9 Threads

#### 2.9.1 Thread safety

The MPS is designed to run in an environment with multiple threads all calling into the MPS. Some code is known to operate with exclusive access to the data it manipulates (for example, allocation via allocation points, in the common
case where the buffer does not need to be refilled, and location dependencies), so this code is safe. For the rest of the code, shared data structures are locked by the use of a single lock per arena. This lock is claimed on entry to the MPS and released on exit from it. So there is at most a single thread (per arena) running “inside” the MPS at a time.

### 2.9.2 Thread registration

In order to scan a thread’s registers for references (which happens at each flip), the MPS needs to be able to suspend that thread, and in order to gain exclusive atomic access to memory in order to scan it, the MPS needs to be able to suspend all threads that might access that memory. This means that threads must be registered with the MPS by calling`mps_thread_reg()` (and thread roots created; see Thread roots).

For simplicity, we recommend that a thread must be registered with an arena if:

- its registers and control stack form a root (this is enforced by `mps_root_create_reg()`); or
- it reads or writes from a location in an automatically managed pool in the arena.

However, some automatically managed pool classes may be more liberal than this. See the documentation for the pool class.

### 2.9.3 Signal and exception handling issues

**Warning:** On Unix platforms (except OS X), the MPS suspends and resumes threads by sending them signals. There’s a shortage of available signals that aren’t already dedicated to other purposes (for example, ValGrind uses `SIGUSR1` and `SIGUSR2`), so the MPS uses `SIGXCPU` and `SIGXFSZ`. This means that programs must not mask these two signals.

If your program needs to handle these signals, then it must co-operate with the MPS. At present, there’s no documented mechanism for co-operating: if you are in this situation, please contact us.

**Warning:** The MPS uses barriers (1) to protect memory from the client program and handles the signals that result from barrier hits.

- On Linux and FreeBSD, your program must not mask or handle `SIGSEGV`.
- On Windows, you must not install a first-chance exception handler.
- On OS X, you must not install a thread-local Mach exception handler for `EXC_BAD_ACCESS` exceptions.

All of these things are, in fact, possible, but your program must co-operate with the MPS. At present, there’s no documented mechanism for co-operating: if you are in this situation, please contact us.

### 2.9.4 Thread interface

**mps_thr_t**  
The type of registered thread descriptions.

In a multi-threaded environment where incremental garbage collection is used, threads must be registered with the MPS by calling `mps_thread_reg()` so that the MPS can suspend them as necessary in order to have exclusive access to their state.

Even in a single-threaded environment it may be necessary to register a thread with the MPS so that its stack can be registered as a root by calling `mps_root_create_reg()`.

**mps_res_t**  
Register the current thread with an arena.

`thr_o` points to a location that will hold the address of the registered thread description, if successful.
arena is the arena.

Returns MPS_RES_OK if successful, or another result code if not.

A thread must be registered with an arena if it ever uses a pointer to a location in an automatically managed pool belonging to that arena.

**Note**

It is recommended that all threads be registered with all arenas.

```c
void mps_thread_dereg (mps_thr_t thr)
```

Deregister a thread.

thr is the description of the thread.

After calling this function, the thread whose registration with an arena was recorded in thr must not read or write from a location in an automatically managed pool belonging to that arena.

**Note**

Some pool classes may be more liberal about what a thread may do after it has been deregistered. See the documentation for the pool class.

**Note**

It is recommended that threads be deregistered only when they are just about to exit.

## 2.10 Roots

Roots tell the garbage collector where to start tracing. The garbage collector determines which blocks are reachable from the roots, and (in automatically managed pools) reclaims the unreachable blocks. This is quite efficient and can be a very good approximation to liveness.

It is therefore important that all references that the client program can directly access are registered as roots, otherwise the garbage collector might recycle an object that would be used in the future. Some collectors, for example Boehm’s, assume that all references stored in static data are roots; the Memory Pool System is more flexible, but requires the client program to declare which references are roots.

### 2.10.1 Registering roots

You can register a root at any time by calling one of the mps_root_create functions. Roots may not be registered twice, and no two roots may overlap (that is, each reference is fixed by at most one root). Roots may be:

1. in registers;
2. on the program’s control stack;
3. in the program’s static data;
4. in heap not managed by the MPS (provided that you destroy the root before freeing it; see the Scheme interpreter’s global symbol table for an example);
5. in manually managed pools (provided that you remove the root before freeing it).
Roots must not be in memory that is subject to garbage collection (and so roots must not be in automatically managed pools).

When you register a root you describe to the MPS how to scan it for references, providing your own scanning function in the cases of `mps_root_create()` and `mps_root_create_fmt()`. Such a root scanning function must follow the Scanning protocol.

All the references in a root are of the same rank (just as in a formatted object). So they are all exact, ambiguous or weak.

**Note**

If the rank of the root is exact, or weak, the references in the root must always be valid while the root is registered: that is, they must be references to actual objects or null pointers. This could be immediately after the root is registered, so the root must be valid before it is registered.

**Note**

As with scanning in general, it’s safe to fix references that point to memory not managed by the MPS. These will be ignored.

Roots can be deregistered at any time by calling `mps_root_destroy()`. All roots registered in an arena must be deregistered before the arena is destroyed.

There are five ways to register a root, depending on how you need to scan it for references:

1. `mps_root_create()` if you need a custom root scanning function (of type `mps_root_scan_t`);
2. `mps_root_create_fmt()` if the root consists of a block of objects belonging to an object format, which can be scanned by the format’s scan method (of type `mps_fmt_scan_t`);
3. `mps_root_create_table()` if the root consists of a table of references;
4. `mps_root_create_table_masked()` if the root consists of a table of tagged references;
5. `mps_root_create_reg()` if the root consists of the registers and control stack of a thread. See Thread roots below.

### 2.10.2 Cautions

Creating a root and then registering is similar to reserving a block and then committing it (in the Allocation point protocol), and similar cautions apply. Before registering a root:

1. The root must be valid (that is, the appropriate root scanning function can scan it).
2. All exact references in the root (references that are fixed by the root scanning function) must contain valid references or null pointers.
3. You must not store a reference in the root to a block in an automatically managed pool (such a reference is hidden from the MPS until you register the root, and may become invalid).

So the typical sequence of operations when creating a root is:

1. Initialize references in the root with null pointers or other safe values.
2. Register the root.
3. Fill in the references in the root.
2.10.3 Thread roots

Every thread’s registers and control stack potentially contain references to allocated objects, so should be registered as a root by calling `mps_root_create_reg()`. It’s not easy to write a scanner for the registers and the stack: it depends on the operating system, the processor architecture, and in some cases on the compiler. For this reason, the MPS provides `mps_stack_scan_ambig()` (and in fact, this is the only supported stack scanner).

A stack scanner needs to know how to find the bottom of the part of the stack to scan. The bottom of the relevant part of stack can be found by taking the address of a local variable in the function that calls the main work function of your thread. You should take care to ensure that the work function is not inlined so that the address is definitely in the stack frame below any potential roots.

For example, here’s the code from the toy Scheme interpreter that registers a thread root and then calls the program:

```c
mps_thr_t thread;
mps_root_t reg_root;
int exit_code;
void *marker = NULL;

res = mps_thread_reg(&thread, arena);
if (res != MPS_RES_OK) error("Couldn't register thread");

res = mps_root_create_reg(&reg_root,
    arena,
    mps_rank_ambig(),
    0,
    thread,
    mps_stack_scan_ambig,
    marker,
    0);
if (res != MPS_RES_OK) error("Couldn't create root");

exit_code = start(argc, argv);
mps_root_destroy(reg_root);
mps_thread_dereg(thread);
```

2.10.4 Ranks

`mps_rank_t`  
The type of `ranks`. It is a transparent alias for `unsigned int`, provided for convenience and clarity.

`mps_rank_t mps_rank_ambig()` (void)  
Return the rank of ambiguous roots.

`mps_rank_t mps_rank_exact()` (void)  
Return the rank of exact roots.

`mps_rank_t mps_rank_weak()` (void)  
Return the rank of weak roots.

2.10.5 Root modes

The root mode provides a way for the client to declare various facts about a root that allow the MPS to make optimizations. Roots that are declared to be `constant` need not be re-scanned, and roots that are declared to be `protectable` may have barriers placed on them, allowing the MPS to detect whether they have changed.
Note
The MPS does not currently perform either of these optimizations, so root modes have no effect. These features may be added in a future release.

mps_rm_t
The type of \textit{root modes}.

It should be zero (meaning neither constant or protectable), or the sum of some of \texttt{MPS\_RM\_CONST}, \texttt{MPS\_RM\_PROT}, and \texttt{MPS\_RM\_PROT\_INNER}.

\texttt{MPS\_RM\_CONST}

\textbf{Deprecated}

starting with version 1.111.

This was introduced in the hope of being able to maintain a remembered set for the root without needing a write barrier, but it can’t work as described, since you can’t reliably create a valid registered constant root that contains any references. (If you add the references before registering the root, they may have become invalid; but you can’t add them afterwards because the root is supposed to be constant.)

The \textit{root mode} for constant roots. This tells the MPS that the \textit{client program} will not change the \textit{root} after it is registered: that is, scanning the root will produce the same set of references every time. Furthermore, for roots registered by \texttt{mps_root_create_fmt()} and \texttt{mps_root_create_table()}, the client program will not write to the root at all.

\texttt{MPS\_RM\_PROT}

The \textit{root mode} for protectable roots. This tells the MPS that it may place a barrier (1) on any page containing any part of the root. No format method or scan method (except for the one for this root) may write data in this root. They may read it.

Note
You must not specify \texttt{MPS\_RM\_PROT} on a root allocated by the MPS.

No page may contain parts of two or more protectable roots. You mustn’t specify \texttt{MPS\_RM\_PROT} if the \textit{client program} or anything other than (this instance of) the MPS is going to protect or unprotect the relevant pages.

This mode may not be suitable if the \textit{client program} wants the operating system to be able to access the root. Many operating systems can’t cope with writing to protected pages.

\texttt{MPS\_RM\_PROT\_INNER}

The \textit{root mode} for protectable roots whose inner pages (only) may be protected. This mode must not be specified unless \texttt{MPS\_RM\_PROT} is also specified. It tells the MPS that it may not place a barrier (1) on a page that’s partly (but not wholly) covered by the \textit{root}.

\subsection{2.10.6 Root interface}

\texttt{mps_root_t}
The type of \textit{root} descriptions.

The \texttt{arena} uses root descriptions to find \textit{references} within the \textit{client program’s} roots.

\texttt{mps_res_t mps_root_create}(mps_root_t *root_o, mps_arena_t arena, mps_rank_t rank, mps_rm_t rm, mps_root_scan_t root_scan, void *p, size_t s)

Register a \textit{root} that consists of the \textit{references} fixed by a scanning function.
root_o points to a location that will hold the address of the new root description.
arena is the arena.
rank is the rank of references in the root.
rm is the root mode.
root_scan is the root scanning function. See mps_root_scan_t.
p and s are arguments that will be passed to root_scan each time it is called. This is intended to make it easy to pass, for example, an array and its size as parameters.
Returns MPS_RES_OK if the root was registered successfully, MPS_RES_MEMORY if the new root description could not be allocated, or another result code if there was another error.
The registered root description persists until it is destroyed by calling mps_root_destroy().

mps_res_t (*mps_root_scan_t) (mps_ss_t ss, void *p, size_t s)
The type of root scanning functions for mps_root_create().
ss is the scan state. It must be passed to MPS_SCAN_BEGIN() and MPS_SCAN_END() to delimit a sequence of fix operations, and to the functions MPS_FIX1() and MPS_FIX2() when fixing a reference.
p and s are the corresponding values that were passed to mps_root_create().
Returns a result code. If a fix function returns a value other than MPS_RES_OK, the scan method must return that value, and may return without fixing any further references. Generally, it is better if it returns as soon as possible. If the scanning is completed successfully, the function should return MPS_RES_OK.

mps_res_t mps_root_create_fmt (mps_root_t *root_o, mps_arena_t arena, mps_rank_t rank, mps_rm_t rm, mps_fmt_scan_t fmt_scan, mps_addr_t base, mps_addr_t limit)
Register a root that consists of the references fixed by a scanning function in a block of formatted objects.
root_o points to a location that will hold the address of the new root description.
arena is the arena.
rank is the rank of references in the root.
rm is the root mode.
fmt_scan is a scanning function. See mps_fmt_scan_t.
base is the address of the base of the block of formatted objects.
limit is the address just beyond the end of the block of formatted objects.
Returns MPS_RES_OK if the root was registered successfully, MPS_RES_MEMORY if the new root description could not be allocated, or another result code if there was another error.
The registered root description persists until it is destroyed by calling mps_root_destroy().

mps_res_t mps_root_create_reg (mps_root_t *root_o, mps_arena_t arena, mps_rank_t rank, mps_rm_t rm, mps_thr_t thr, mps_reg_scan_t reg_scan, void *p, size_t s)
Register a root that consists of the references fixed in a thread’s stack by a scanning function.
root_o points to a location that will hold the address of the new root description.
arena is the arena.
rank is the rank of references in the root.
rm is the root mode.
thr is the thread.
reg_scan is a scanning function. See mps_reg_scan_t.

p and s are arguments that will be passed to reg_scan each time it is called. This is intended to make it easy to pass, for example, an array and its size as parameters.

Returns MPS_RES_OK if the root was registered successfully, MPS_RES_MEMORY if the new root description could not be allocated, or another result code if there was another error.

The registered root description persists until it is destroyed by calling mps_root_destroy().

Note
It is not supported for client programs to pass their own scanning functions to this function. The built-in MPS function mps_stack_scan_ambig() must be used.

This function is intended as a hook should we ever need to allow client-specific extension or customization of stack and register scanning. If you’re in a position where you need this, for example, if you’re writing a compiler and have control over what goes in the registers, contact us.

mps_res_t (*mps_reg_scan_t) (mps_ss_t ss, mps_thr_t thr, void *p, size_t s)
The type of a root scanning function for roots created with mps_root_create_reg().

ss is the scan state. It must be passed to MPS_SCAN_BEGIN() and MPS_SCAN_END() to delimit a sequence of fix operations, and to the functions MPS_FIX1() and MPS_FIX2() when fixing a reference.

thr is the thread.

p and s are the corresponding values that were passed to mps_root_create_reg().

Returns a result code. If a fix function returns a value other than MPS_RES_OK, the scan method must return that value, and may return without fixing any further references. Generally, it is better if it returns as soon as possible. If the scanning is completed successfully, the function should return MPS_RES_OK.

A root scan method is called whenever the MPS needs to scan the root. It must then indicate references within the root by calling MPS_FIX1() and MPS_FIX2().

See also:
Scanning.

Note
Client programs are not expected to write scanning functions of this type. The built-in MPS function mps_stack_scan_ambig() must be used.

mps_reg_scan_t mps_stack_scan_ambig()
A root scanning function for ambiguous scanning of threads, suitable for passing to mps_root_create_reg().

It scans all integer registers and everything on the stack of the thread given, and can therefore only be used with ambiguous roots. It only scans locations that are at, or higher on the stack (that is, more recently added), the stack bottom that was passed to mps_thread_reg(). References are assumed to be represented as machine words, and are required to be 4-byte-aligned; unaligned values are ignored.

Note
The MPS provides this function because it’s hard to write: it depends on the operating system, the processor architecture, and in some cases on the compiler.
Register a root that consists of a vector of references.

- `root_o` points to a location that will hold the address of the new root description.
- `arena` is the arena.
- `rank` is the rank of references in the root.
- `rm` is the root mode.
- `base` points to a vector of references.
- `count` is the number of references in the vector.

Returns `MPS_RES_OK` if the root was registered successfully, `MPS_RES_MEMORY` if the new root description could not be allocated, or another result code if there was another error.

The registered root description persists until it is destroyed by calling `mps_root_destroy()`.

### Warning:
The base argument has type `mps_addr_t *` (a typedef for `void **`) but the table of references most likely has some other pointer type, `my_object *` say. It is tempting to write:

```c
mps_root_create_table(..., (mps_addr_t *)my_table, ...)
```

but this is type punning, and its behaviour is not defined in ANSI/ISO Standard C. (GCC and Clang have a warning flag `-Wstrict-aliasing` which detects some errors of this form.) To ensure well-defined behaviour, the pointer must be converted via `void *` (or via `mps_addr_t`, which is a typedef for `void *`), like this:

```c
mps_addr_t base = my_table;
mps_root_create_table(..., base, ...)
```

Register a root that consists of a vector of tagged references.

- `root_o` points to a location that will hold the address of the new root description.
- `arena` is the arena.
- `rank` is the rank of references in the root.
- `rm` is the root mode.
- `base` points to a vector of tagged references.
- `count` is the number of tagged references in the vector.
- `mask` is a bitmask whose set bits specify the location of the tag. References are assumed to have a tag of zero: any value in the vector with a non-zero tag is ignored.

Returns `MPS_RES_OK` if the root was registered successfully, `MPS_RES_MEMORY` if the new root description could not be allocated, or another result code if there was another error.

The registered root description persists until it is destroyed by calling `mps_root_destroy()`.

For example:

```c
#define TAG_MASK 0x3 /* bottom two bits */
/* Global symbol table. */
```
```c
size_t symtab_size;
struct {
    obj_t symbol;
    obj_t value;
} *symtab;

mps_res_t res;
mps_root_t root;
mps_addr_t base = symtab;
res = mps_root_create_table_masked(&root, arena,
    mps_rank_exact(),
    (mps_rm_t)0,
    base, symtab_size * 2,
    (mps_word_t)TAG_MASK);
if (res != MPS_RES_OK) error("can't create symtab root");
```

**Warning:** See the warning for `mps_root_create_table()` above.

```c
void mps_root_destroy (mps_root_t root)
{
    Deregister a root and destroy its description.
    root is the root.
}
```

### 2.10.7 Root introspection

```c
void mps_arena_roots_walk (mps_arena_t arena, mps_roots_stepper_t f, void *p, size_t s)
```

**Deprecated**

starting with version 1.111.

If you think you need this, there’s probably a better way to achieve what you’re trying to do. *Contact us.*

Visit references in registered roots in an arena.

arena is the arena whose roots you want to visit.

f is a function that will be called for each reference to an object in an automatically managed pool class that was found in a registered root belonging to the arena. It takes four arguments: ref is the address of a reference to an object in the arena, root is the root in which ref was found, and p and s are the corresponding arguments that were passed to `mps_arena_roots_walk()`.

p and s are arguments that will be passed to f each time it is called. This is intended to make it easy to pass, for example, an array and its size as parameters.

This function may only be called when the arena is in the parked state.

**See also:**

* Arenas.

**Note**

If a root is ambiguous then the reference might not be to the start of an object; the client program should handle this case. There is no guarantee that the reference corresponds to the actual location that holds the pointer to the object (since this might be a register, for example), but the actual location will be passed if possible. This may aid analysis of roots via a debugger.
void (*mps_roots_stepper_t)(mps_addr_t *ref, mps_root_t root, void *p, size_t s)

The type of a root stepper function.

A function of this type can be passed to mps_arena_roots_walk(), in which case it will be called for each reference into the arena from a root registered with the arena. It receives four arguments:

ref points to a reference in a root. The reference points to something in the arena. If the root is exact then the reference points to the start of an allocated block, but if the root is ambiguous it might point to somewhere in the middle of an allocated block.

root is the description of the root which contains ref.

p and s are the corresponding values that were passed to mps_arena_roots_walk().

2.11 Garbage collection

The arena contains a garbage collector that coordinates the collection of garbage in all of its automatically managed pools. The collector efficiently traces references between roots and pools, and between objects in different pools. It is capable of collecting many automatically managed pools simultaneously.

2.11.1 Generation chains

A generation chain describes a sequence of generations used by a set of automatically managed pools.

A generation is a set of blocks that are managed together by the garbage collector: they are condemned together, and in moving pools they are moved together. If two pools allocate blocks that are expected to live and die together, then it is efficient for them to share a chain.

Typically blocks are allocated in the first generation in the chain, the nursery generation (though you can change this using the MPS_KEY_GEN keyword argument to mps_pool_create_k()), and each time a block survives one collection then it is promoted to the next generation. Thus a generation contains a set of blocks of similar ages.

By default, all pools in an arena share the same generation chain (“the arena’s default generation chain”), but if this doesn’t meet your requirements, then when creating an automatically managed pool, you can choose which chain it should use by passing the MPS_KEY_CHAIN keyword argument to mps_pool_create_k().

Create a generation chain by preparing an array of mps_gen_param_s structures giving the capacity (in kilobytes) and predicted mortality (between 0 and 1) of each generation, and passing them to mps_chain_create().

When the new size of a generation exceeds its capacity, the MPS will be prepared to start collecting the chain to which the generation belongs. See Scheduling of collections below.

For example:

```c
mps_gen_param_s gen_params[] = {
    {1024, 0.8},
    {2048, 0.4},
};

mps_chain_t chain;
mps_res_t res;
res = mps_chain_create(&chain, arena,
                       sizeof(gen_params) / sizeof(gen_params[0]),
                       gen_params);
if (res != MPS_RES_OK) error("Couldn't create chain");
```
mps_chain_t
The type of generation chains. A generation chain describes the structure of generations in a set of pools.

mps_gen_param_s
The type of the structure used to specify a generation in a generation chain.

typedef struct mps_gen_param_s {
    size_t mps_capacity;
    double mps_mortality;
} mps_gen_param_s;

mps_capacity is the capacity of the generation, in kilobytes. When the size of the generation exceeds this, the MPS will be prepared to start collecting it.

mps_mortality is the predicted mortality of the generation: the proportion (between 0 and 1) of blocks in the generation that are expected to be dead when the generation is collected.

These numbers are hints to the MPS that it may use to make decisions about when and what to collect: nothing will go wrong (other than suboptimal performance) if you make poor choices. See Scheduling of collections.

mps_res_t mps_chain_create(mps_chain_t *chain_o, mps_arena_t arena, size_t gen_count, mps_gen_param_s *gen_params)
Create a generation chain.

chain_o points to a location that will hold a pointer to the new generation chain.
arena is the arena to which the generation chain will belong.
gen_count is the number of generations in the chain.
gen_params points to an array describing the generations.

Returns MPS_RES_OK if the generation chain is created successfully, or another result code if it fails.

The generation chain persists until it is destroyed by calling mps_chain_destroy().

void mps_chain_destroy(mps_chain_t chain)
Destroy a generation chain.

chain is the generation chain.

It is an error to destroy a generation chain if there is a garbage collection in progress on the chain, or if there are any pools using the chain. Before calling this function, the arena should be parked (by calling mps_arena_park()) to ensure that there are no collections in progress, and pools using the chain must be destroyed.

2.11.2 Scheduling of collections

Note
It’s likely that the algorithm the MPS uses to schedule its collections will change in future releases. There’s a lot of room for improvement here.

The new size of a generation is the total size of the newly allocated (in generation 0) or newly promoted (in other generations) blocks in that generation. These are the blocks that have not been condemned since they were allocated or promoted into this generation. In pools like AMC (Automatic Mostly-Copying) where the survivors get promoted to the next generation in the chain, the new size of each generation (other than the topmost) is the same as its total size, but in pools like AMS (Automatic Mark and Sweep) where survivors do not get promoted, the two sizes can be different.
When a generation’s new size exceeds its capacity, the MPS considers collecting the chain to which the generation belongs. (How long it takes to get around to it depends on which other collections are in progress.)

**Note**

You can affect the decision as to when to collect the chain by using the *ramp allocation pattern*.

If the MPS decides to collect a chain, all generations are collected up to, and including, the highest generation whose new size exceeds its capacity.

In pools such as **AMC (Automatic Mostly-Copying)**, blocks in generation $g$ that survive collection get promoted to generation $g+1$. If the last generation in the chain is collected, the survivors are promoted into an arena-wide “top” generation.

The predicted mortality is used to estimate how long the collection will take, and this is used in turn to decide how much work the collector will do each time it has an opportunity to do some work. The constraints here are:

1. The *client program* might have specified a limit on the acceptable length of the pause if the work is being done inside `mps_arena_step()`.
2. The collector needs to keep up with the *client program*: that is, it has to collect garbage at least as fast as the client is producing it, otherwise the amount of garbage will grow without bound.

With perfect prediction, the collector’s work should be smoothly distributed, with a small maximum pause time. Getting the predicted mortality wrong leads to “lumpy” distribution of collection work with a longer maximum pause time. If the predicted mortality is too high, the collector will start out by taking small time slices and then find that it has to catch up later by taking larger time slices. If the predicted mortality is too low, the collector will take larger time slices up front and then find that it is idle later on.

### 2.11.3 Garbage collection start messages

```c
mps_message_type_t mps_message_type_gc_start (void)
```

Return the message type of garbage collection start messages.

Garbage collection start messages contain information about why the *garbage collection* started.

The access method specific to a *message* of this message type is:

```c
*mps_message_gc_start_why (mps_arena_t arena, mps_message_t message)
```

Return a string that describes why the *garbage collection* that posted a *message* started.

**See also:**

*Messages.*

```c
const char *mps_message_gc_start_why (mps_arena_t arena, mps_message_t message)
```

Return a string that describes why the *garbage collection* that posted a *message* started.

**See also:**

*Messages.*
2.11.4 Garbage collection messages

mps_message_type_t mps_message_type_gc (void)
Return the message type of garbage collection statistic messages.

Garbage collection statistic messages are used by the MPS to give the client program information about a garbage collection that has taken place. Such information may be useful in analysing the client program’s memory usage over time.

The access methods specific to a message of this type are:

• mps_message_gc_live_size() returns the total size of the condemned set that survived the garbage collection that generated the message;

• mps_message_gc_condemned_size() returns the approximate size of condemned set in the garbage collection that generated the message;

• mps_message_gc_not_condemned_size() returns the approximate size of the set of blocks that were in collected pools, but were not condemned in the garbage collection that generated the message.

See also:
Messages.

size_t mps_message_gc_condemned_size (mps_arena_t arena, mps_message_t message)
Return the “condemned size” property of a message.

arena is the arena which posted the message.

message is a message retrieved by mps_message_get() and not yet discarded. It must be a garbage collection message: see mps_message_type_gc().

The “condemned size” property is the approximate size of the condemned set in the garbage collection that generated the message.

See also:
Messages.

size_t mps_message_gc_live_size (mps_arena_t arena, mps_message_t message)
Return the “live size” property of a message.

arena is the arena which posted the message.

message is a message retrieved by mps_message_get() and not yet discarded. It must be a garbage collection message: see mps_message_type_gc().

The “live size” property is the total size of the set of blocks that survived the garbage collection that generated the message.

See also:
Messages.

size_t mps_message_gc_not_condemned_size (mps_arena_t arena, mps_message_t message)
Return the “not condemned size” property of a message.

arena is the arena which posted the message.

message is a message retrieved by mps_message_get() and not yet discarded. It must be a garbage collection message: see mps_message_type_gc().

The “not condemned size” property is the approximate size of the set of blocks that were in collected pools, but were not in the condemned set in the garbage collection that generated the message.

See also:
2.12 Messages

The MPS sometimes needs to communicate with the client program about events which occur asynchronously, and so information cannot be returned as function call results. Messages are the mechanism for this asynchronous communication, implemented in the form of a message queue attached to each arena.

The client program must enable each message type that it is prepared to handle, by calling `mps_message_type_enable()`. Then it must poll the message queue at regular intervals when it is convenient to do so, calling `mps_message_get()` to retrieve each message from the queue, and finally calling `mps_message_discard()` when done with the message.

Messages are thus manually managed: if the client program enables one or more message types, and then neglects to poll the message queue or neglects to discard the messages it retrieved, then messages will leak.

There is no requirement on the client program to retrieve and discard messages promptly. However, a client program that allows the number of garbage collection (or garbage collection start) messages on the message queue to grow without limit will eventually find that new garbage collections no longer start until some of these messages are retrieved and discarded.

2.12.1 Finalization messages

Finalization is implemented by posting a finalization message (of type `mps_message_type_finalization()`) to the arena’s message queue. This allows the client program to perform the finalization at a convenient time and so avoid synchronization difficulties.

The block is not actually reclaimed until the finalization message is removed from the message queue and discarded, by calling `mps_message_get()` followed by `mps_message_discard()`.

See Finalization.

2.12.2 Example: interactive chatter

The toy Scheme interpreter enables garbage collection messages when started in interactive mode:

```c
mps_message_type_enable(arena, mps_message_type_gc());
mps_message_type_enable(arena, mps_message_type_gc_start());
```

Then, after every interactive command finishes, it reads these messages from the message queue and prints a description of the contents of each one:

```c
static void mps_chat(void)
{
    mps_message_type_t type;
    while (mps_message_queue_type(&type, arena)) {
        mps_message_t message;
        mps_bool_t b;
        b = mps_message_get(&message, arena, type);
        assert(b); /* we just checked there was one */
        if (type == mps_message_type_gc_start()) {
```
Here's how this looks in operation:

MPS Toy Scheme Example
9960, 0> (define (make-list n e) (if (eqv? n 0) '() (cons e (make-list (- n 1) e))))
make-list
10824, 0> (length (make-list 1000 #t))
1000
Collection started.
   Why: Generation 0 of a chain has reached capacity: start a minor collection.
   Clock: 6649
507408, 1> (length (make-list 200 #f))
200
Collection finished.
   live 112360
   condemned 196600
   not_condemned 0
   clock: 18431
607192, 1> Bye.

Note
This kind of interactive “chatter” may be useful when testing and debugging memory management, but should not be used otherwise. The scheduling of garbage collections is not normally of interest even to programmers, and chatter of this sort may give the illusion that a program is spending much more time garbage collecting than is actually the case.

Versions of GNU Emacs prior to 19.31 (May 1996) used to display the message “Garbage collecting...” during a collection. Erik Naggum commented on this feature:

I have run some tests at the U of Oslo with about 100 users who generally agreed that Emacs had become faster in the latest Emacs pretest. All I had done was to remove the “Garbage collecting” message which people perceive as slowing Emacs down and tell them that it had been sped up.

2.12.3 Message types

mps_message_type_t
The type of message types.

There are three message types:

1. mps_message_type_finalization()
2. mps_message_type_gc()
3. mps_message_type_gc_start()
void mps_message_type_disable (mps_arena_t arena, mps_message_type_t message_type)

Restore an arena to the default state whereby messages of the specified message type are not posted, reversing the effect of an earlier call to mps_message_type_enable().

arena is an arena.
message_type is the message type to be disabled.

Any existing messages of the specified type are flushed from the message queue of arena.

---

Note

It is permitted to call this function when message_type is already disabled, in which case it has no effect.

void mps_message_type_enable (mps_arena_t arena, mps_message_type_t message_type)

Enable an arena to post messages of a specified message type.

arena is an arena.
message_type is the message type to be enabled.

This function tells the MPS that arena may post messages of message_type to its message queue. By default, the MPS does not generate any messages of any type.

A client program that enables messages for a message type must access messages by calling mps_message_get() and discard them by calling mps_message_discard(), or the message queue may consume unbounded resources.

The client program may disable the posting of messages by calling mps_message_type_disable().

---

Note

It is permitted to call this function when message_type is already enabled, in which case it has no effect.

---

2.12.4 Message interface

mps_message_t

The type of a message.

Messages are manually managed. They are created at the instigation of the MPS (but see mps_message_type_enable()), and are deleted by the client program by calling mps_message_discard().

An arena has a message queue from which messages can be obtained by calling mps_message_get().

An mps_message_t is a reference into MPS managed memory, and can safely be fixed.

mps_clock_t mps_message_clock (mps_arena_t arena, mps_message_t message)

Returns the time at which the MPS posted a message.

arena is the arena which posted the message.
message is a message retrieved by mps_message_get() and not yet discarded.

If message belongs to one of the following supported message, return the time at which the MPS posted the message:

• mps_message_type_gc;
• mps_message_type_gc_start.
For other message types, the value returned is always zero.

Messages are asynchronous: they are posted by the MPS, wait on a queue, and are later collected by the client program. Each message (of the supported message types) records the time that it was posted, and this is what `mps_message_clock()` returns.

The time returned is the `mps_clock_t` value returned by the `plinth` function `mps_clock()` at the time the message was posted. You can subtract one clock value from another to get the time interval between the posting of two messages.

```c
void mps_message_discard (mps_arena_t arena, mps_message_t message)
```

Indicate to the MPS that the client program has no further use for a message and the MPS can now reclaim any storage associated with the message.

- `arena` is the arena which posted the message.
- `message` is the message. After this call, `message` is invalid and should not be passed as an argument to any message functions.

Messages are essentially manually managed. This function allows the MPS to reclaim storage associated with messages. If the client does not discard messages then the resources used may grow without bound.

As well as consuming resources, messages may have other effects that require them to be tidied by calling this function. In particular finalization messages refer to a finalized block, and prevent the object from being reclaimed (subject to the usual garbage collection liveness analysis). A finalized block cannot be reclaimed until all its finalization messages have been discarded. See `mps_message_type_finalization()`.

**See also:**

Finalization.

```c
mps_message_type_t mps_message_type (mps_arena_t arena, mps_message_t message)
```

Return the message type of a message.

- `arena` is the arena that posted the message.
- `message` is a message retrieved by `mps_message_get()` and not yet discarded.

### 2.12.5 Message queue interface

```c
mps_bool_t mps_message_get (mps_message_t *message_o, mps_arena_t arena, mps_message_type_t message_type)
```

Get a message of a specified type from the message queue for an arena.

- `message_o` points to a location that will hold the address of the message if the function succeeds.
- `arena` is the arena.
- `message_type` is the type of message to return.

If there is at least one message of the specified type on the message queue of the specified arena, then this function removes one such message from the queue, stores a pointer to the message in the location pointed to by `message_o`, and returns true. Otherwise it returns false.

```c
mps_bool_t mps_message_poll (mps_arena_t arena)
```

Determine whether there are currently any messages on a message queue for an arena.

- `arena` is the arena whose message queue will be polled.

Returns true if there is at least one message on the message queue for `arena`, or false if the message queue is empty.

**Note**
If you are interested in a particular type of message, it is usually simpler to call `mps_message_get()`.

```c
def mps_bool_t mps_message_queue_type(mps_message_type_t *message_type_o, mps_arena_t arena)
    Determine whether there are currently any messages on a message queue for an arena, and return the message type of the first message, if any.

    message_type_o points to a location that will hold the message type of the first message on the queue, if any.

    arena is the arena whose message queue will be polled.

    If there is at least one message on the message queue of arena, then this function returns true, and also writes the message type of the first message on the queue into the location pointed to by message_type_o. If there are no messages on the message queue, it returns false.
```

Note

If you are interested in a particular type of message, it is usually simpler to call `mps_message_get()`.

## 2.13 Finalization

It is sometimes necessary to perform actions when a block of memory dies. For example, a block may represent the acquisition of an external resource such as a file handle or a network connection. When the block dies, the corresponding resource must be released. This procedure is known as finalization.

A block requiring finalization must be registered by calling `mps_finalize()`:

```c
mps_addr_t ref = block_requiring_finalization;
mps_finalize(arena, &ref);
```

A block that been registered for finalization becomes finalizable as soon as the garbage collector observes that it would otherwise be reclaimed (that is, the only thing keeping it alive is the fact that it needs to be finalized). If a block is finalizable the MPS may choose to finalize it (by posting a finalization message: see below) at any future time.

Note

This means that a block that was determined to be finalizable, but then became unconditionally live by the creation of a new strong reference to it, may still be finalized.

Weak references do not prevent blocks from being finalized. At the point that a block is finalized, weak references will still validly refer to the block. The fact that a block is registered for finalization prevents weak references to that block from being splatted. See Weak references.

The Memory Pool System finalizes a block by posting a finalization message to the message queue of the arena in which the block was allocated.

Note

This design avoids the problems that can result from the garbage collector calling a function in the client program to do the finalization. In such an implementation, the client program’s finalization code may end up running concurrently with other code that accesses the underlying resource, and so access to the resource needs to be guarded with a lock, but then an unlucky scheduling of finalization can result in deadlock. See Boehm (2002) for a detailed discussion of this issue.
The message type of finalization messages is \texttt{mps\_message\_type\_finalization()}, and the client program must enable the posting of these messages by calling \texttt{mps\_message\_type\_enable()} before any block becomes finalizable:

\begin{verbatim}
mps_message_type_enable(arena, mps_message_type_finalization());
\end{verbatim}

When a finalization message has been retrieved from the message queue by calling \texttt{mps\_message\_get()}, the finalization reference may be accessed by calling \texttt{mps\_message\_finalization\_ref()}. The finalization message keeps the block alive until it is discarded by calling \texttt{mps\_message\_discard()}.

\begin{verbatim}
Note
The client program may choose to keep the finalized block alive by keeping a strong reference to the finalized object after discarding the finalization message.

This process is known as resurrection and in some finalization systems requires special handling, but in the MPS this just is just the usual result of the rule that strong references keep objects alive.

It is fine to re-register a block for finalization after retrieving its finalization message from the message queue. This will cause it to be finalized again should all strong references disappear again.

\end{verbatim}

\begin{verbatim}
Note
Calling \texttt{mps\_message\_discard()} does not reclaim the space occupied by the finalized block (that happens at the next collection, if the block is found to be dead at that point), and so the block must remain validly formatted (scannable, skippable, and so on). It might make sense to replace it with a padding object.

See Messages for details of the message mechanism.

2.13.1 Multiple finalizations

A block may be registered for finalization multiple times. A block that has been registered for finalization \textit{n} times will be finalized at most \textit{n} times.

This may mean that there are multiple finalization messages on the queue at the same time, or it may not (it may be necessary for the client program to discard previous finalization messages for a block before a new finalization messages for that block are posted to the message queue). The MPS provides no guarantees either way: a client program that registers the same block multiple times must cope with either behaviour.

2.13.2 Cautions

1. Don’t rely on finalization for your program to work. Treat it as an optimization that enables the freeing of resources that the garbage collector can prove are unreachable.

2. The MPS provides no guarantees about the promptness of finalization. The MPS does not finalize a block until it determines that the block is finalizable, which may require a full garbage collection in the worst case, and such a collection may not be scheduled for some time. Or the block may never become finalizable because it is incorrectly determined to be reachable due to an ambiguous reference pointing to it. Or the block may never become finalizable because it remains reachable through a reference, even if that reference might never be used.

3. Even when blocks are finalized in a reasonably timely fashion, the client needs to process the finalization messages in time to avoid the resource running out. For example, in the Scheme interpreter, finalization messages are only processed at the end of the read–eval–print loop, so a program that opens many files may run out of handles even though the associated objects are all finalizable, as shown here:
A less naïve interpreter might process finalization messages on a more regular schedule, or might take emergency action in the event of running out of open file handles by carrying out a full garbage collection and processing any finalization messages that are posted as a result.

If you are designing a programming language then it is generally a good idea to provide the programmer with a mechanism for ensuring prompt release of scarce resources. For example, Scheme provides the \[\text{(with-input-from-file)}\] procedure which specifies that the created port has dynamic extent (and so can be closed as soon as the procedure exits, even if it is still reachable).

4. The MPS does not finalize objects in the context of \[\text{mps_arena_destroy()}\] or \[\text{mps_pool_destroy()}\]. Moreover, if you have pools containing objects registered for finalization, you must destroy these pools by following the “safe tear-down” procedure described under \[\text{mps_pool_destroy()}\].

**Note**
Under normal circumstances, finalization code can assume that objects referenced by the object being finalized (“object F”) have themselves not yet been finalized. (Because object F is keeping them alive.) If finalization code is run at program exit, this assumption is no longer true. It is much more difficult to write correct code if it has to run under both circumstances.

This is why Java’s \[\text{System.runFinalizersOnExit}\] is deprecated. See Appendix A of \[\text{Boehm (2002)}\] for a discussion of this problem.

**Note**
The only reliable way to ensure that all finalizable objects are finalized is to maintain a table of weak references (1) to all such objects. The weak references don’t prevent the objects from being finalized, but you can iterate over the list at an appropriate point and finalize any remaining objects yourself.

5. Not all \[\text{pool classes}\] support finalization. In general, only pools that manage objects whose liveness is determined by garbage collection do so. See the \[\text{Pool reference}\].

### 2.13.3 Finalization interface

\[\text{mps_finalize(mps_arena_t arena, mps_addr_t *ref_p)}\]

Register a block for finalization.

- \[\text{arena}\] is the arena in which the block lives.
- \[\text{ref_p}\] points to a reference to the block to be registered for finalization.

Returns \[\text{MPS_RES_OK}\] if successful, or another \[\text{result code}\] if not.

This function registers the block pointed to by \[\text{*ref_p}\] for finalization. This block must have been allocated from a pool in \[\text{arena}\]. Violations of this constraint may not be checked by the MPS, and may be unsafe, causing the MPS to crash in undefined ways.

**Note**
This function receives a pointer to a reference. This is to avoid placing the restriction on the \[\text{client program}\] that the C call stack be a root.
mps_res_t mps_definalize (mps_arena_t arena, mps_addr_t *ref_p)
Deregister a block for finalization.

arena is the arena in which the block lives.
ref_p points to a reference to the block to be deregistered for finalization.

Returns MPS_RES_OK if successful, or MPS_RES_FAIL if the block was not previously registered for finalization.

Note
This function receives a pointer to a reference. This is to avoid placing the restriction on the client program that the C call stack be a root.

2.13.4 Finalization messages

mps_message_type_t mps_message_type_finalization (void)
Return the message type of finalization messages.

Finalization messages are used by the MPS to implement finalization. When the MPS detects that a block that has been registered for finalization (by calling mps_finalize()) is finalizable, it finalizes it by posting a message of this type.

Note that there might be delays between the block becoming finalizable, the MPS detecting that, and the message being posted.

In addition to the usual methods applicable to messages, finalization messages support the mps_message_finalization_ref() method which returns a reference to the block that was registered for finalization.

See also:
Messages.

void mps_message_finalization_ref (mps_addr_t *ref_o, mps_arena_t arena, mps_message_t message)
Returns the finalization reference for a finalization message.
ref_o points to a location that will hold the finalization reference.
arena is the arena which posted the message.
message is a message retrieved by mps_message_get() and not yet discarded. It must be a finalization message: see mps_message_type_finalization().

The reference returned by this method is a reference to the block that was originally registered for finalization by a call to mps_finalize().

Note
The reference returned is subject to the normal constraints, such as might be imposed by a moving collection, if appropriate. For this reason, it is stored into the location pointed to by ref_o in order to enable the client program to place it directly into scanned memory, without imposing the restriction that the C stack be a root.

The message itself is not affected by invoking this method. Until the client program calls mps_message_discard() to discard the message, it will refer to the object and prevent its reclamation.

See also:
Messages.
2.14 Location dependency

Location dependencies provide a means by which the client program can depend on the location of blocks (that is, on the bits in pointers to the blocks) in the presence of a moving memory manager (where the location of blocks may change and the client program needs to recognize and correctly deal with such cases).

The interface is intended to support (amongst other things) address-based hash tables and that will be used as a running example. See the section Location dependency in the Guide for a more detailed look at this example.

2.14.1 Terminology

A location dependency is represented by a structure of type mps_ld_s. It encapsulates a set of dependencies on the locations of blocks. It can be used to determine whether any of the blocks have been moved by the memory manager.

To depend on the location of a block is to perform a computation whose result depends on the particular representation (that is, the “bit-pattern”) of a reference to the block. This includes any sort of hash operation on a pointer to the block (such as treating the pointer as an integer and taking it modulo 257). It is possible to depend on the location of more than one block.

A dependency has been made stale if the block whose location was depended on might have moved since the dependency was made. If this is the case, then computations that depend on the location of a block may give different results. A location dependency has been made stale if any of the blocks whose location has been depended on might have moved since the respective dependency was made.

2.14.2 Creating dependencies

The client program must provide space for the mps_ld_s structure. Typically, this will be inlined in some larger structure. This structure can be in memory managed by the MPS or elsewhere; that doesn’t matter.

For example, the toy Scheme interpreter inlines the location dependency in its hash table structure:

```c
typedef struct table_s {
    type_t type;       /* TYPE_TABLE */
    hash_t hash;       /* hash function */
    cmp_t cmp;         /* comparison function */
    mps_ld_s ld;       /* location dependency */
    obj_t buckets;     /* hash buckets */
} table_s;
```

Before the first use, the location dependency must be reset by calling the function mps_ld_reset().

Note

This means that it is not possible to statically create a location dependency that has been reset.

You can call mps_ld_reset() at any later point to clear all dependencies from the structure. For example, this is normally done whenever mps_ld_isstale() returns true.

2.14.3 Adding dependencies

Before the location of a block is depended on (for example, hashed) a reference to the block may be added to a location dependency by calling mps_ld_add(). Dependencies on many blocks can be added to the same location dependency.
It is also possible to merge two location dependencies by calling `mps_ld_merge()`, which has the same effect as adding all of the references from one dependency to another.

For example, in an address-based hash table implementation, each key that is added to the table must be added to the dependency before its address is hashed. In the toy Scheme interpreter this is most easily done in the function that hashes an address:

```c
static unsigned long eq_hash(obj_t obj, mps_ld_t ld)
{
    union {char s[sizeof(obj_t)]; obj_t addr;} u;
    if (ld) mps_ld_add(ld, arena, obj);
    u.addr = obj;
    return hash(u.s, sizeof(obj_t));
}
```

### 2.14.4 Testing dependencies for staleness

When the location of a block is used, first carry out the computation in the normal way. For example, when looking up a key in an address-based hash table, start by hashing the pointer and looking up the corresponding index in the table.

If this succeeds (for example, the key was found in the table at the place indicated by the hash of its address), then no further test is required: the operation can proceed as usual.

But if the operation fails, you might be in one of two cases:

1. the location of the block has not been depended on before (for example, the key has never been added to the hash table);
2. the location of the block has been depended on before (for example, the key was added to the hash table), but the block has moved and the dependency has become stale.

At this point you should call `mps_ld_isstale()`. If it returns false, then you know that the block has not moved, so you must be in case (1).

But if `mps_ld_isstale()` returns true, you could still be in either case (1) or case (2). All `mps_ld_isstale()` tells you is that the block might have moved, not whether the block has moved. At this point you must:

1. reset the location dependency;
2. repeat the computation in some way that doesn’t depend on the old locations of all the blocks that were added to that dependency; and
3. re-add a dependency on each block.

For example, in the case of a hash table you should rehash based on the new locations of the blocks.

In the toy Scheme interpreter this behaviour is encapsulated into `table_find`:

```c
static struct bucket_s *table_find(obj_t tbl, obj_t buckets, obj_t key, int add)
{
    struct bucket_s *b;
    assert(TYPE(tbl) == TYPE_TABLE);
    b = buckets_find(tbl, tbl->table.buckets, key, add);
    if ((b == NULL || b->key == NULL || b->key == obj_deleted)
        && mps_ld_isstale(tbl->table.ld, arena, key))
    {
        b = table_rehash(tbl, tbl->table.buckets->buckets.length, key);
    }
    return b;
}
```
After `mps_ld_isstale()` has returned true, and you’ve rehashed the table, it might be tempting to repeat the usual address-based lookup. But the MPS does not guarantee that `mps_ld_isstale()` will not return true again: if the re-hashing took a long time or touched lots of memory, there might have been another garbage collection. (The only time that `mps_ld_isstale()` guarantees to return false is immediately after `mps_ld_reset()`.)

You might put in a loop here, but for reliability it is better to fall back to a non-address-based version of the computation: here, since `table_rehash` has to loop over all the entries in the table anyway, it might as well find the bucket containing key at the same time and return it.

**Warning:** Don’t forget to check for staleness when setting a key in a table. If the key is stale then it would be a mistake to add it to the table as then the key will be present twice, at the positions given by the hash of its old and new addresses, thus violating the invariant of the hash table (that a key appears at most once). Similarly, staleness must be tested when deleting a key from a table.

### 2.14.5 Thread safety

The functions are all thread-safe with respect to operations on different location dependencies. That means that it is not necessary for threads to interlock if they are performing operations on different location dependencies. The descriptions of the individual functions detail their thread-safety attributes if multiple threads need to access the same location dependency.

### 2.14.6 Location dependency interface

**mps ld t**

The type of location dependencies. It is a transparent alias for a pointer to `mps ld s`.

A location dependency records the fact that the client program depends on the bit patterns of some references (and not merely on the identity of the block to which the reference refers), and provides a function (`mps ld_isstale()`) to find out whether any of these references have been changed because a block has been moved.

A typical use is in the implementation of a hash table which hashes blocks by hashing their addresses. After a block has moved, the table needs to be rehashed, otherwise it will not be found in the table.

**mps ld s**

The type of the structure used to represent a location dependency.

```c
typedef struct mps ld s {
    mps_word_t w0, w1;
} mps ld s;
```

It is an opaque structure type: it is supplied so that the client program can inline the structure (because its size is known), but the client must not access it other than via the functions `mps ld_add()`, `mps ld_isstale()`, `mps ld_merge()`, and `mps ld_reset()`.

**void mps ld add (mps ld t ld, mps arena t arena, mps addr t addr)**

Add a dependency on a block to a location dependency.

`ld` is a location dependency.

`arena` is the arena to which `addr` belongs.

`addr` is the address of the block.

After calling `mps ld_add()`, and until `ld` is passed to `mps ld_reset()`, the call
mps_ld_isstale(ld, arena, addr)

will return true if the block has moved.

Note

It is an error to call mps_ld_add() on the same location dependency with addresses from two different arenas. If you need to test for staleness against multiple arenas, then you need at least one location dependency for each arena.

mps_ld_add() is not thread-safe with respect to mps_ld_add(), mps_ld_merge(), or mps_ld_reset() on the same location dependency, but it is thread-safe with respect to mps_ld_isstale() operations. This means that calls to mps_ld_add() from different threads must interlock if they are using the same location dependency. The practical upshot of this is that there should be a lock associated with each location dependency.

mps_bool_t mps_ld_isstale(mps_ld_t ld, mps_arena_t arena, mps_addr_t addr)

Determine if a dependency on the location of a block in a location dependency might be stale with respect to an arena.

ld is the location dependency.

arena is the arena to test for staleness against. It must be the same arena that was passed to all calls to mps_ld_add() on ld.

addr is the address of the block that is to be tested for staleness.

If there have been no calls to mps_ld_add() on ld since the last call to mps_ld_reset(), then return false.

If the block at addr was formerly added to the location dependency ld and subsequently moved by arena, then return true.

Otherwise, mps_ld_isstale() may return either true or false. (The function strives to return true in the case where addr was added to the location dependency and subsequently moved, and false otherwise, but cannot ensure this.)

Note

mps_ld_isstale() may report a false positive: it may return true in the case where addr was not added to the location dependency, or in the case where it was added but not moved. It never reports a false negative.

mps_ld_isstale() is thread-safe with respect to itself and with respect to mps_ld_add(), but not with respect to mps_ld_reset().

mps_bool_t mps_ld_isstale_any(mps_ld_t ld, mps_arena_t arena)

Determine if a dependency on the location of any block in a location dependency might be stale with respect to an arena.

ld is the location dependency.

arena is the arena to test for staleness against. It must be the same arena that was passed to all calls to mps_ld_add() on ld.

If there have been no calls to mps_ld_add() on ld since the last call to mps_ld_reset(), then return false.

If any block added to the location dependency ld has been moved by arena, then return true.

2.14. Location dependency
Otherwise, `mps_ld_isstale_any()` may return either true or false. (The function strives to return true in the case where a block was added to the location dependency and subsequently moved, and false otherwise, but cannot ensure this.)

**Note**

`mps_ld_isstale_any()` has the same thread-safety properties as `mps_ld_isstale()`.

```c
void mps_ld_merge(mps_ld_t dest_ld, mps_arena_t arena, mps_ld_t src_ld)
merge one location dependency into another.
dest_ld is the destination of the merge.
arena is the arena.
src_ld is the source of the merge.
The effect of this is to add all the addresses that were added to src_ld to the dest_ld.

**Note**

`mps_ld_merge()` has the same thread-safety properties as `mps_ld_add()`.
```

```c
void mps_ld_reset(mps_ld_t ld, mps_arena_t arena)
reset a location dependency.
ld is the location dependency.
arena is an arena.
After this call, ld encapsulates no dependencies. After the call to mps_ld_reset() and prior to any call to mps_ld_add() on ld, mps_ld_isstale() on ld will return false for all arenas.

**Note**

`mps_ld_reset()` is not thread-safe with respect to any other location dependency function.
```

### 2.15 Segregated allocation caches

A segregated allocation cache is a data structure that can be attached to any manually managed pool, that maintains a segregated free list, that is, a reserve of free blocks segregated by size.

Create a segregated allocation cache by preparing an array of structures of type `mps_sac_class_s` and passing them to `mps_sac_create()`. The values in these structures are hints as to the size of the blocks, the number of blocks of each size, and the relative frequency of allocations and deallocations at that size.

For example, suppose we have a pool where we expect to allocate a small number of relatively long-lived 128-byte objects, and a large number of relatively short-lived 8-byte objects, we might create a cache as follows:

```c
mps_sac_class_s classes[3] = {{8, 100, 10}, {128, 8, 1}};
mps_sac_t sac;
res = mps_sac_create(&sac, pool, sizeof classes / sizeof classes[0], classes);
if (res != MPS_RES_OK)
    error("failed to create allocation cache");
```
Allocations through the cache (using `mps_sac_alloc()` or `MPS_SAC_ALLOC_FAST()`) are serviced from the cache if possible, otherwise from the pool. Similarly, deallocations through the cache (using `mps_sac_free()` or `MPS_SAC_FREE_FAST()`) return the block to the appropriate free list for its size. For example:

```c
Foo *foo;
mps_addr_t p;
mps_res_t res;

res = mps_sac_alloc(&p, sac, sizeof *foo, false);
if (res != MPS_RES_OK)
    error("failed to alloc foo");
foo = p;
/* use 'foo' */

mps_sac_free(sac, p, sizeof *foo);
```

The macros `MPS_SAC_ALLOC_FAST()` and `MPS_SAC_FREE_FAST()` allow allocation and deallocation to be inlined in the calling functions, in the case where a free block is found in the cache.

**Note**

It is recommended that you deallocate a block via the same segregated allocation cache that you allocated it from. However, the system is more general than that, and in fact a block that was allocated from cache A can be deallocated via cache B, provided that:

1. the two caches are attached to the same pool; and
2. the two caches have the same class structure, that is, they were created by passing identical arrays of size classes.

**Warning:** Segregated allocation caches work poorly with debugging pool classes: the debugging checks only happen when blocks are moved between the cache and the pool.

### 2.15.1 Cache interface

**mps_sac_t**

The type of segregated allocation caches.

**MPS_SAC_CLASS_LIMIT**

The number of size classes that `mps_sac_create()` is guaranteed to accept.

More might be accepted: in fact, there might not be any limit in the implementation on the maximum number of size classes, but if you specify more than this many, you should be prepared to handle the result code `MPS_RES_LIMIT`.

**mps_sac_class_s**

The type of the structure describing a size class in a segregated allocation cache.

```c
typedef struct mps_sac_class_s {
    size_t mps_block_size;
    size_t mps_cached_count;
    unsigned mps_frequency;
} mps_sac_class_s;
```

An array of these structures must be passed to `mps_sac_create()` when creating a segregated allocation cache.

### 2.15. Segregated allocation caches
mps_block_size is the maximum size of any block in this size class. It must be a multiple of the alignment of the pool to which the cache belongs.

mps_cached_count is the number of blocks of this size class to cache. It is advice to the MPS on how many blocks to cache, not an absolute limit. The cache policy tries to accommodate fluctuations in the population and minimize the cost of responding to client requests; the purpose of this parameter is to limit how much memory the client program is willing to set aside for this purpose. However, a cached_count of zero prevents any caching of blocks falling into that size class.

mps_frequency is a number that describes the frequency of requests (allocation and deallocation combined) in this size class relative to the other size classes in the cache.

```c
mps_sac_create(mps_sac_t *sac_o, mps_pool_t pool, size_t classes_count, mps_sac_class_s *classes)
```

Create a segregated allocation cache for a pool.

sac_o points to a location that will hold the address of the segregated allocation cache.

pool is the pool the cache is attached to.

classes_count is the number of size classes in the cache.

classes points to an array describing the size classes in the cache.

Returns MPS_RES_OK if the segregated allocation cache is created successfully. Returns MPS_RES_MEMORY or MPS_RES_COMMIT_LIMIT when it fails to allocate memory for the internal cache structure. Returns MPS_RES_LIMIT if you ask for too many size classes: in this case, combine some small adjacent classes.

After this function returns, the array of size classes pointed to be classes is no longer needed and may be discarded. The segregated allocation cache pointed to by sac_o persists until it is destroyed by calling mps_sac_destroy().

This function creates an allocation cache whose free list is segregated into the given size classes. The cache can get more memory from the given pool, or return memory to it.

Segregated allocation caches can be associated with any pool that supports manual allocation with the functions mps_alloc() and mps_free().

The size classes are described by an array of element type mps_sac_class_s. This array is used to initialize the segregated allocation cache, and is not needed after mps_sac_create() returns. The following constraints apply to the array:

- You must specify at least one size class.
- All size classes must have different sizes.
- The size classes must be given in the order of increasing size.
- The smallest size must be at least as large as sizeof(void *).
- Each size must be a multiple of the alignment of the pool.
- There might be a limit on how many classes can be described, but it will be at least MPS_SAC_CLASS_LIMIT.

The MPS automatically provides an “overlarge” size class for arbitrarily large allocations above the largest size class described. Allocations falling into the overlarge size class are not cached.

Any allocations whose size falls between two size classes are allocated from the larger size class.

Note

Too many size classes will slow down allocation; too few size classes waste more space in internal fragmentation. It is assumed that overlarge allocations are rare; otherwise, you would add another size class for them, or even create separate allocation caches or pools for them.
void mps_sac_destroy(mps_sac_t sac)

Destroy a segregated allocation cache.

sac is the segregated allocation cache to destroy.

Returns all memory in the cache to the associated pool. The pool might then return some memory to the arena, but that’s up to the pool’s usual policy.

Destroying the cache has no effect on blocks allocated through it.

void mps_sac_flush(mps_sac_t sac)

Flush a segregated allocation cache, returning all memory held in it to the associated pool.

sac is the segregated allocation cache to flush.

This is something that you’d typically do when you know you won’t be using the segregated allocation cache for awhile, but want to hold on to the cache itself. Destroying a cache has the effect of flushing it.

Flushing the segregated allocation cache might well cause the pool to return some memory to the arena, but that’s up to the pool’s usual policy.

Note

The MPS might also decide to take memory from the segregated allocation cache without the client program requesting a flush.

Note

The client program is responsible for synchronizing the access to the cache, but if the cache decides to access the pool, the MPS will properly synchronize with any other threads that might be accessing the same pool.

2.15.2 Allocation interface

mps_res_t mps_sac_alloc(mps_addr_t *p_o, mps_sac_t sac, size_t size, mps_bool_t has_reservoir_permit)

Allocate a block using a segregated allocation cache. If no suitable block exists in the cache, ask for more memory from the associated pool.

p_o points to a location that will hold the address of the allocated block.

sac is the segregated allocation cache.

size is the size of the block to allocate. It does not have to be one of the size classes of the cache; nor does it have to be aligned.

has_reservoir_permit should be false.

Returns MPS_RES_OK if successful: in this case the address of the allocated block is *p_o. The allocated block can be larger than requested. Blocks not matching any size class are allocated from the next largest class, and blocks larger than the largest size class are simply allocated at the requested size (rounded up to alignment, as usual).

Returns MPS_RES_MEMORY if there wasn’t enough memory, MPS_RES_COMMIT_LIMIT if the commit limit was exceeded, or MPS_RES_RESOURCE if it ran out of virtual memory.

Note

There’s also a macro MPS_SAC_ALLOC_FAST() that does the same thing. The macro is faster, but generates more code and does less checking.
Note
The client program is responsible for synchronizing the access to the cache, but if the cache decides to access the pool, the MPS will properly synchronize with any other threads that might be accessing the same pool.

Note
Blocks allocated through a segregated allocation cache should only be freed through a segregated allocation cache with the same class structure. Calling mps_free() on them can cause memory leaks, because the size of the block might be larger than you think. Naturally, the cache must also be attached to the same pool.

MPS_SAC_ALLOC_FAST (mps_res_t res_v, mps_addr_t *p_v, mps_sac_t sac, size_t size, mps_bool_t has_reservoir_permit)
A macro alternative to mps_sac_alloc(). It is faster than the function, but generates more code, does less checking.

It takes an lvalue p_v which is assigned the address of the allocated block (instead of a pointer to a location to store it). It takes an additional first argument, the lvalue res_v, which is assigned the result code.

Note
MPS_SAC_ALLOC_FAST() may evaluate its arguments multiple times, except for has_reservoir_permit, which it evaluates at most once, and only if it decides to access the pool.

void mps_sac_free (mps_sac_t sac, mps_addr_t p, size_t size)
Free a block using a segregated allocation cache. If the cache would become too full, some blocks may be returned to the associated pool.

sac is the segregated allocation cache.

p points to the block to be freed. This block must have been allocated through a segregated allocation cache with the same class structure, attached to the same pool. (Usually, you’d use the same cache to allocate and deallocate a block, but the MPS is more flexible.)

size is the size of the block. It should be the size that was specified when the block was allocated (the cache knows what the real size of the block is).

Note
The client program is responsible for synchronizing the access to the cache, but if the cache decides to access the pool, the MPS will properly synchronize with any other threads that might be accessing the same pool.

Note
There’s also a macro MPS_SAC_FREE_FAST() that does the same thing. The macro is faster, but generates more code and does no checking.

Note
mps_sac_free() does very little checking: it’s optimized for speed. Double frees and other mistakes will only be detected when the cache is flushed (either by calling mps_sac_flush() or automatically), and may not be detected at all, if intervening operations have obscured symptoms.

MPS_SAC_FREE_FAST (mps_sac_t sac, mps_addr_t p, size_t size)
A macro alternative to mps_sac_free() that is faster than the function but does no checking. The arguments
are identical to the function.

## 2.16 Allocation patterns

An *allocation pattern* is a hint to the MPS to expect a particular pattern of allocation on an *allocation point*. The MPS may use this hint to schedule more effective garbage collection.

There are two allocation patterns, `mps_alloc_pattern_ramp()` and `mps_alloc_pattern_ramp_collect_all()`.

### mps_alloc_pattern_t

The type of *allocation patterns*.

### mps_res_t mps_ap_alloc_pattern_begin(mps_ap_t ap, mps_alloc_pattern_t alloc_pattern)

Start a period of allocation that behaves according to an *allocation pattern*. The period persists until a corresponding call to `mps_ap_alloc_pattern_end()`.

- `ap` is the *allocation point* in which the patterned allocation will occur.
- `alloc_pattern` is the allocation pattern.

Returns `MPS_RES_OK` if the allocation pattern is supported by this allocation point. At present this is always the case, but in future this function may return another *result code* if the allocation pattern is not supported by the allocation point.

**Note**

It is harmless to call `mps_ap_alloc_pattern_begin()` even if it isn’t supported by the allocation point. The pattern is simply ignored in that case.

If `mps_ap_alloc_pattern_begin()` is used multiple times on the same allocation point without intervening calls to `mps_ap_alloc_pattern_end()`, the calls match in a stack-like way, outermost and innermost: that is, allocation patterns may nest, but not otherwise overlap.

Some allocation patterns may additionally support overlap: if so, the documentation for the individual pattern types will specify this.

### mps_res_t mps_ap_alloc_pattern_end(mps_ap_t ap, mps_alloc_pattern_t alloc_pattern)

End a period of allocation on an *allocation point* that behaves according to an *allocation pattern*.

- `ap` is the allocation point in which the patterned allocation occurred.
- `alloc_pattern` is the allocation pattern.

Returns `MPS_RES_OK` if the period of allocation was successfully ended, or `MPS_RES_FAIL` if there was no matching call to `mps_ap Alloc_pattern_begin()`. Calls match in a stack-like way, outermost and innermost: that is, allocation patterns may nest, but not otherwise overlap.

Some allocation patterns may additionally support overlap: if so, the documentation for the individual pattern types will specify this.

### mps_res_t mps_ap_alloc_pattern_reset(mps_ap_t ap)

End all *patterned allocation* on an *allocation point*.

- `ap` is the allocation point on which to end all patterned allocation.

Returns `MPS_RES_OK`. It may fail in future if certain allocation patterns cannot be ended for that allocation point at that point in time.

This function may be used to recover from error conditions.
2.16.1 Ramp allocation

*Ramp allocation* a pattern of allocation whereby the *client program* builds up an increasingly large data structure, the live size of which increases until a particular time, at which time most of the data structure is discarded, resulting in sharp cutoff and decline in the live size.

This pattern is useful if you are building a structure that involves temporarily allocating much more memory than will fit into your *nursery generation*. By applying the ramp allocation pattern, the collection of that generation can be deferred until the ramp allocation is over.

In detail: if the ramp allocation pattern is applied to an *allocation point*, then allocation on that AP is ignored by the MPS when it is deciding whether to schedule a collection of the chain containing the generation into which the AP is allocating. See *Scheduling of collections*.

**Note**

This does not prevent the generation from being collected altogether: there may be other APs allocating into the generation, or the MPS may have to collect the generation in order to avoid running out of memory.

**Note**

Ramp allocation is only supported by *AMC (Automatic Mostly-Copying)*.

```c
mps_alloc_pattern_t mps_alloc_pattern_ramp(void)
    Return an allocation pattern indicating that allocation will follow a ramp allocation pattern.
    This indicates to the MPS that most of the blocks allocated after the call to
    mps_ap_alloc_pattern_begin() are likely to be dead by the time of the corresponding call to
    mps_ap_alloc_pattern_end().

mps_alloc_pattern_t mps_alloc_pattern_ramp_collect_all(void)
    Return an allocation pattern indicating that allocation will follow a ramp allocation pattern, and that the next
    garbage collection following the ramp should be a full collection.
    This indicates to the MPS that most of the blocks allocated after the call to
    mps_ap_alloc_pattern_begin() are likely to be dead by the time of the corresponding call to
    mps_ap_alloc_pattern_end().
    This allocation pattern may nest with, but should not otherwise overlap with, allocation patterns of type
    mps_alloc_pattern_ramp(). In this case, the MPS may defer the full collection until after all ramp
    allocation patterns have ended.
```

2.17 Allocation frames

**Deprecated**

starting with version 1.111.

If you need special handling of stack-like allocation, *contact us*.

An allocation frame is a marker that can pushed onto an *allocation point* by calling `mps_ap_frame_push()`, and then popped by calling `mps_ap_frame_pop()` to indicate that all blocks allocated on the allocation point are *dead* (in the case of *manual* pools), or very likely dead (in the case of *automatic* pools).

Allocation frames can be used by the *client program* to efficiently implement stack-like patterns of allocation.
Note
All pool classes that support allocation points also support pushing and popping of allocation frames, but only the SNC (Stack No Checking) pool class actually uses these frames to manage its blocks.

mps_frame_t
The type of allocation frames.

mps_res_t mps_ap_frame_push (mps_frame_t *frame_o, mps_ap_t ap)
Declare a new allocation frame and push it onto an allocation point's frame stack.
frame_o points to a location that will hold the new frame if the function is successful.
ap is the allocation point in which the new frame is declared.

Returns a result code. The creation of new frames (which is implicit in the action of this function) can consume resources, so this function can fail because there are insufficient resources, or if the correct protocol is not followed by the client program.

mps_res_t mps_ap_frame_pop (mps_ap_t ap, mps_frame_t frame)
Declare that a set of blocks in a allocation frame are dead or likely to be dead, and pop the frame from the allocation point’s frame stack.
ap is the allocation point in which frame was pushed.
frame is the allocation frame whose blocks are likely to be dead.

Returns a result code.

This function pops frame, making its parent the current frame. Popping invalidates frame and all frames pushed since frame. Popping frame also makes a declaration about the set of blocks which were allocated in frame and all frames which were pushed since frame.

The interpretation of this declaration depends on the pool that the allocation point belongs to. Typically, manual pool classes use this declaration to mean that the blocks are dead and their space can be reclaimed immediately, whereas automatic pool classes use this declaration to mean that the blocks are likely to be mostly dead, and may use this declaration to alter its collection decisions. See the documentation for the pool class.

In general a frame other than the current frame can be popped (all frames pushed more recently will be invalidated as well, as described above), but a pool class may impose the restriction that only the current frame may be popped. This restriction means that every push must have a corresponding pop. See the documentation for the pool class.

It is illegal to pop frames out of order (so the sequence “A = push; B = push; pop A; pop B” is illegal) or to pop the same frame twice (so the sequence “A = push, pop A, pop A” is illegal).

2.18 Debugging pools

Several pool classes have debugging counterparts:

<table>
<thead>
<tr>
<th>Pool class</th>
<th>Debugging counterpart</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMS (Automatic Mark and Sweep)</td>
<td>mps_class_ams_debug()</td>
</tr>
<tr>
<td>MV (Manual Variable)</td>
<td>mps_class_mv_debug()</td>
</tr>
<tr>
<td>MVFF (Manual Variable First Fit)</td>
<td>mps_class_mvff_debug()</td>
</tr>
</tbody>
</table>

These debugging pool classes provide two features that are useful for debugging:

- fenceposts are patterns of data that are written before and after each allocated block. In manually managed pools, fenceposts are checked when the block is deallocated, to see that they are unchanged. This helps detect
underwriting and overwriting errors. Fenceposts for all objects in a pool are checked when the pool is destroyed, and can be checked at any time by calling \texttt{mps\_pool\_check\_fenceposts()}.

- \textit{free space splatting} overwrites recycled space with a pattern of data. If the pattern is designed so that it does not resemble a live object (and if code checks the consistency of its data structures), then this helps to detect \textit{dangling pointer} dereferences. The pattern is checked just before allocation, and when a block of memory is released from the pool to the arena, to see that it is unchanged. All free space in a pool can be checked for the pattern at any time by calling \texttt{mps\_pool\_check\_free\_space()}.

The \textit{client program} may optionally specify templates for both of these features via the \texttt{mps\_pool\_debug\_option\_s} structure. This allows it to specify patterns:

- that mimic illegal data values;
- that cause bus errors if wrongly interpreted as pointers;
- that cause assertions to fire if wrongly interpreted as data values;
- that contain an instruction sequence that would cause the program to signal an error or stop if wrongly interpreted as executable code.

For example:

\begin{verbatim}
mps_pool_debug_option_s debug_options = {
   "fencepost", 9,
   "free", 4,
};
mps_pool_t pool;
mps_res_t res;
MPS_ARGS_BEGIN(args) {
   MPS_ARGS_ADD(args, MPS_KEY_POOL_DEBUG_OPTIONS, &debug_options);
   MPS_ARGS_ADD(args, MPS_KEY_FORMAT, &fmt);
   res = mps_pool_create_k(pool, arena, mps_class_ams_debug(), args);
} MPS_ARGS_END(args);
if (res != MPS_RES_OK) error("can't create debug pool");
\end{verbatim}

The type of the structure passed as the value for the optional \texttt{MPS\_KEY\_POOL\_DEBUG\_OPTIONS} keyword argument to \texttt{mps\_pool\_create\_k()} when creating a debugging pool class.

\begin{verbatim}
typedef struct mps_pool_debug_option_s {
   const void *fence_template;
   size_t fence_size;
   const void *free_template;
   size_t free_size;
} mps_pool_debug_option_s;
\end{verbatim}

- \texttt{fence_template} points to a template for fenceposts.
- \texttt{fence_size} is the size of \texttt{fence_template} in bytes (1), or zero if the debugging pool should not use fenceposts.
- \texttt{free_template} points to a template for splatting free space.
- \texttt{free_size} is the size of \texttt{free_template} in bytes, or zero if the debugging pool should not splat free space.

The debugging pool will copy the \texttt{fence_size} bytes pointed to by \texttt{fence_template} in a repeating pattern onto each fencepost during allocation, and it will copy the bytes pointed to by \texttt{free_template} in a repeating pattern over free space after the space is reclaimed.

The MPS may not always use the whole of a template: it may use pieces smaller than the given size, for example to pad out part of a block that was left unused because of alignment requirements.
If the client omits to pass the \texttt{MPS\_KEY\_POOL\_DEBUG\_OPTIONS} keyword argument to \texttt{mps\_pool\_create\_k()}, then the fencepost template consists of the four bytes 50 4F 53 54 (\texttt{POST} in ASCII), and the free space template consists of the four bytes 46 52 45 45 (\texttt{FREE} in ASCII).

\textbf{void} \texttt{mps\_pool\_check\_fenceposts (mps\_pool\_t pool)}

Check all the fenceposts in a pool.

\texttt{pool} is the pool whose fenceposts are to be checked.

If a corrupted fencepost is found, the MPS will \texttt{assert}. It is only useful to call this on a debugging pool that has fenceposts turned on. It does nothing on non-debugging pools.

\textbf{void} \texttt{mps\_pool\_check\_free\_space (mps\_pool\_t pool)}

Check all the free space in a pool for overwriting errors.

\texttt{pool} is the pool whose free space is to be checked.

If corrupted free space is found, the MPS will \texttt{assert}. It is only useful to call this on a debugging pool that has free space splatting turned on. It does nothing on non-debugging pools.

\section*{2.19 Telemetry}

In its \textit{cool} and \textit{hot} varieties, the MPS is capable of outputting a configurable stream of events to assist with debugging and profiling.

The selection of events that appear in the stream is controlled by the environment variable \texttt{MPS\_TELEMETRY\_CONTROL} (by default none), and the stream is written to the file named by the environment variable \texttt{MPS\_TELEMETRY\_FILENAME} (by default \texttt{mpsio.log}).

The telemetry system writes blocks of binary output, and is fast enough to be left turned on in production code (the \textit{hot} variety avoids emitting events on the critical path), which can be useful for diagnosing memory management problems in production environments.

The reporting of garbage collection statistics hasn’t always been suitable for deployment. John McCarthy described the first on-line demonstration of Lisp in an appendix to his paper “\textit{History of Lisp}”:  

\begin{quote}
Everything was going well, if slowly, when suddenly the Flexowriter began to type (at ten characters per second)
\end{quote}

\begin{quote}
\begin{tabular}{|l|}
\hline
\texttt{THE GARBAGE COLLECTOR HAS BEEN CALLED.} \\
\texttt{SOME INTERESTING STATISTICS ARE AS FOLLOWS:} \\
\hline
\end{tabular}
\end{quote}

and on and on and on. The garbage collector was quite new at the time, we were rather proud of it and curious about it, and our normal output was on a line printer, so it printed a full page every time it was called giving how many words were marked and how many were collected and the size of list space, etc. [...] 

Nothing had ever been said about the garbage collector, and I could only imagine the reaction of the audience. We were already behind time on a tight schedule, it was clear that typing out the garbage collector message would take all the remaining time allocated to the demonstration, and both the lecturer and the audience were incapacitated with laughter. I think some of them thought we were victims of a practical joker.

\subsection*{2.19.1 Telemetry utilities}

The telemetry system relies on three utility programs:
• *mpseventcnv* decodes the machine-dependent binary event stream into a portable text format. It must be compiled for the same architecture as the MPS-linked program whose event stream it decodes.

• *mpseventtxt* takes the output of *mpseventcnv* and outputs it in a more human-readable form.

• *mpsevenstsql* takes the output of *mpseventcnv* and loads it into a SQLite database for further analysis.

You must build and install these programs as described in *Building the Memory Pool System*. These programs are described in more detail below.

### 2.19.2 Example

Here’s an example of turning on telemetry in the debugger and then encountering a corrupted object:

```bash
$ gdb ./scheme
GNU gdb 6.3.50-20050815 (Apple version gdb-1820) (Sat Jun 16 02:40:11 UTC 2012)
[...]
(gdb) set environment MPS_TELEMETRY_CONTROL=all
(gdb) run
Starting program: example/scheme/scheme
Reading symbols for shared libraries +............................. done
MPS Toy Scheme Example
[...]
7944, 0> (gc)
[...]
7968, 1> foo
Assertion failed: (TYPE(frame) == TYPE_PAIR), function lookup_in_frame, file scheme.c, line 1066.
```

Program received signal SIGABRT, Aborted.
0x00007fff91aeed46 in __kill ()

At this point there’s still output in the MPS’s internal event buffers, which needs to be flushed. It would be a good idea to add a call to *mps_telemetry_flush()* to the error handler, but for now we can just call it directly from the debugger:

```bash
(gdb) print mps_telemetry_flush()
$1 = void
```

The MPS writes the telemetry to the log in an encoded form for speed. It can be decoded using the *mpseventcnv* and *mpseventtxt* programs:

```bash
(gdb) shell mpseventcnv | sort | mpseventtxt > mpsio.txt
```

The *sort* is useful because the events are not necessarily written to the telemetry file in time order, but each event starts with a timestamp so sorting makes a time series. The decoded events look like this, with the timestamp in the first column, the event type in the second column, and then addresses or other data related to the event in the remaining columns. The source of the timestamp depends on the platform; it may be a low-cost high-resolution processor timer, such as the Time Stamp Counter on IA-32 and x86-64, if one is available. All numbers are given in hexadecimal.

```plaintext
- 000AE0397336E3C 002B VMCreate  
  vm:00000001003FC000 base:0000001003FD000 limit:00000001003FE000
- 000AE0397333BC6D 002D VMMap  
  vm:00000001003FC000 base:00000001003FD000 limit:00000001003FE000
- 000AE0397334DF9F 001A Intern  
  stringId:0000000000000002 string:"Reservoir"
- 000AE0397334E0A0 001B Label  
  address:00000001078C85B8"Reservoir" stringId:0000000000000002
- 000AE03973352375 0015 PoolInit  
  pool:00000001003FD328 arena:00000001003FF000 poolClass:00000001078C85B8
- 000AE039733592F9 002B VMCreate  
  vm:00000001003FE000 base:00000001003FF000 limit:00000001003FE000
- 000AE0397335C8B5 002D VMMap  
  vm:00000001003FD000 base:00000001003FF000 limit:00000001003FE000
- 000AE03973361D5A 0005 ArenaCreateVM  
  arena:00000001003FD000 userSize:0000000002000000 chunkSize:0000000002000000
```

You can search through the telemetry for events related to particular addresses of interest.
In the example, we might look for events related to the address of the corrupted frame object:

(gdb) frame 3
#3 0x0000000100003f55 in lookup_in_frame (frame=0x1003fa7d0, symbol=0x1003faf20) at scheme.c:1066
1066 assert(TYPE(frame) == TYPE_PAIR);
(gdb) print frame
$2 = (obj_t) 0x1003fa7d0
(gdb) shell grep -i 1003fa7d0 mpsio.txt || echo not found
not found

There are no events related to this address, so in particular this address was never fixed (no TraceFix event).

Note
You may find it useful to add the command:

set environment MPS_TELEMETRY_CONTROL=all

to your .gdbinit.

### 2.19.3 Event categories

The “bit” column gives the bit number in the telemetry filter. These numbers are liable to change, but the current meanings (zero being the least significant bit) are:

<table>
<thead>
<tr>
<th>Bit</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Arena</td>
<td>Per space or arena.</td>
</tr>
<tr>
<td>1</td>
<td>Pool</td>
<td>Per pool.</td>
</tr>
<tr>
<td>2</td>
<td>Trace</td>
<td>Per trace or scan.</td>
</tr>
<tr>
<td>3</td>
<td>Seg</td>
<td>Per page (segment).</td>
</tr>
<tr>
<td>4</td>
<td>Ref</td>
<td>Per reference or fix.</td>
</tr>
<tr>
<td>5</td>
<td>Object</td>
<td>Per allocation, block, or object.</td>
</tr>
<tr>
<td>6</td>
<td>User</td>
<td>User-invoked events: see mps_telemetry_intern().</td>
</tr>
</tbody>
</table>

### 2.19.4 Environment variables

In the ANSI plinth (the plinth that comes as default with the MPS), these two environment variables control the behaviour of the telemetry feature.

**MPS_TELEMETRY_CONTROL**

The event categories which should be included in the telemetry stream.

If its value can be interpreted as a number, then this number represents the set of event categories as a bitmap. For example, this turns on the Pool and Seg event categories:

```
MPS_TELEMETRY_CONTROL=6
```

Otherwise, the value is split into words at spaces, and any word that names an event category turns it on. For example:

```
MPS_TELEMETRY_CONTROL="arena pool trace"
```

The special event category all turns on all events.

**MPS_TELEMETRY_FILENAME**

The name of the file to which the telemetry stream should be written. Defaults to mpsio.log. For example:
MPS_TELEMETRY_FILENAME=$(mktemp -t mps)

In addition, the following environment variable controls the behaviour of the mpseventsql program.

**MPS_TELEMETRY_DATABASE**

The name of a SQLite database file that will be updated with the events from the decoded telemetry stream, if it is not specified with the -d option. If this variable is not assigned, mpsevent.db is used.

### 2.19.5 Decoding the telemetry stream

The MPS writes the telemetry stream in a binary encoded format for speed. The encoding is specific to the platform the program was running on, and so the output needs to be decoded before it can be processed.

The decoding takes place in two stages. First, the program mpseventcnv converts the binary encoded format into a portable text format suitable for input to one of the second-stage tools (mpseventtxt and mpseventsql).

- **-f** <filename>
  
  The name of the file containing the telemetry stream to decode. Defaults to mpsio.log.

- **-h**
  
  Help: print a usage message to standard output.

**Note**

mpseventcnv can only read telemetry streams that were written by an MPS compiled on the same platform.

Here’s some example output. The first column contains the timestamp of the event, the second column contains the event type, and remaining columns contain parameters related to the event.

<table>
<thead>
<tr>
<th>Timestamp</th>
<th>Type</th>
<th>Parameter 1</th>
<th>Parameter 2</th>
<th>Parameter 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>000AE03973336E3C 2B</td>
<td>1003FC000</td>
<td>1003FD000</td>
<td>1003FE000</td>
<td></td>
</tr>
<tr>
<td>000AE0397333BC6D 2D</td>
<td>1003FC000</td>
<td>1003FD000</td>
<td>1003FE000</td>
<td></td>
</tr>
<tr>
<td>000AE0397334DF9F 1A</td>
<td>2 &quot;Reservoir&quot;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000AE0397334E0A0 1B</td>
<td>1078C85B8 2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000AE03973352375 15</td>
<td>1003FD328</td>
<td>1003FD000</td>
<td>1078C85B8</td>
<td></td>
</tr>
<tr>
<td>000AE039733592F9 2B</td>
<td>1003FE000</td>
<td>1003FF000</td>
<td>10992F000</td>
<td></td>
</tr>
<tr>
<td>000AE0397335C8B5 2D</td>
<td>1003FE000</td>
<td>1003FF000</td>
<td>10793000</td>
<td></td>
</tr>
<tr>
<td>000AE03973361D5A 5</td>
<td>1003FD000</td>
<td>2000000</td>
<td>2000000</td>
<td></td>
</tr>
</tbody>
</table>

### 2.19.6 Making the telemetry stream readable

The output of mpseventcnv can be made more readable by passing it through mpseventtxt, which takes the following options:

- **-l** <filename>
  
  The name of a file containing telemetry events that have been decoded by mpseventcnv. Defaults to standard input.

- **-h**
  
  Help: print a usage message to standard output.

For example, here’s the result of passing the output shown above through mpseventtxt:

| VMCreate | vm:00000001003FC000 base:00000001003FD000 limit:00000001003FE000 |
| VMMAP | vm:00000001003FC000 base:00000001003FD000 limit:00000001003FE000 |
| Intern | stringId:0000000000000002 string:"Reservoir"
| Label | address:00000001078C85B8["Reservoir"] stringId:0000000000000001 |
| PoolInit | pool:00000001003FD328 arena:00000001003FD000 poolClass:00000001003FD000 |
2.19.7 Loading the telemetry stream into SQLite

The decoded telemetry stream (as output by `mpseventcnv`) can be loaded into a SQLite database for further analysis by running `mpseventsqrl`.

`mpseventsqrl` takes the following options:

- `-i <filename>`
  The name of a file containing a decoded telemetry stream. Defaults to standard input.

- `-o <filename>`
  The name of a SQLite database file that will be updated with the events from the decoded telemetry stream specified by the `-l` option. The database will be created if it does not exist. If not specified, the file named by the environment variable `MPS_TELEMERY_DATABASE` is used; if this variable is not assigned, `mpsevent.db` is used.

  Updating a database with events from a file is idempotent unless the `-f` option is specified.

- `-d`
  Delete the database before importing.

- `-f`
  Forces the database to be updated with events from the decoded telemetry stream specified by the `-i` option, even if those events have previously been added.

- `-v`
  Increase the verbosity. With one or more `-v` options, `mpseventsqrl` prints informative messages to standard error. Verbosity levels up to 3 (`-vvv`) produce successively more detailed information.

  This option implies `-p`.

- `-p`
  Show progress by printing a dot to standard output for every 100,000 events processed.

- `-t`
  Run internal tests.

- `-r`
  Rebuild the tables `event_kind`, `event_type`, and `event_param`. (This is necessary if you changed the event descriptions in `eventdef.h`.)

2.19.8 Telemetry interface

```c
void mps_telemetry_flush(void)

Flush the internal event buffers into the telemetry stream.

This function also calls `mps_io_flush()` on the event stream itself. This ensures that even the latest events are now properly recorded, should the client program terminate (uncontrollably as a result of a bug, for example) or some interactive tool require access to the telemetry stream.
```

**Note**

Unless all arenas are properly destroyed (by calling `mps_arena_destroy()`), there are likely to be unflushed telemetry events when the program finishes. So in the case of abnormal program termination such as a fatal exception, you may want to call `mps_telemetry_flush()` explicitly.
mps_word_t mps_telemetry_get (void)

    Return the telemetry filter.

void mps_telemetry_set (mps_word_t set_mask)

    Set bits in the telemetry filter.

    set_mask is a bitmask indicating the bits in the telemetry filter that should be set.

void mps_telemetry_reset (mps_word_t reset_mask)

    Reset bits in the telemetry filter.

    reset_mask is a bitmask indicating the bits in the telemetry filter that should be reset.

2.19.9 Telemetry labels

Telemetry labels allow the client program to associate strings with addresses in the telemetry stream. The string must first be interned by calling mps_telemetry_intern(), returning a label, and then the address can be associated with the label by calling mps_telemetry_label().

Typical uses of telemetry labels include:

- labelling pools with a human-meaningful name;
- labelling allocated objects with their type, class, or other description.

It is necessary to enable User events in the telemetry filter in order for telemetry labels to work. For example:

```c
mps_label_t label;
mps_telemetry_set(1 << 6);
label = mps_telemetry_intern("symbol pool");
mps_telemetry_label(symbol_pool, label);
```

Labels are represented by the type mps_label_t. These are unsigned integers. After processing by mpseventsql, the association of addresses with labels appears in the EVENT_Label table, and the association of labels with strings appears in the EVENT_Intern table. These can then be used in queries, for example:

```sql
/* Pool name and creation time */
SELECT I.string, P.time
FROM EVENT_PoolInit AS P,
    EVENT_Label AS L,
    EVENT_Intern AS I
WHERE I.stringId = L.stringId AND L.address = P.pool;
```

mps_label_t mps_telemetry_intern (const char *label)

    Registers a string with the MPS, and receives a telemetry label, suitable for passing to mps_telemetry_label().

    label is a NUL-terminated string. Its length should not exceed 256 characters, including the terminating NUL.

    Returns a telemetry label: a unique identifier that may be used to represent the string in future.

    The intention of this function is to provide an identifier that can be used to concisely represent a string for the purposes of mps_telemetry_label().

    **Note**

    If the User event category is not turned on in the telemetry filter (via mps_telemetry_control()) then the string is not sent to the telemetry stream. A label is still returned in this case, but it is useless.
void \texttt{mps\_telemetry\_label}(\texttt{mps\_addr\_t addr, mps\_label\_t label})

Associate a telemetry label returned from \texttt{mps\_telemetry\_intern()} with an address.

\texttt{addr} is an address.

\texttt{label} is a telemetry label returned from \texttt{mps\_telemetry\_intern()}.

The label will be associated with the address when it appears in the telemetry stream.

\begin{itemize}
\item \textbf{Note}
If the User event category is not turned on in the telemetry filter (via \texttt{mps\_telemetry\_control()}) then calling this function has no effect.
\end{itemize}

\section*{2.19.10 Customizing the telemetry system}

If you need the telemetry system to support features not described here (for example, you need to transmit telemetry data over a network rather than writing it to a file on the local filesystem) then you may be able to do so by providing your own implementation of the I/O module.

When it first needs to output telemetry, the MPS call the plinth function \texttt{mps\_io\_create()} to create an I/O stream. It then calls \texttt{mps\_io\_write()} to write binary data to the stream and \texttt{mps\_io\_flush()} to flush the stream in response to \texttt{mps\_telemetry\_flush()}. By providing your own implementations of these functions, you can direct the telemetry stream wherever you like.

See \textit{Plinth} for details.

\section*{2.20 Weak references}

A \textit{weak reference} is a \textit{reference} that does not keep the block it refers to \textit{alive}.

The open source MPS supports weak references only:

1. in \textit{roots} that are registered with \texttt{rank mps\_rank\_weak();}
2. in objects allocated on an \textit{allocation point} in a pool of class \textit{AWL (Automatic Weak Linked)} that was created with \texttt{rank mps\_rank\_weak().}

\begin{itemize}
\item \textbf{Note}
If you need more general handling of weak references, \textit{contact us}.
\end{itemize}

When the MPS determines that a block is only kept alive by one or more weak references, it may choose to \textit{splat} those references by replacing them with null pointers when they are \textit{fixed}. When all weak references to the block have been splatted, the block may be reclaimed.

For example, a \textit{scan method} for objects in an AWL pool might look like this:

\begin{verbatim}
mps\_res\_t obj\_scan(mps\_ss\_t ss, mps\_addr\_t base, mps\_addr\_t limit)
{
    MPS\_SCAN\_BEGIN(ss) {
        \textbf{while} (base < limit) {
            obj\_t obj = base;
            mps\_addr\_t p = obj->ref;
            if (MPS\_FIX1(ss, p)) {
                mps\_res\_t res = MPS\_FIX2(ss, &p);
                if (res != MPS\_RES\_OK) \textbf{return} res;
            }
        }
    }
}
\end{verbatim}
if (p == NULL) {
    /* reference was splatted */
}

obj->ref = p;

base += sizeof(*obj);

} MPS_SCAN_END(ss);
return MPS_RES_OK;

A reference that passes the “interesting” test MPS_FIX1() can’t be a null pointer, so if the reference is discovered to be null after calling MPS_FIX2() then it must have just been splatted.

Note
Because weak references are splatted when they are fixed, not all weak references to a block are splatted at the same time. Depending on the decisions the MPS makes about which objects to scan, a weak reference may live on for some time after other weak references to the same block have been splatted.

Note
A common way in which weak references are used in programming languages is in weak-key and weak-value hash tables. A weak-key hash table contains weak references to its keys: when it detects that a key has been splatted, it deletes the corresponding value. The AWL (Automatic Weak Linked) pool class supports this by allowing you to specify for each object, a dependent object which may be written to by the scan method. See Dependent objects.

Note
Weak references do not prevent blocks from being finalized. At the point that a block is finalized, weak references will still validly refer to the block. The fact that a block is registered for finalization prevents weak references to that block from being splatted. See Finalization.

2.21 Plinth

The plinth is a program module that provides the MPS with the support it needs from the execution environment. The MPS uses the plinth instead of (say) the Standard C Library because:

1. The MPS is designed to be portable to systems that have only a freestanding implementation of the C language: that is, systems which potentially lack some of the facilities of the Standard C Library, such as standard I/O. The plinth provides a way to map MPS requirements to the facilities provided on the platform, whatever they are.

2. The plinth gives the client program complete control of interaction between the MPS and the user, including assertions and telemetry.

The plinth may be provided by the client program; however, a sample implementation of the plinth using ANSI Standard C Library facilities is included with the MPS, and this is good enough for most applications.

There are many reasons why you might want to write your own plinth. You may be targeting an embedded system with only a freestanding implementation of the C language. You might need to write the telemetry stream to a system logging facility, or transmit it over a serial port or network connection. Or you might need to direct debugging output to a convenient window in the user interface.

The plinth is divided into two parts:
1. The **I/O module** provides general-purpose I/O functionality. It is used to output a *telemetry stream* of events to assist with debugging and profiling.

2. The **Library module** provides miscellaneous functionality that would be available via the Standard C Library on a hosted platform, including functions for reporting errors and accessing a processor clock.

The functions in the plinth module may be called in the context of a signal handler for a protection fault (or equivalent), so they must not access memory that is managed by the MPS, and they need to take into account the restrictions imposed by the operating system. (See “Defining Signal Handlers” in the GNU C Library Reference Manual for useful advice.)

**CONFIG_PLINTH_NONE**

If this preprocessor constant is defined, exclude the ANSI plinth (mpsioan.c and mpsliban.c) from the MPS. For example:

```
c -DCONFIG_PLINTH_NONE -c mps.c  (Unix/OS X)
c l /Gs /DCONFIG_PLINTH_NONE /c mps.c  (Windows)
```

Having excluded the ANSI plinth, you must of course supply your own.

### 2.21.1 I/O module

```c
#include "mpsio.h"
```

**mps_io_t**

The type of an I/O stream.

This is an alias for a pointer to the incomplete structure `mps_io_s`, which the plinth may define if it needs to. Alternatively, it may leave the structure type undefined and simply cast its own pointer to and from `mps_io_t`.

**Note**

In the ANSI I/O module, mpsioan.c, this is an alias for `FILE *`.

**mps_res_t mps_io_create(mps_io_t *io_o)**

A plinth function for creating an I/O stream for the *telemetry stream*.

- `io_o` points to a location suitable for storing a pointer to an I/O stream.

- If successful, the function must update this location with a suitable pointer for the telemetry stream and return `MPS_RES_OK`. Otherwise, it must return some other result code.

The MPS calls this function to create the I/O stream for telemetry output. A typical plinth will use it to open a file for writing, or to connect to the system logging interface.

**Note**

In the ANSI I/O module, mpsioan.c, this calls `fopen()` on the file named by the environment variable `MPS_TELEMETRY_FILENAME`.

`void mps_io_destroy(mps_io_t io)`

A plinth function for destroying an I/O stream.

- `io` is a pointer to the I/O stream to be destroyed. It was previously created by a call to `mps_io_create()`.

- After calling this function, the MPS guarantees not to use the value `io` again.

**Note**

In the ANSI I/O module, mpsioan.c, this calls `fclose()`.
mps_res_t mps_io_write(mps_io_t io, void *buf, size_t size)
A plinth function for writing data to an I/O stream.

io is the I/O stream.
buf points to the data to write.
size is the size of the data in bytes (1).

Returns MPS_RES_OK if successful.

Note
In the ANSI I/O module, mpsioan.c, this calls fwrite().

mps_res_t mps_io_flush(mps_io_t io)
A plinth function for flushing an I/O stream.

io is the I/O stream.

Returns MPS_RES_OK if successful.

The MPS calls this function when it is done with the telemetry stream, or when the client program calls mps_telemetry_flush(). This function should ensure that the buffers of data passed to the latest calls to mps_io_write() are properly recorded, should the client program terminate (uncontrollably as a result of a bug, for example) or some interactive tool require access to the event data.

Note
In the ANSI I/O module, mpsioan.c, this calls fflush().

2.21.2 Library module

#include "mpslib.h"

mps_clock_t mps_clock(void)
Return the time since some epoch, in units given by mps_clocks_per_sec().

Note
The ANSI Library module, mpsliban.c, calls clock.

The MPS calls this function to make scheduling decisions (see Scheduling of collections), and to calibrate the time stamps on events in the telemetry stream. If your platform has a low-resolution clock(), and there are higher-resolution clocks readily available, then using one of those will improve MPS scheduling decisions and the quality of telemetry output. For instance, with getrusage():

#include <sys/resource.h>

mps_clock_t mps_clock(void) {
    struct rusage s;
    int res = getrusage(RUSAGE_SELF, &s);
    if (res != 0) {
        /* handle error */
    }
    return ((mps_clock_t)s.ru_utime.tv_sec) * 1000000 + s.ru_utime.tv_usec;
}
**mps_clock_t mps_clocks_per_sec (void)**
Return the number of clock units per second, as returned by `mps_clock()`.

---

**Note**
The ANSI Library module, `mpsliban.c`, returns `CLOCKS_PER_SEC`.

---

`void mps_lib_assert_fail (const char *message)`
Report an assertion failure.

message is a NUL-terminated string describing the assertion failure.

---

**Note**
In the ANSI Library module, `mpsliban.c`, this reports the failure by calling `fprintf(stderr, "%s", message)`, flushes the telemetry stream by calling `mps_telemetry_flush()`, and, in the *cool variety*, terminates the program by calling `abort()`. You can change this behaviour with `mps_lib_assert_fail_install()`. For a discussion of the default behaviour, see Assertion handling.

**Warning:** This function must not call any function in MPS, and it must not access memory managed by the MPS.

---

**Warning:** The installed assertion handler must not call any function in MPS, and it must not access memory managed by the MPS.

---

`extern mps_lib_assert_fail_t mps_lib_assert_fail_install (mps_lib_assert_fail_t handler)`
This function customises the behaviour of the default assertion handler in the ANSI Library module. It is not otherwise required by the MPS and you need not implement it if you are providing an alternative plinth.

If you’re using the ANSI Library module, you can use this function to change the behaviour of the MPS when an assertion fails. For example, you could terminate the program in the *hot variety* too. (The MPS test programs do exactly that.)

handler is the assertion handler to install.

Returns the previously installed handler.

---

**Warning:** The type of assertion handlers passed to and returned by `mps_lib_assert_fail_install()`.

---

**mps_lib_FILE**
The type of output streams provided by the plinth.

---

**Note**
In the ANSI Library module, `mpsliban.c`, this is an alias for `FILE *`.

---

`int mps_lib_fputc (int c, mps_lib_FILE *stream)`
Write a character to an output stream.

`c` is the character.

stream is the stream.

Return the character written if successful, or `mps_lib_get_EOF()` if not.

This function is intended to have the same semantics as the `fputc()` function of the ANSI C Standard (*ISO/IEC 9899:1990* §7.11.7.3).
In the ANSI Library module, **mpsliban.c**, this is a simple wrapper around `fputc()`.

```c
int mps_lib_fputs (const char *, mps_lib_FILE *)
```

Write a string to an output stream.

- `s` is the NUL-terminated string.
- `stream` is the stream.

This function is intended to have the same semantics as the `fputs()` function of the ANSI C Standard (*ISO/IEC 9899:1990* §7.11.7.4).

Return a non-negative integer if successful, or `mps_lib_get_EOF()` if not.

In the ANSI Library module, **mpsliban.c**, this is a simple wrapper around `fputs()`.

```c
int mps_lib_get_EOF (void)
```

Return the value that is returned from `mps_lib_fputc()` and `mps_lib_fputs()` to indicate failure.

Note

In the ANSI Library module, **mpsliban.c**, this returns `EOF`.

Returns an output stream suitable for reporting errors.

Note

In the ANSI Library module, **mpsliban.c**, this returns `stderr`.

Note

The MPS does not use this at present, but it may be required in future.

```c
mps_lib_FILE *mps_lib_get_stdout (void)
```

Returns an output stream suitable for reporting informative output.

Note

In the ANSI Library module, **mpsliban.c**, this returns `stdout`.

Note

The MPS does not use this at present, but it may be required in future.

```c
int mps_lib_memcmp (const void *, const void *, size_t)
```

A *plinth* function similar to the standard C function `memcmp()`.

- `s1` and `s2` point to *blocks* of memory to be compared.
- `n` is the *size* of the blocks.

Returns an integer that is greater than, equal to, or less than zero, accordingly as the block pointed to by `s1` is greater than, equal to, or less than the block pointed to by `s2`.
This function is intended to have the same semantics as the `memcmp()` function of the ANSI C Standard (ISO/IEC 9899:1990 §7.11.4.1).

**Note**
In the ANSI Library module, mpsliban.c, this is a simple wrapper around `memcmp()`.

```c
void *mps_lib_memcpy(void *dest, const void *source, size_t n)
A plinth function similar to the standard C function `memcpy()`.
dest points to the destination.
source points to the source.
n is the number of bytes to copy from source to dest.
Returns dest.
```

This function is intended to have the same semantics as the `memcpy()` function of the ANSI C Standard (ISO/IEC 9899:1990 §7.11.2.1).

The MPS never passes overlapping blocks to `mps_lib_memcpy()`.

**Note**
In the ANSI Library module, mpsliban.c, this is a simple wrapper around `memcpy()`.

```c
void *mps_lib_memset(void *s, int c, size_t n)
A plinth function similar to the standard C function `memset()`.
s points to the block to fill with the byte c.
c is the byte to fill with (when converted to unsigned char).
n is the size of the block.
Returns s.
```

This function is intended to have the same semantics as the `memset()` function of the ANSI C Standard (ISO/IEC 9899:1990 §7.11.6.1).

**Note**
In the ANSI Library module, mpsliban.c, this is a simple wrapper around `memset()`.

**Note**
The MPS does not use this at present, but it may be required in future.

```c
unsigned long mps_lib_telemetry_control()
A plinth function to supply a default value for the telemetry filter from the environment. See
mps_telemetry_control() for more information on the significant of the value.
```

Returns the default value of the telemetry filter, as derived from the environment. It is recommended that the environment be consulted for a symbol analogous to `MPS_TELEMETRY_CONTROL`, subject to local restrictions.

In the absence of environmental data, a default of zero is recommended.

**Note**
In the ANSI Library module, mpsliban.c, this reads the environment variable `MPS_TELEMETRY_CONTROL`.

---

2.21. Plinth
2.22 Platforms

2.22.1 Platform codes

The MPS uses a six-character platform code to express a combination of operating system, CPU architecture, and compiler toolchain. Each six-character code breaks down into three pairs of characters: OSARCT. The first pair of characters names the operating system:

<table>
<thead>
<tr>
<th>OS</th>
<th>Operating system</th>
<th>Constant</th>
</tr>
</thead>
<tbody>
<tr>
<td>fr</td>
<td>FreeBSD</td>
<td>MPS_OS_FR</td>
</tr>
<tr>
<td>li</td>
<td>Linux</td>
<td>MPS_OS_LI</td>
</tr>
<tr>
<td>w3</td>
<td>Windows</td>
<td>MPS_OS_W3</td>
</tr>
<tr>
<td>xc</td>
<td>OS X</td>
<td>MPS_OS_XC</td>
</tr>
</tbody>
</table>

The second pair of characters names the processor architecture:

<table>
<thead>
<tr>
<th>AR</th>
<th>Processor architecture</th>
<th>Constant</th>
</tr>
</thead>
<tbody>
<tr>
<td>i3</td>
<td>Intel/AMD IA-32</td>
<td>MPS_ARCH_I3</td>
</tr>
<tr>
<td>i6</td>
<td>Intel/AMD x86-64</td>
<td>MPS_ARCH_I6</td>
</tr>
</tbody>
</table>

The third pair of characters names the compiler toolchain:

<table>
<thead>
<tr>
<th>CT</th>
<th>Compiler toolchain</th>
<th>Constant</th>
</tr>
</thead>
<tbody>
<tr>
<td>gc</td>
<td>GNU Compiler collection</td>
<td>MPS_BUILD_GC</td>
</tr>
<tr>
<td>ll</td>
<td>Clang/LLVM</td>
<td>MPS_BUILD_LL</td>
</tr>
<tr>
<td>mv</td>
<td>Microsoft Visual C/C++</td>
<td>MPS_BUILD_MV</td>
</tr>
</tbody>
</table>

In each case the aspect of the platform can be tested by checking whether the preprocessor constant in the third column in the table is defined, and the full platform can be tested by checking whether the corresponding MPS_PF_ preprocessor constant is defined. For example, “xcI611” platform corresponds to the MPS_PF_XCI6LL preprocessor constant.

Not all combinations of operating system, processor architecture, and compiler are supported.

2.22.2 Platform interface

```c
#include "mpstd.h"
```

**MPS_ARCH_I3**
A C preprocessor macro that indicates, if defined, that the target processor architecture of the compilation is a member of the IA-32 Intel/AMD family of 32-bit processors.

**MPS_ARCH_I6**
A C preprocessor macro that indicates, if defined, that the target processor architecture of the compilation is a member of the x86-64 Intel/AMD family of 64-bit processors.

**Note**
The MPS is not supported on IA-64 (Itanium).

**MPS_BUILD_GC**
A C preprocessor macro that indicates, if defined, that the MPS was compiled by the C compiler from the GNU Compiler Collection (GCC).
MPS_BUILD_LL
A C preprocessor macro that indicates, if defined, that the MPS was compiled by Clang, the C compiler from the LLVM (Low Level Virtual Machine) system.

MPS_BUILD_MV
A C preprocessor macro that indicates, if defined, that the MPS was compiled by the C compiler from Microsoft Visual Studio.

MPS_OS_FR
A C preprocessor macro that indicates, if defined, that the MPS was compiled on a FreeBSD operating system.

MPS_OS_LI
A C preprocessor macro that indicates, if defined, that the MPS was compiled on a Linux operating system.

MPS_OS_W3
A C preprocessor macro that indicates, if defined, that the MPS was compiled on a Windows operating system.

MPS_OS_XC
A C preprocessor macro that indicates, if defined, that the MPS was compiled on an OS X operating system.

MPS_PF_ALIGN
A C preprocessor macro that expands to an integer giving the natural alignment of the platform.

MPS_PF_FRI3GC
A C preprocessor macro that indicates, if defined, that the platform consists of the FreeBSD operating system, the IA-32 processor architecture, and the GCC compiler.

MPS_PF_FRI6GC
A C preprocessor macro that indicates, if defined, that the platform consists of the FreeBSD operating system, the x86-64 processor architecture, and the GCC compiler.

MPS_PF_LII3GC
A C preprocessor macro that indicates, if defined, that the platform consists of the Linux operating system, the IA-32 processor architecture, and the GCC compiler.

MPS_PF_LII6GC
A C preprocessor macro that indicates, if defined, that the platform consists of the Linux operating system, the x86-64 processor architecture, and the GCC compiler.

MPS_PF_LII6LL
A C preprocessor macro that indicates, if defined, that the platform consists of the Linux operating system, the x86-64 processor architecture, and the Clang/LLVM compiler.

MPS_PF_STRING
A C preprocessor macro that names the platform for which the MPS was built.

MPS_PF_W3I3MV
A C preprocessor macro that indicates, if defined, that the platform consists of the Windows operating system, the IA-32 processor architecture, and the Microsoft Visual C/C++ compiler.

MPS_PF_W3I6MV
A C preprocessor macro that indicates, if defined, that the platform consists of the Windows operating system, the x86-64 processor architecture, and the Microsoft Visual C/C++ compiler.

MPS_PF_XCI3GC
A C preprocessor macro that indicates, if defined, that the platform consists of the OS X operating system, the IA-32 processor architecture, and the GCC compiler.

MPS_PF_XCI3LL
A C preprocessor macro that indicates, if defined, that the platform consists of the OS X operating system, the IA-32 processor architecture, and the Clang/LLVM compiler.
MPS_PF_XCI6LL
A C preprocessor macro that indicates, if defined, that the platform consists of the OS X operating system, the x86-64 processor architecture, and the Clang/LLVM compiler.

MPS_T_ULONGEST
A C preprocessor macro that expands to the name of the largest unsigned integral type.

The exact identity of this type is platform-dependent. Typical identities are unsigned long and unsigned __int_64.

MPS_T_WORD
A C preprocessor macro that expands to the name of an unsigned integral type that is the same size as an object pointer, so that sizeof(MPS_T_WORD) == sizeof(void*).

The exact identity of this type is platform-dependent. Typical identities are unsigned long and unsigned __int_64.

MPS_WORD_SHIFT
A C preprocessor macro that expands to the logarithm to base 2 of the constant MPS_WORD_WIDTH, so that 1 << MPS_WORD_SHIFT == MPS_WORD_WIDTH.

The value is platform-dependent. Typical values are 5 and 6.

MPS_WORD_WIDTH
A C preprocessor macro that expands to the width in bits of the type MPS_T_WORD, so that MPS_WORD_WIDTH == sizeof(MPS_T_WORD) * CHAR_BIT.

This value is platform-dependent. It is always a power of 2: typical values are 32 and 64.

### 2.22.3 Historical platform codes

The platform codes in the tables below were in use in older versions of the Memory Pool System, but are not currently supported.

**Formerly supported operating systems:**

<table>
<thead>
<tr>
<th>OS</th>
<th>Operating system</th>
<th>Constant</th>
</tr>
</thead>
<tbody>
<tr>
<td>i5</td>
<td>Irix 5 or 6 (old ABI)</td>
<td>MPS_OS_I5</td>
</tr>
<tr>
<td>ia</td>
<td>Irix 6 (new ABI)</td>
<td>MPS_OS_IA</td>
</tr>
<tr>
<td>o1</td>
<td>OSF/1 aka Tru64</td>
<td>MPS_OS_O1</td>
</tr>
<tr>
<td>s7</td>
<td>Macintosh System 7, 8, or 9</td>
<td>MPS_OS_S7</td>
</tr>
<tr>
<td>so</td>
<td>Solaris</td>
<td>MPS_OS_SO</td>
</tr>
<tr>
<td>su</td>
<td>SunOS</td>
<td>MPS_OS_SU</td>
</tr>
</tbody>
</table>

**Formerly supported processor architectures:**

<table>
<thead>
<tr>
<th>AR</th>
<th>Processor architecture</th>
<th>Constant</th>
</tr>
</thead>
<tbody>
<tr>
<td>i4</td>
<td>Intel/AMD IA-32 ¹</td>
<td>MPS_ARCH_I4</td>
</tr>
<tr>
<td>al</td>
<td>Digital Alpha</td>
<td>MPS_ARCH_AL</td>
</tr>
<tr>
<td>m2</td>
<td>MIPS R2000</td>
<td>MPS_ARCH_M2</td>
</tr>
<tr>
<td>m4</td>
<td>MIPS R4000</td>
<td>MPS_ARCH_M4</td>
</tr>
<tr>
<td>m6</td>
<td>Motorola 68000</td>
<td>MPS_ARCH_M6</td>
</tr>
<tr>
<td>pp</td>
<td>PowerPC</td>
<td>MPS_ARCH_PP</td>
</tr>
<tr>
<td>s8</td>
<td>SPARC V8</td>
<td>MPS_ARCH_S8</td>
</tr>
<tr>
<td>s9</td>
<td>SPARC V9 (32-bit)</td>
<td>MPS_ARCH_S9</td>
</tr>
</tbody>
</table>

**Formerly supported compiler toolchains:**

¹Obsolete: the MPS used to make a distinction between the 80386 and 80486 processor architectures.
### Historical platform list

This is the full list of platforms that have ever been supported by the Memory Pool System, with their current status.

<table>
<thead>
<tr>
<th>Platform</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>fri3gc</td>
<td>Supported</td>
</tr>
<tr>
<td>fri4gc</td>
<td>Corrected to fri3gc</td>
</tr>
<tr>
<td>fri6gc</td>
<td>Supported</td>
</tr>
<tr>
<td>i5m2cc</td>
<td>Not supported</td>
</tr>
<tr>
<td>iam4cc</td>
<td>Not supported</td>
</tr>
<tr>
<td>lii3eg</td>
<td>Not supported</td>
</tr>
<tr>
<td>lii3gc</td>
<td>Supported</td>
</tr>
<tr>
<td>lii4gc</td>
<td>Corrected to lii3gc</td>
</tr>
<tr>
<td>lii6gc</td>
<td>Supported</td>
</tr>
<tr>
<td>lii611</td>
<td>Supported</td>
</tr>
<tr>
<td>lippgc</td>
<td>Not supported</td>
</tr>
<tr>
<td>olalcc</td>
<td>Not supported</td>
</tr>
<tr>
<td>olalgc</td>
<td>Not supported</td>
</tr>
<tr>
<td>s7m6mw</td>
<td>Not supported</td>
</tr>
<tr>
<td>s7ppac</td>
<td>Not supported</td>
</tr>
<tr>
<td>s7ppmw</td>
<td>Not supported</td>
</tr>
<tr>
<td>sos8cx</td>
<td>Not supported</td>
</tr>
<tr>
<td>sos8gc</td>
<td>Not supported</td>
</tr>
<tr>
<td>sos8gp</td>
<td>Not supported</td>
</tr>
<tr>
<td>sos9sc</td>
<td>Not supported</td>
</tr>
<tr>
<td>sus8gc</td>
<td>Not supported</td>
</tr>
<tr>
<td>w3almv</td>
<td>Not supported</td>
</tr>
<tr>
<td>w3l3m9</td>
<td>Not supported</td>
</tr>
<tr>
<td>w3l3mv</td>
<td>Supported</td>
</tr>
<tr>
<td>w3l3pc</td>
<td>Not supported</td>
</tr>
<tr>
<td>w3l6mv</td>
<td>Supported</td>
</tr>
<tr>
<td>w3l6pc</td>
<td>Not supported</td>
</tr>
<tr>
<td>w3ppmv</td>
<td>Not supported</td>
</tr>
<tr>
<td>xci3gc</td>
<td>Not supported</td>
</tr>
<tr>
<td>xci3ll</td>
<td>Supported</td>
</tr>
<tr>
<td>xci611</td>
<td>Supported</td>
</tr>
</tbody>
</table>

2 This was the MIPSpro C compiler on IRIX; and the Digital C Compiler on OSF/1.
2.23 Porting the MPS

This chapter lists the steps involved in porting the MPS to a new operating system, processor architecture, or compiler. It assumes that you are familiar with Building the Memory Pool System and the Platforms chapter.

2.23.1 Platform code

Pick two-character codes for the new platform’s operating system, processor architecture, and compiler toolchain, as described under Platforms, and concatenate them to get a six-character platform code “osarct”.

2.23.2 Functional modules

The MPS requires platform-specific implementations of the functional modules in the list below. You’ll probably find that it’s unnecessary to port them all: unless the new platform is very exotic, some of the existing implementations ought to be usable. In most cases there is a generic (“ANSI”) implementation of the module, that uses only the features of the Standard C Library. These generic implementations are partially functional or non-functional, but can be used as a starting point for a new port if none of the existing implementations is usable.

1. The lock module provides binary locks that ensure that only a single thread may be running with a lock held, and recursive locks, where the same thread may safely take the lock again without deadlocking.

   See design-lock for the design, and lock.h for the interface. There are implementations for Linux in lockli.c, POSIX in lockix.c, and Windows in lockw3.c.

   There is a generic implementation in lockan.c, which cannot actually take any locks and so only works for a single thread.

2. The memory protection module applies protection to areas of memory, ensuring that attempts to read or write from those areas cause protection faults, and implements the means for the MPS to catch and handle these faults.

   See design-prot for the design, and prot.h for the interface. There are implementations for POSIX in protix.c plus protsgix.c, Linux in protli.c, Windows in protw3.c, and OS X using Mach in protxc.c.

   There is a generic implementation in protan.c, which can’t provide memory protection, so it forces memory to be scanned until that there is no further need to protect it. This means it can’t support incremental collection, and has no control over pause times.

3. The protection mutator context module figures out what the mutator was doing when it caused a protection fault, so that access to a protected region of memory can be handled, or when a thread was suspended, so that its registers and control stack can be scanned.

   See design-prmc for the design, and prot.h for the interface. There are implementations on Unix, Windows, and OS X for IA-32 and x86-64.

   There is a generic implementation in prmcan.c, which can’t provide these features, and so only supports a single thread.
4. The **stack probe** module checks that there is enough space on the **control stack** for the MPS to complete any operation that it might start. The purpose is to provoke a stack overflow exception, if necessary, before taking the arena lock.

See design-sp for the design, and sp.h for the interface. There are implementations on Windows on IA-32 in spi3w3.c and x86-64 in spi6w3.c.

There is a generic implementation in span.c, which can’t provide this feature, and so is only suitable for use with a client program that does not handle stack overflow faults, or does not call into the MPS from the handler.

5. The **stack and register scanning** module **scans** the **registers** and **control stack** of a thread.

See design-ss for the design, and ss.h for the interface. There are implementations for POSIX on IA-32 in ssixi3.c and x86-64 in ssixi6.c, and for Windows with Microsoft Visual C/C++ on IA-32 in ssw3i3mv.c and x86-64 in ssw3i6mv.c.

There is a generic implementation in ssan.c, which calls setjmp() to spill the registers and scans the whole jump buffer, thus overscanning compared to a platform-specific implementation.

6. The **thread manager** module suspends and resumes **threads**, so that the MPS can gain exclusive access to **memory**, and so that it can scan the **registers** and **control stack** of suspended threads.

See design-thread-manager for the design, and th.h for the interface. There are implementations for POSIX in thix.c plus pthrdext.c, OS X using Mach in thxc.c, Windows in thw3.c.

There is a generic implementation in than.c, which necessarily only supports a single thread.

7. The **virtual mapping** module reserves **address space** from the operating system (and returns it), and **maps** address space to **main memory** (and unmaps it).

See design-vm for the design, and vm.h for the interface. There are implementations for POSIX in vmix.c, and Windows in vmw3.c. There is a generic implementation in vman.c, which fakes virtual memory by calling malloc().

### 2.23.3 Platform detection

The new platform must be detected in mpstd.h and preprocessor constants like **MPS_WORD_WIDTH** defined. See design-config for the design of this header, and Platform interface for the list of preprocessor constants that may need to be defined. For example:

```c
/* Predefined Macros" from "Visual Studio 2010" on MSDN
 * Note that Win32 includes 64-bit Windows!
 * We use the same alignment as MS malloc: 16, which is used for XMM
 * operations.
 * See MSDN -> x64 Software Conventions -> Overview of x64 Calling Conventions
 */

#if defined(_MSC_VER) && defined(_WIN32) && defined(_WIN64) && defined(_M_X64) && !defined(__POCC__)
#error "specified CONFIG_PF_... inconsistent with detected w3i6mv"
#endif
#define MPS_PF_W3I6MV
#define MPS_PF_STRING "w3i6mv"
#define MPS_OS_W3
#define MPS_ARCH_I6
#define MPS_BUILD_MV
#define MPS_T_WORD unsigned __int64
#define MPS_T_ULONGEST unsigned __int64
```
#define MPS_WORD_WIDTH 64
#define MPS_WORD_SHIFT 6
#define MPS_PF_ALIGN 16

The comment should justify the platform test (with reference to documentation or to the output of a command like `gcc -E -dM`), and explain any unusual definitions. For example, here we need to explain the choice of 16 bytes for `MPS_PF_ALIGN`, since normally a 64-bit platform requires 8-byte alignment.

## 2.23.4 Platform configuration

The new platform may be configured, if necessary, in `config.h`. See `design-config` for the design of this header. Avoid platform-specific configuration if possible, to reduce the risk of errors being introduced on one platform and not detected when other platforms are tested.

## 2.23.5 Module selection

In `mps.c`, add a section for the new platform. This must test the platform constant `MPS_PF_OSARCT` that is now defined in `mpstd.h`, and then include all the module sources for the platform. For example:

```c
/* Linux on 64-bit Intel with GCC or Clang */
#elif defined(MPS_PF_LII6GC) || defined(MPS_PF_LII6LL)
#include "lockli.c" /* Linux locks */
#include "thix.c" /* Posix threading */
#include "pthrdrx.c" /* Posix thread extensions */
#include "vmix.c" /* Posix virtual memory */
#include "protix.c" /* Posix protection */
#include "protli.c" /* Linux protection */
#include "proti6.c" /* 64-bit Intel mutator context */
#include "prmc6li.c" /* 64-bit Intel for Linux mutator context */
#include "span.c" /* generic stack probe */
#include "ssixi6.c" /* Posix on 64-bit Intel stack scan */
```

## 2.23.6 Makefile

Add a makefile even if you expect to use an integrated development environment (IDE) like Visual Studio or Xcode. Makefiles make it easier to carry out continuous integration and delivery, and are less likely to stop working because of incompatibilities between IDE versions.

On Unix platforms, the makefile must be named `osarct.gmk`, and must define `PFM` to be the platform code, `MPMPF` to be the list of platform modules (the same files included by `mps.c`), and `LIBS` to be the linker options for any libraries required by the test cases. Then it must include the compiler-specific makefile and `comm.gmk`. For example, `l1i61l.gmk` looks like this:

```makefile
PFM = l1i61l
MPMPF = \
    lockli.c \
    prmc6li1.c \
    proti6.c \
    proti6.c \
    proti6.c \
    pthrdcxt.c \
```
If the platform needs specific compilation options, then define `PFMDEFS` accordingly, but avoid this if at all possible. We recommend in *Building the Memory Pool System* that users compile the MPS using a simple command like `cc -c mps.c`, and we suggest that they can improve performance by compiling the MPS and their object format in the same compilation unit. These steps would be more complicated if the MPS required particular compilation options.

On Windows, the makefile must be named `osarct.nmk`, and must define `PFM` to be the platform code, and `MPMPF` to be the list of platform modules (the same files included by `mps.c`) in square brackets. Then it must include the compiler-specific makefile and `comm.nmk`. For example, `w3i6mv.nmk` looks like this:

```plaintext
PFM = w3i6mv

MPMPF = \
    [lockw3] \\
    [mpsiw3] \\
    [prmci6w3] \\
    [proti6] \\
    [protw3] \\
    [spw3i6] \\
    [ssw3i6mv] \\
    [thw3] \\
    [thw3i6] \\
    [vmw3]

!INCLUDE mv.nmk
!INCLUDE comm.nmk
```

### 2.23.7 Porting strategy

Start the port by selecting existing implementations of the functional modules, using the generic implementations where nothing else will do. Then check that the “smoke tests” pass, by running:

```plaintext
make -f osarct.gmk testrun # Unix
nmake /f osarct.nmk testrun # Windows
```

Most or all of the test cases should pass at this point. If you’re using the generic threading implementation, then the multi-threaded test cases are expected to fail. If you’re using the generic lock implementation, then the lock utilization test case `lockut` is expected to fail. If you’re using the generic memory protection implementation, all the tests that rely on incremental collection are expected to fail. See `tool/testcases.txt` for a database of test cases and the configurations in which they are expected to pass.

Now that there is a working system to build on, porting the necessary modules to the new platform can be done incrementally. It’s a good idea to measure the performance as you go along (for example, using the `gcbench` benchmark) to check that the new memory protection module is effective.
2.23.8 Update the documentation

These sections of the manual should be updated to mention the new platform:

- Building the Memory Pool System
- Platforms

In addition, if aspects of the port were especially tricky, then consider writing a design document (see Design) justifying the implementation.

2.23.9 Contribute

Consider contributing the new platform to the MPS. See Contributing to the MPS.

2.24 Deprecated interfaces

This chapter documents the public symbols in the MPS interface that are now deprecated. These symbols may be removed in any future release (see Support policy for details). If you are using one of these symbols, then you should update your code to use the supported interface.

Note
If you are relying on a deprecated interface, and there is no supported alternative, please contact us. It makes a difference if we know that someone is using a feature.

2.24.1 Deprecated in version 1.115

typedef mps_pool_class_t mps_class_t

 Deprecated
The former name for mps_pool_class_t, chosen when pools were the only objects in the MPS that belonged to classes.

size_t mps_mv_free_size (mps_pool_t pool)

 Deprecated
Use the generic function mps_pool_free_size() instead.

Return the total amount of free space in an MV pool.
pool is the MV pool.
Returns the total free space in the pool, in bytes (1).

size_t mps_mv_size (mps_pool_t pool)

 Deprecated
Use the generic function mps_pool_total_size() instead.
Return the total size of an MV pool.

pool is the MV pool.

Returns the total size of the pool, in bytes. This is the sum of allocated space and free space.

```c
size_t mps_mvff_free_size(mps_pool_t pool)
```

Deprecated
Use the generic function `mps_pool_free_size()` instead.

Return the total amount of free space in an MVFF pool.

pool is the MVFF pool.

Returns the total free space in the pool, in bytes.

```c
size_t mps_mvff_size(mps_pool_t pool)
```

Deprecated
Use the generic function `mps_pool_total_size()` instead.

Return the total size of an MVFF pool.

pool is the MVFF pool.

Returns the total size of the pool, in bytes. This is the sum of allocated space and free space.

```c
size_t mps_mvt_free_size(mps_pool_t pool)
```

Deprecated
Use the generic function `mps_pool_free_size()` instead.

Return the total amount of free space in an MVT pool.

pool is the MVT pool.

Returns the total free space in the pool, in bytes.

```c
size_t mps_mvt_size(mps_pool_t pool)
```

Deprecated
Use the generic function `mps_pool_total_size()` instead.

Return the total size of an MVT pool.

pool is the MVT pool.

Returns the total size of the pool, in bytes. This is the sum of allocated space and free space.

### 2.24. Deprecated interfaces

143
2.24.2 Deprecated in version 1.113

**MPS_ARGS_DONE (args)**

Deprecated
Formerly this was used to finalize a list of *keyword arguments* before passing it to a function. It is no longer needed.

2.24.3 Deprecated in version 1.112

*mps_res_t* `mps_arena_create (mps_arena_t *arena_o, mps_arena_class_t arena_class, ...)`

Deprecated
Use `mps_arena_create_k()` instead.

An alternative to `mps_arena_create_k()` that takes its extra arguments using the standard *C* variable argument list mechanism.

When creating an arena of class `mps_arena_class_cl()`, pass the values for the keyword arguments MPS_KEY_ARENA_SIZE and MPS_KEY_ARENA_CL_BASE like this:

```c
mps_res_t mps_arena_create (mps_arena_t *arena_o,
                           mps_arena_class_t mps_arena_class_cl(),
                           size_t arena_size,
                           mps_addr_t cl_base)
```

When creating an arena of class `mps_arena_class_vm()`, pass the value for the keyword argument MPS_KEY_ARENA_SIZE like this:

```c
mps_res_t mps_arena_create (mps_arena_t *arena_o,
                           mps_arena_class_t mps_arena_class_vm(),
                           size_t arena_size)
```

*mps_res_t* `mps_arena_create_v (mps_arena_t *arena_o, mps_arena_class_t arena_class, va_list args)`

Deprecated
Use `mps_arena_create_k()` instead.

An alternative to `mps_arena_create_k()` that takes its extra arguments using the standard *C* variable argument list mechanism.

```c
mps_res_t mps_arena_create (mps_arena_t *pool_o, mps_arena_t arena, mps_pool_class_t pool_class, ...)
```

Deprecated
Use `mps_pool_create_k()` instead.

An alternative to `mps_pool_create_k()` that takes its extra arguments using the standard *C* variable argument list mechanism.
When creating a pool of class `mps_class_amc()` or `mps_class_amcz()`, pass the values for the keyword arguments `MPS_KEY_FORMAT` and `MPS_KEY_CHAIN` like this:

```c
mps_res_t mps_pool_create(mps_pool_t *pool_o, mps_arena_t arena,
                           mps_pool_class_t mps_class_amc(),
                           mps_fmt_t format,
                           mps_chain_t chain)
```

When creating a pool of class `mps_class_ams()`, pass the values for the keyword arguments `MPS_KEY_FORMAT`, `MPS_KEY_CHAIN` and ambiguous flag `MPS_KEY_AMS_SUPPORT_AMBIGUOUS` like this:

```c
mps_res_t mps_pool_create(mps_pool_t *pool_o, mps_arena_t arena,
                           mps_pool_class_t mps_class_ams(),
                           mps_fmt_t format,
                           mps_chain_t chain,
                           mps_bool_t ams_support_ambiguous)
```

When creating a pool of class `mps_class_ams_debug()`, pass the values for the keyword arguments `MPS_KEY_POOL_DEBUG_OPTIONS`, `MPS_KEY_FORMAT`, `MPS_KEY_CHAIN` and `MPS_KEY_AMS_SUPPORT_AMBIGUOUS` like this:

```c
mps_res_t mps_pool_create(mps_pool_t *pool_o, mps_arena_t arena,
                           mps_pool_class_t mps_class_ams_debug(),
                           mps_pool_debug_option_s *pool_debug_options,
                           mps_fmt_t format,
                           mps_chain_t chain,
                           mps_bool_t ams_support_ambiguous)
```

When creating a pool of class `mps_class_awl()`, pass the values for the keyword arguments `MPS_KEY_FORMAT` and `MPS_KEY_AWL_FIND_DEPENDENT` like this:

```c
mps_res_t mps_pool_create(mps_pool_t *pool_o, mps_arena_t arena,
                           mps_pool_class_t mps_class_awl(),
                           mps_fmt_t format,
                           mps_awl_find_dependent_t awl_find_dependent)
```

When creating a pool of class `mps_class_lo()`, pass the value for the keyword argument `MPS_KEY_FORMAT` like this:

```c
mps_res_t mps_pool_create(mps_pool_t *pool_o, mps_arena_t arena,
                           mps_pool_class_t mps_class_lo(),
                           mps_fmt_t format)
```

When creating a pool of class `mps_class_mfs()`, pass the values for the keyword arguments `MPS_KEY_EXTEND_BY` and `MPS_KEY_MFS_UNIT_SIZE` like this:

```c
mps_res_t mps_pool_create(mps_pool_t *pool_o, mps_arena_t arena,
                           mps_pool_class_t mps_class_mfs(),
                           size_t extend_by,
                           size_t unit_size)
```

When creating a pool of class `mps_class_mv()`, pass the values for the keyword arguments `MPS_KEY_EXTEND_BY`, `MPS_KEY_MEAN_SIZE`, and `MPS_KEY_MAX_SIZE` like this:

```c
mps_res_t mps_pool_create(mps_pool_t *pool_o, mps_arena_t arena,
                           mps_pool_class_t mps_class_mv(),
                           size_t extend_by,
                           size_t mean_size,
                           size_t max_size)
```
When creating a pool of class `mps_class_mv_debug()`, pass the values for the keyword arguments `MPS_KEY_POOL_DEBUG_OPTIONS`, `MPS_KEY_EXTEND_BY`, `MPS_KEY_MEAN_SIZE` and `MPS_KEY_MAX_SIZE` like this:

```c
mps_res_t mps_pool_create(mps_pool_t *pool_o, mps_arena_t arena,
    mps_pool_class_t mps_class_mv_debug(),
    mps_pool_debug_option_s *pool_debug_options,
    size_t extend_by,
    size_t mean_size,
    size_t max_size)
```

When creating a pool of class `mps_class_mvff()`, pass the values for the keyword arguments `MPS_KEY_EXTEND_BY`, `MPS_KEY_MEAN_SIZE`, `MPS_KEY_ALIGN`, `MPS_KEY_MVFF_SLOT_HIGH`, `MPS_KEY_MVFF_ARENA_HIGH` and `MPS_KEY_MVFF_FIRST_FIT` like this:

```c
mps_res_t mps_pool_create(mps_pool_t *pool_o, mps_arena_t arena,
    mps_pool_class_t mps_class_mvff(),
    size_t extend_by,
    size_t mean_size,
    mps_align_t align,
    mps_bool_t mvff_slot_high,
    mps_bool_t mvff_arena_high,
    mps_bool_t mvff_first_fit)
```

When creating a pool of class `mps_class_mvff_debug()`, pass the values for the keyword arguments `MPS_KEY_POOL_DEBUG_OPTIONS`, `MPS_KEY_EXTEND_BY`, `MPS_KEY_MEAN_SIZE`, `MPS_KEY_ALIGN`, `MPS_KEY_MVFF_SLOT_HIGH`, `MPS_KEY_MVFF_ARENA_HIGH`, and `MPS_KEY_MVFF_FIRST_FIT` like this:

```c
mps_res_t mps_pool_create(mps_pool_t *pool_o, mps_arena_t arena,
    mps_pool_class_t mps_class_mvff_debug(),
    mps_pool_debug_option_s *pool_debug_options,
    size_t extend_by,
    size_t mean_size,
    mps_align_t align,
    mps_bool_t mvff_slot_high,
    mps_bool_t mvff_arena_high,
    mps_bool_t mvff_first_fit)
```

When creating a pool of class `mps_class_mvt()`, pass the values for the keyword arguments `MPS_KEY_MIN_SIZE`, `MPS_KEY_MEAN_SIZE`, `MPS_KEY_MAX_SIZE`, `MPS_KEY_MVT_RESERVE_DEPTH` and `MPS_KEY_MVT_FRAG_LIMIT` like this:

```c
mps_res_t mps_pool_create(mps_pool_t *pool_o, mps_arena_t arena,
    mps_pool_class_t mps_class_mvt(),
    size_t min_size,
    size_t mean_size,
    size_t max_size,
    mps_word_t mvt_reserve_depth,
    mps_word_t mvt_frag_limit)
```

Note

The `mvt_frag_limit` is a percentage from 0 to 100 inclusive when passed to `mps_pool_create()`, not a double from 0.0 to 1.0 as in `mps_pool_create_k()`.

When creating a pool of class `mps_class_snc()`, pass the value for the keyword argument `MPS_KEY_FORMAT` like this:
mps_res_t mps_pool_create(mps_pool_t *pool_o, mps_arena_t arena,
                       mps_pool_class_t mps_class_snc(),
                       mps_fmt_t format)

mps_res_t mps_pool_create_v(mps_pool_t *pool_o, mps_arena_t arena, mps_pool_class_t pool_class,
                           va_list args)

Deprecated
Use mps_pool_create_k() instead.

An alternative to mps_pool_create_k() that takes its extra arguments using the standard C va_list mechanism. See mps_pool_create() for details of which arguments to pass for the different pool classes.

mps_res_t mps_ap_create(mps_ap_t *ap_o, mps_pool_t pool, ...

Deprecated
Use mps_ap_create_k() instead.

An alternative to mps_ap_create_k() that takes its extra arguments using the standard C variable argument list mechanism.

When creating an allocation point on a pool of class mps_class_ams(), mps_class_ams_debug(), mps_class_awl() or mps_class_snc(), pass the keyword argument MPS_KEY_RANK like this:

mps_res_t mps_ap_create(mps_ap_t *ap_o, mps_pool_t pool,
                        mps_rank_t rank)

mps_res_t mps_ap_create_v(mps_ap_t *ap_o, mps_pool_t pool, va_list args)

Deprecated
Use mps_ap_create_k() instead.

An alternative to mps_ap_create_k() that takes its extra arguments using the standard C va_list mechanism. See mps_ap_create() for details of which arguments to pass for the different pool classes.

mps_fmt_A_s

Deprecated
Use mps_fmt_create_k() instead.

The type of the structure used to create an object format of variant A.

typedef struct mps_fmt_A_s {
    mps_align_t align;
    mps_fmt_scan_t scan;
    mps_fmt_skip_t skip;
    mps_fmt_copy_t copy;
    mps_fmt_fwd_t fwd;
    mps_fmt_isfwd_t isfwd;
    mps_fmt_pad_t pad;
} mps_fmt_A_s;
The fields of this structure correspond to the keyword arguments to `mps_fmt_create_k()`, except for `copy`, which is not used. In older versions of the MPS this was a `copy method` that copied objects belonging to this format.

```c
mps_res_t mps_fmt_create_A(mps_fmt_t *fmt_o, mps_arena_t arena, mps_fmt_A_s *fmt_A)
```

Deprecated
Use `mps_fmt_create_k()` instead.

Create an `object format` based on a description of an object format of variant A.

```c
mps_fmt_B_s
```

Deprecated
Use `mps_fmt_create_k()` instead.

The type of the structure used to create an `object format` of variant B.

```c
typedef struct mps_fmt_B_s {
    mps_align_t align;
    mps_fmt_scan_t scan;
    mps_fmt_skip_t skip;
    mps_fmt_copy_t copy;
    mps_fmt_fwd_t fwd;
    mps_fmt_isfwd_t isfwd;
    mps_fmt_pad_t pad;
    mps_fmt_class_t mps_class;
} mps_fmt_B_s;
```

Variant B is the same as variant A except for the addition of the `mps_class` method. See `mps_fmt_A_s`.

```c
mps_res_t mps_fmt_create_B(mps_fmt_t *fmt_o, mps_arena_t arena, mps_fmt_B_s *fmt_B)
```

Deprecated
Use `mps_fmt_create_k()` instead.

Create an `object format` based on a description of an object format of variant B.

```c
mps_fmt_auto_header_s
```

Deprecated
Use `mps_fmt_create_k()` instead.

The type of the structure used to create an `object format` of variant auto-header.

```c
typedef struct mps_fmt_auto_header_s {
    mps_align_t align;
    mps_fmt_scan_t scan;
    mps_fmt_skip_t skip;
    mps_fmt_fwd_t fwd;
    mps_fmt_isfwd_t isfwd;
} mps_fmt_auto_header_s;
```
mps_fmt_pad_t pad;
size_t mps_headerSize;
} mps_fmt_auto_header_s;

Variant auto-header is the same as variant A except for the removal of the unused copy method, and the addition of the mps_headerSize field. See mps_fmt_A_s.

mps_res_t mps_fmt_create_auto_header(mps_fmt_t *fmt_o, mps_arena_t arena, mps_fmt_auto_header_s *fmt_ah)

Deprecated
Use mps_fmt_create_k() instead.

Create an object format based on a description of an object format of variant auto-header.

mps_fmt_fixed_s

Deprecated
Use mps_fmt_create_k() instead.

The type of the structure used to create an object format of variant fixed.

typedef struct mps_fmt_fixed_s {
  mps_align_t align;
  mps_fmt_scan_t scan;
  mps_fmt_fwd_t fwd;
  mps_fmt_isfwd_t isfwd;
  mps_fmt_pad_t pad;
} mps_fmt_fixed_s;

Variant fixed is the same as variant A except for the removal of the unused copy method, and the lack of a skip method (this is not needed because the objects are fixed in size). See mps_fmt_A_s.

mps_res_t mps_fmt_create_fixed(mps_fmt_t *fmt_o, mps_arena_t arena, mps_fmt_fixed_s *fmt_fixed)

Deprecated
Use mps_fmt_create_k() instead.

Create an object format based on a description of an object format of variant fixed.

2.24.4 Deprecated in version 1.111

mps_res_t mps_fix(mps_ss_t ss, mps_addr_t *ref_io)

Deprecated
Use MPS_FIX1() and MPS_FIX2() instead.

Fix a reference.
This is a function equivalent to:
MPS_SCAN_BEGIN(ss);
res = MPS_FIX12(ss, ref_io);
MPS_SCAN_END(ss);
return res;

Because scanning is an operation on the critical path, we recommend that you use MPS_FIX12() (or MPS_FIX1() and MPS_FIX2()) to ensure that the “stage 1 fix” is inlined.

Note
If you call this between MPS_SCAN_BEGIN() and MPS_SCAN_END(), you must use MPS_FIX_CALL() to ensure that the scan state is passed correctly.

mps_word_t mps_telemetry_control (mps_word_t reset_mask, mps_word_t flip_mask)

Deprecated
Use mps_telemetry_get(), mps_telemetry_reset(), and mps_telemetry_set() instead.

Update and return the telemetry filter.
reset_mask is a bitmask indicating the bits in the telemetry filter that should be reset.
flip_mask is a bitmask indicating the bits in the telemetry filter whose value should be flipped after the resetting.

Returns the previous value of the telemetry filter, prior to the reset and the flip.

The parameters reset_mask and flip_mask allow the specification of any binary operation on the filter control. For typical operations, the parameters should be set as follows:

<table>
<thead>
<tr>
<th>Operation</th>
<th>reset_mask</th>
<th>flip_mask</th>
</tr>
</thead>
<tbody>
<tr>
<td>set (M)</td>
<td>M</td>
<td>M</td>
</tr>
<tr>
<td>reset (M)</td>
<td>M</td>
<td>0</td>
</tr>
<tr>
<td>flip (M)</td>
<td>0</td>
<td>M</td>
</tr>
<tr>
<td>read ()</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

void mps_tramp (void **r_o, mps_tramp_t f, void *p, size_t s)

Deprecated
The MPS trampoline is no longer required on any operating system supported by the MPS.

Call a function via the MPS trampoline.
r_o points to a location that will store the result of calling f.
f is the function to call.
p and s are arguments that will be passed to f each time it is called. This is intended to make it easy to pass, for example, an array and its size as parameters.

The MPS relies on barriers (1) to protect memory that is in an inconsistent state. On some operating systems, barrier hits generate exceptions that have to be caught by a handler that is on the stack. On these operating systems, any code that uses memory managed by the MPS must be called from inside such an exception handler, that is, inside a call to mps_tramp().

If you have multiple threads that run code that uses memory managed by the MPS, each thread must execute such code inside a call to mps_tramp().
void *(\*mps_tramp_t)(void *p, size_t s)

**Deprecated**
The MPS trampoline is no longer required on any operating system supported by the MPS.

The type of a function called by `mps_tramp()`. p and s are the corresponding arguments that were passed to `mps_tramp()`.

void `mps_arena_expose(mps_arena_t arena)`

**Deprecated**
If you need access to protected memory for debugging, contact us.

Ensure that the MPS is not protecting any page in the arena with a read barrier or write barrier. arena is the arena to expose.

This is expected to only be useful for debugging. The arena is left in the clamped state.

Since barriers are used during a collection, calling this function has the same effect as calling `mps_arena_park()`: all collections are run to completion, and the arena is clamped so that no new collections begin. The MPS also uses barriers to maintain remembered sets, so calling this function will effectively destroy the remembered sets and any optimization gains from them.

Calling this function is time-consuming: any active collections will be run to completion; and the next collection will have to recompute all the remembered sets by scanning the entire arena.

The recomputation of the remembered sets can be avoided by calling `mps_arena_unsafe_expose_remember_protection()` instead of `mps_arena_expose()`, and by calling `mps_arena_unsafe_restore_protection()` before calling `mps_arena_release()`. Those functions have unsafe aspects and place restrictions on what the client program can do (basically no exposed data can be changed).

void `mps_arena_unsafe_expose_remember_protection(mps_arena_t arena)`

**Deprecated**
If you need access to protected memory for debugging, contact us.

Ensure that the MPS is not protecting any page in the arena with a read barrier or write barrier. In addition, request the MPS to remember some parts of its internal state so that they can be restored later.

arena is the arena to expose.

This function is the same as `mps_arena_expose()`, but additionally causes the MPS to remember its protection state. The remembered protection state can optionally be restored later by calling the `mps_arena_unsafe_restore_protection()` function. This is an optimization that avoids the MPS having to recompute all the remembered sets by scanning the entire arena.

However, restoring the remembered protections is only safe if the contents of the exposed pages have not been changed; therefore this function should only be used if you do not intend to change the pages, and the remembered protection must only be restored if the pages have not been changed.

The MPS will only remember the protection state if resources (memory) are available. If memory is low then only some or possibly none of the protection state will be remembered, with a corresponding necessity to
recompute it later. The MPS provides no mechanism for the client program to determine whether the MPS has in fact remembered the protection state.

The remembered protection state, if any, is discarded after calling `mps_arena_unsafe_restore_protection()`, or as soon as the arena leaves the clamped state by calling `mps_arena_release()`.

```c
void mps_arena_unsafe_restore_protection (mps_arena_t arena)
```

**Deprecated**

If you need access to protected memory for debugging, contact us.

Restore the remembered protection state for an arena.

arena is the arena to restore the protection state for.

This function restores the protection state that the MPS has remembered when the client program called `mps_arena_unsafe_expose_remember_protection()`. The purpose of remembering and restoring the protection state is to avoid the need for the MPS to recompute all the remembered sets by scanning the entire arena, that occurs when `mps_arena_expose()` is used, and which causes the next garbage collection to be slow.

The client program must not change the exposed data between the call to `mps_arena_unsafe_expose_remember_protection()` and `mps_arena_unsafe_restore_protection()`. If the client program has changed the exposed data then `mps_arena_unsafe_restore_protection()` must not be called: in this case simply call `mps_arena_release()`.

Calling this function does not release the arena from the clamped state: `mps_arena_release()` must be called to continue normal collections.

Calling this function causes the MPS to forget the remembered protection state; as a consequence the same remembered state cannot be restored more than once.
3.1 Choosing a pool class

This section contains a simple procedure for choosing a pool class based on the properties of the data you plan to store in it. The MPS works well if you can segregate your data into a variety of pools, choosing the most appropriate pool class for each.

Note

Pool classes can differ in many ways not considered here: speed, vulnerability to fragmentation, control overhead, and so on. This procedure gives you a decent recommendation, but an expert in the MPS might be able to make a better recommendation. And if no pool class in the open source MPS exactly matches your needs, then it is possible to develop new pool classes. See Writing a new pool class.

First, do you need the MPS to automatically reclaim unreachable blocks? If so, you need an automatically managed (garbage collected) pool class and you should consult Choosing an automatic pool class below. Otherwise, you need a manually managed pool class and you should consult Choosing a manual pool class below.

3.1.1 Choosing an automatic pool class

Answer these questions about your data:

1. Is it acceptable for the MPS to move blocks in memory and to place barriers (1) on blocks? (For example, it might not be acceptable to move a block if it has been passed to foreign code that remembered its location.)

2. Do your blocks contain references to blocks stored in automatically managed pools (including references to other blocks in the same pool, if it’s automatically managed)? And if so, are these references exact or weak?

Second, look up your answers in this table to find the recommended pool class to use:

<table>
<thead>
<tr>
<th>Movable &amp; protectable?</th>
<th>References?</th>
<th>Use this pool class</th>
</tr>
</thead>
<tbody>
<tr>
<td>yes</td>
<td>none</td>
<td>AMCZ (Automatic Mostly-Copying Zero-rank)</td>
</tr>
<tr>
<td>yes</td>
<td>exact</td>
<td>AMC (Automatic Mostly-Copying)</td>
</tr>
<tr>
<td>yes</td>
<td>weak</td>
<td>AWL (Automatic Weak Linked)</td>
</tr>
<tr>
<td>no</td>
<td>none</td>
<td>LO (Leaf Object)</td>
</tr>
<tr>
<td>no</td>
<td>exact</td>
<td>AMS (Automatic Mark and Sweep)</td>
</tr>
<tr>
<td>no</td>
<td>weak</td>
<td>nothing suitable</td>
</tr>
</tbody>
</table>
3.1.2 Choosing a manual pool class

Answer these questions about your data:

1. Are the blocks fixed in size? If so, use *MFS (Manual Fixed Small)*.
2. Are the lifetimes of blocks predictable? If so, use *MVT (Manual Variable Temporal)*, and arrange that objects that are predicted to die at about the same time are allocated from the same *allocation point*.
3. Otherwise, use *MVFF (Manual Variable First Fit)*.

3.2 Pool class properties

This table summarizes the properties of each pool class provided by the open source MPS. For “block” properties, “yes” means that the property holds for all blocks allocated from the pool. An entry “—” indicates that a property makes no sense for a pool class: for example, if blocks in a pool may not contain *references*, it makes no sense to ask whether they may contain *weak references* (1).
## 3.2. Pool class properties

<table>
<thead>
<tr>
<th>Property</th>
<th>AMC</th>
<th>CAM</th>
<th>CAMS</th>
<th>SALLO</th>
<th>MFS</th>
<th>MV</th>
<th>MVFF</th>
<th>MVT</th>
<th>SNC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Supports <code>mps_alloc()</code>?</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Supports <code>mps_free()</code>?</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Supports allocation points?</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Supports allocation frames?</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Supports segregated allocation caches?</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Timing of collections?</td>
<td>auto</td>
<td>auto</td>
<td>auto</td>
<td>auto</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>May contain references?</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>May contain exact references?</td>
<td>yes</td>
<td>—</td>
<td>yes</td>
<td>yes</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>yes</td>
</tr>
<tr>
<td>May contain ambiguous references?</td>
<td>no</td>
<td>—</td>
<td>no</td>
<td>no</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>no</td>
</tr>
<tr>
<td>May contain weak references?</td>
<td>no</td>
<td>—</td>
<td>no</td>
<td>yes</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>no</td>
</tr>
<tr>
<td>Allocations fixed or variable in size?</td>
<td>var</td>
<td>var</td>
<td>var</td>
<td>var</td>
<td>fixed</td>
<td>var</td>
<td>var</td>
<td>var</td>
<td>var</td>
</tr>
<tr>
<td>Alignment?</td>
<td>conf</td>
<td>conf</td>
<td>conf</td>
<td>conf</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>Dependent objects?</td>
<td>no</td>
<td>—</td>
<td>no</td>
<td>yes</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>no</td>
<td>—</td>
</tr>
<tr>
<td>May use remote references?</td>
<td>no</td>
<td>—</td>
<td>no</td>
<td>no</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>no</td>
</tr>
<tr>
<td>Blocks are automatically managed?</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Blocks are promoted between generations</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>Blocks are manually managed?</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Blocks are scanned?</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>Blocks support base pointers only?</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Pools that support internal pointers can be switched to base pointers only, by setting the optional keyword argument <code>MPS_KEY_INTERIOR</code> to <code>FALSE</code> when calling <code>mps_pool_create_k()</code>.</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>yes</td>
</tr>
<tr>
<td>Blocks may be protected by barriers?</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Blocks may move?</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Blocks may be finalized?</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Blocks must be</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>—</td>
</tr>
<tr>
<td>Blocks may use <code>in-band headers</code>?</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>no</td>
</tr>
</tbody>
</table>

### Notes

1. “Timing of collections” is “auto” if garbage collection is under the control of the MPS, which decides when collection should take place and

---

1.55
3.3 Writing a new pool class

If none of the pool classes supplied with the MPS are quite right for your application, don’t despair: the MPS is designed to be extensible with new pool classes, and designed so that the properties of pools are as orthogonal as possible. So if you need a pool containing objects that are scannable but unformatted, or movable objects which are manually managed, or a pool all of whose objects are roots, there is no technical reason why it should not be possible to write it.

If you’d be interested in our developing new pool classes for your requirements, or if you’ve started writing a new pool class yourself, we’d love to hear from you.

3.4 AMC (Automatic Mostly-Copying)

AMC is a general-purpose automatically managed pool class. This is the most mature pool class in the MPS, intended for the majority of objects in the client program. Use this pool class unless you need a particular feature that it doesn’t provide.

“Mostly Copying” means that it uses copying garbage collection except for blocks that are pinned by ambiguous references.

It uses generational garbage collection. That is, it exploits assumptions about object lifetimes and inter-connection variously referred to as “the generational hypothesis”. In particular, the following tendencies will be efficiently exploited by an AMC pool:

- most objects die young;
- objects that don’t die young will live a long time.

3.4.1 AMC properties

- Does not support allocation via mps_alloc() or deallocation via mps_free().
- Supports allocation via allocation points. If an allocation point is created in an AMC pool, the call to mps_ap_create_k() takes no keyword arguments.

<table>
<thead>
<tr>
<th>References</th>
<th>Descriptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>The references in question are references to blocks in automatically managed pools.</td>
</tr>
<tr>
<td>3</td>
<td>Pools “may contain ambiguous/exact/weak references” if the references that the client program fixes during scanning may include references of the indicated rank.</td>
</tr>
<tr>
<td>4</td>
<td>“Alignment” is “conf” if the client program may specify alignment for each pool.</td>
</tr>
<tr>
<td>5</td>
<td>“Alignment” is “conf” if the client program may specify alignment for each pool. The alignment of blocks allocated from MFS (Manual Fixed Small) pools is the platform’s natural alignment, MPS_PF_ALIGN.</td>
</tr>
<tr>
<td>6</td>
<td>MVT (Manual Variable Temporal) and MVFF (Manual Variable First Fit) pools have configurable alignment, but it may not be smaller than sizeof(void *).</td>
</tr>
<tr>
<td>7</td>
<td>In pools with this property, each object may specify an dependent object which the client program guarantees will be accessible during the scanning of the first object. This may be used in the implementation of weak hash tables.</td>
</tr>
<tr>
<td>8</td>
<td>“Remote references” are references that are stored outside the block to which they logically belong (for example, in some kind of auxiliary table). A pool containing remote references cannot rely on a write barrier to detect changed references.</td>
</tr>
<tr>
<td>9</td>
<td>Blocks are “automatically managed” if they may be automatically discarded when the MPS determines that they are unreachable; they are “manually managed” if they can be discarded when the client program requests it. Note that these properties are not mutually exclusive, although the MPS does not provide a pool class that satisfies both.</td>
</tr>
</tbody>
</table>
| 10         | Blocks “are scanned” if the MPS scan them for references; blocks “must be formatted” if they are described to the MPS by an object format. At present, the MPS only knows how to scan blocks using the scan method from an object format, but the MPS design does not preclude pools that scan unformatted blocks.
• Supports *allocation frames* but does not use them to improve the efficiency of stack-like allocation.
• Does not support *segregated allocation caches*.
• Garbage collections are scheduled automatically. See *Scheduling of collections*.
• Uses *generational garbage collection*: blocks are promoted from generation to generation in the pool’s chain.
• Blocks may contain *exact references* to blocks in the same or other pools (but may not contain *ambiguous references* or *weak references* (1), and may not use *remote references*).
• Allocations may be variable in size.
• The *alignment* of blocks is configurable.
• Blocks do not have *dependent objects*.
• Blocks that are not *reachable* from a *root* are automatically *reclaimed*.
• Blocks are *scanned*.
• Blocks may be referenced by *interior pointers* (unless MPS_KEY_INTERIOR is set to FALSE, in which case only *base pointers*, or *client pointers* if the blocks have *in-band headers*, are supported).
• Blocks may be protected by *barriers* (1).
• Blocks may *move*.
• Blocks may be registered for *finalization*.
• Blocks must belong to an *object format* which provides *scan*, *skip*, *forward*, *is-forwarded*, and *padding* methods.
• Blocks may have *in-band headers*.

### 3.4.2 AMC interface

```c
#include "mpscamc.h"

mps_pool_class_t mps_class_amc(void)
Return the pool class for an AMC (Automatic Mostly-Copying) pool.

When creating an AMC pool, mps_pool_create_k() requires one keyword argument:

• MPS_KEY_FORMAT (type mps_fmt_t) specifies the object format for the objects allocated in the pool.
  The format must provide a *scan* method, a *skip* method, a *forward* method, an *is-forwarded* method and a *padding* method.

It accepts three optional keyword arguments:

• MPS_KEY_CHAIN (type mps_chain_t) specifies the generation chain for the pool. If not specified, the pool will use the arena’s default chain.

• MPS_KEY_INTERIOR (type mps_bool_t, default TRUE) specifies whether ambiguous interior pointers to blocks in the pool keep objects alive. If this is FALSE, then only client pointers keep objects alive.

• MPS_KEY_EXTEND_BY (type size_t, default 4096) is the minimum size of the memory segments that the pool requests from the arena. Larger segments reduce the per-segment overhead, but increase fragmentation and retention.

For example:
```
3.4.3 AMC introspection

```c
#include "mpscamc.h"

void mps_amc_apply (mps_pool_t pool, mps_amc_apply_stepper_t f, void *p, size_t s)

Visit all formatted objects in an AMC pool.

pool is the pool whose formatted objects you want to visit.

f is a function that will be called for each formatted object in the pool.

p and s are arguments that will be passed to f each time it is called. This is intended to make it easy to pass, for example, an array and its size as parameters.

It is an error to call this function when the arena is not in the parked state. You need to call mps_arena_collect() or mps_arena_park() before calling mps_amc_apply().

The function f will be called on both client and padding objects. It is the job of f to distinguish, if necessary, between the two. It may also be called on dead objects that the collector has not recycled or has been unable to recycle.

Note
There is no equivalent function for other pool classes, but there is a more general function mps_arena_formatted_objects_walk() that visits all formatted objects in the arena.

Note
This function is intended for heap analysis, tuning, and debugging, not for frequent use in production.
```

```c
void (*mps_amc_apply_stepper_t) (mps_addr_t addr, void *p, size_t s)

The type of a stepper function for formatted objects in an AMC pool.

addr is the address of an object in the pool.

p and s are the corresponding arguments that were passed to mps_amc_apply().

The function may not call any function in the MPS. It may access:

1. memory inside the object or block pointed to by addr;
2. memory managed by the MPS that is in pools that do not protect their contents;
3. memory not managed by the MPS;

It must not access other memory managed by the MPS.
```

3.5 AMCZ (Automatic Mostly-Copying Zero-rank)

AMCZ is a general-purpose automatically managed pool class for leaf objects (“zero-rank” objects that contain no references).

It is otherwise identical to AMC (Automatic Mostly-Copying).

AMCZ is intended for “simple” objects like numbers, characters, and strings. Segregating these objects into one or more AMCZ pools avoids the cost of scanning them that would be incurred if they were interleaved in a pool with objects containing references. It may also simplify the scanning of the objects that are left behind.

See Segregation of objects for an example.
3.5.1 AMCZ properties

AMCZ is identical to AMC (Automatic Mostly-Copying), except that:

- Blocks may not contain references to blocks in automatically managed pools.
- Blocks are not scanned. A consequence of this is that the pool’s object format need not provide a scan method.
- Blocks are not protected by barriers (1).

3.5.2 AMCZ interface

```c
#include "mpscamc.h"

mps_pool_class_t mps_class_amcz() (void)

Return the pool class for an AMCZ (Automatic Mostly-Copying Zero-rank) pool.

When creating an AMCZ pool, mps_pool_create_k() requires one keyword argument:

- MPS_KEY_FORMAT (type mps_fmt_t) specifies the object format for the objects allocated in the pool.
  The format must provide a skip method, a forward method, an is-forwarded method and a padding method.

It accepts two optional keyword arguments:

- MPS_KEY_CHAIN (type mps_chain_t) specifies the generation chain for the pool. If not specified, the pool will use the arena’s default chain.
- MPS_KEY_INTERIOR (type mps_bool_t, default TRUE) specifies whether ambiguous interior pointers to blocks in the pool keep objects alive. If this is FALSE, then only client pointers keep objects alive.

For example:

```c
MPS_ARGS_BEGIN(args) {
    MPS_ARGS_ADD(args, MPS_KEY_FORMAT, fmt);
    res = mps_pool_create_k(&pool, arena, mps_class_amcz(), args);
} MPS_ARGS_END(args);
```

3.5.3 AMCZ introspection

See AMC introspection.

3.6 AMS (Automatic Mark and Sweep)

AMS is an automatically managed but non-moving pool class. It should be used instead of AMC (Automatic Mostly-Copying) for blocks that need to be automatically managed, but cannot be moved.

Note

AMS is likely to be useful as a step in integrating a program with the MPS. It allows you to work on scanning (and investigate errors resulting from underscanning) without having to deal with objects moving as well. When you are confident that scanning is correct, you can switch to AMC (Automatic Mostly-Copying).

AMS is not currently suitable for production use. However, it could be developed into a solid mark-and-sweep pool. If you have a use case that needs this, contact us.
3.6.1 AMS properties

- Does not support allocation via `mps_alloc()` or deallocation via `mps_free()`.
- Supports allocation via *allocation points*. If an allocation point is created in an AMS pool, the call to `mps_ap_create_k()` takes one optional keyword argument, `MPS_KEY_RANK`.
- Supports *allocation frames* but does not use them to improve the efficiency of stack-like allocation.
- Does not support *segregated allocation caches*.
- Garbage collections are scheduled automatically. See *Scheduling of collections*.
- Does not use *generational garbage collection*, so blocks are never promoted out of the generation in which they are allocated.
- Blocks may contain *exact references* to blocks in the same or other pools, or *ambiguous references* (unless the `MPS_KEY_AMS_SUPPORT_AMBIGUOUS` keyword argument is set to `FALSE` when creating the pool). Blocks may not contain *weak references* (1), and may not use *remote references*.
- Allocations may be variable in size.
- The *alignment* of blocks is configurable.
- Blocks do not have *dependent objects*.
- Blocks that are not *reachable* from a *root* are automatically *reclaimed*.
- Blocks are *scanned*.
- Blocks may only be referenced by *base pointers* (unless they have *in-band headers*).
- Blocks are not protected by *barriers* (1).
- Blocks do not *move*.
- Blocks may be registered for *finalization*.
- Blocks must belong to an *object format* which provides *scan* and *skip* methods.
- Blocks may have *in-band headers*.

3.6.2 AMS interface

```c
#include "mpscams.h"

mps_pool_class_t mps_class_ams(void)  
Return the pool class for an AMS (Automatic Mark & Sweep) pool.

When creating an AMS pool, `mps_pool_create_k()` requires one keyword argument:

- `MPS_KEY_FORMAT` (type `mps_fmt_t`) specifies the *object format* for the objects allocated in the pool. The format must provide a *scan method* and a *skip method*.

It accepts three optional keyword arguments:

- `MPS_KEY_CHAIN` (type `mps_chain_t`) specifies the *generation chain* for the pool. If not specified, the pool will use the arena’s default chain.

- `MPS_KEY_GEN` (type `unsigned`) specifies the *generation* in the chain into which new objects will be allocated. If you pass your own chain, then this defaults to 0, but if you didn’t (and so use the arena’s default chain), then an appropriate generation is used.

Note that AWL does not use generational garbage collection, so blocks remain in this generation and are not promoted.
• MPS_KEY_AMS_SUPPORT_AMBIGUOUS (type mps_bool_t, default TRUE) specifies whether references to blocks in the pool may be ambiguous.

For example:

```c
MPS_ARGS_BEGIN(args) {
  MPS_ARGS_ADD(args, MPS_KEY_FORMAT, fmt);
  res = mps_pool_create_k(&pool, arena, mps_class_ams(), args);
} MPS_ARGS_END(args);
```

When creating an allocation point on an AMS pool, mps_ap_create_k() accepts one optional keyword argument:

• MPS_KEY_RANK (type mps_rank_t, default mps_rank_exact()) specifies the rank of references in objects allocated on this allocation point. It must be mps_rank_exact() (if the objects allocated on this allocation point will contain exact references), or mps_rank_ambig() (if the objects may contain ambiguous references).

For example:

```c
MPS_ARGS_BEGIN(args) {
  MPS_ARGS_ADD(args, MPS_KEY_RANK, mps_rank_ambig());
  res = mps_ap_create_k(&ap, ams_pool, args);
} MPS_ARGS_END(args);
```

mps_pool_class_t mps_class_ams_debug(void)

A debugging version of the AMS pool class.

When creating a debugging AMS pool, mps_pool_create_k() accepts the following keyword arguments: MPS_KEY_FORMAT, MPS_KEY_CHAIN, MPS_KEY_GEN, and MPS_KEY_AMS_SUPPORT_AMBIGUOUS are as described above, and MPS_KEY_POOL_DEBUG_OPTIONS specifies the debugging options. See mps_pool_debug_option_s.

### 3.7 AWL (Automatic Weak Linked)

AWL is an automatically managed non-moving pool class that may contain weak references (1).

The purpose of this pool class is to allow the client to implement weak-key, weak-value, and doubly weak hash tables. In a weak-key hash table, the keys are weakly referenced, so their presence in the table will not prevent the key object from being garbage collected. Once the key is no longer reachable, weak references to it may get splatted (that is, replaced with null pointers). Once that has happened, the client program can’t get at the value corresponding to the key any more, so the implementation is free to splat the value slot as well.

AWL allows the implementation to splat the value slot at the same time that the weak key slot is splatted. (Or the other way around for weak-value tables.) See Dependent objects.

See Weak hash tables in the Advanced topics section of the user guide for a detailed example of using this pool class.

Note

AWL is the only pool in the open source MPS that allows its formatted objects to contain weak references. It was designed to support the weak hash tables in Open Dylan, and may be awkward to use for other use cases. If you need more general handling of weak references, contact us.
3.7.1 AWL properties

- Does not support allocation via `mps_alloc()` or deallocation via `mps_free()`.
- Supports allocation via allocation points. If an allocation point is created in an AWL pool, the call to `mps_ap_create_k()` accepts one optional keyword argument, `MPS_KEY_RANK`.
- Supports allocation frames but does not use them to improve the efficiency of stack-like allocation.
- Does not support segregated allocation caches.
- Garbage collections are scheduled automatically. See Scheduling of collections.
- Does not use generational garbage collection, so blocks are never promoted out of the generation in which they are allocated.
- Blocks may contain exact references or weak references (1) to blocks in the same or other pools (but may not contain ambiguous references, and may not use remote references).
- Allocations may be variable in size.
- The alignment of blocks is configurable.
- Blocks may have dependent objects.
- Blocks that are not reachable from a root are automatically reclaimed.
- Blocks are scanned.
- Blocks may only be referenced by base pointers (unless they have in-band headers).
- Blocks may be protected by barriers (1).
- Blocks do not move.
- Blocks may be registered for finalization.
- Blocks must belong to an object format which provides scan and skip methods.
- Blocks may have in-band headers.

3.7.2 Dependent objects

In order to support prompt deletion of values in a weak-key hash table when the key is splatted (and prompt deletion of keys in a weak-value hash table), an AWL pool allows each object to have a dependent object. (This is where the “Linked” in the name of the pool class comes from.)

The dependent object is specified by the `MPS_KEY_AWL_FIND_DEPENDENT` keyword argument to `mps_pool_create_k()` when creating an AWL pool. This is a function of type `mps_awl_find_dependent_t` that takes the address of an object in the pool and returns the address of its dependent object (or a null pointer if there is no corresponding dependent object).

When scanning an object in an AWL pool, the MPS ensures that the dependent object is not protected. This means that the scan method in the pool’s object format can read or write the dependent object.

If an object contains a reference to its dependent object, you should fix that reference, and be aware that if it is a weak reference then it may be splatted when the dependent object dies.

The way you would normally use this feature in a weak hash table would be to put the table’s keys in one object, and its values in another. (This would be necessary in any case, because the MPS does not support a mixture of exact references and weak references (1) in the same object.) The dependent object for the keys objects is the values object, and vice versa (if necessary). The scan method looks out for the splatting of a reference, and when this is detected, it splats the corresponding reference in the dependent object.
For example:

```
obj_t obj_deleted;        /* deleted entry in hash table */

typedef struct weak_array_s {
    struct weak_array_s *dependent;
    size_t length;         /* tagged as "length * 2 + 1" */
    obj_t slot[1];
} weak_array_s, *weak_array_t;

typedef weak_table_s {
    type_s type;           /* TYPE_WEAK_TABLE */
    weak_array_t keys, values;
} weak_table_s, *weak_table_t;

mps_addr_t weak_array_find_dependent(mps_addr_t addr) {
    weak_array_t a = addr;
    return a->dependent;
}

mps_res_t weak_array_scan(mps_ss_t ss, mps_addr_t base, mps_addr_t limit) {
    MPS_SCAN_BEGIN(ss) {
        while (base < limit) {
            mps_addr_t p;
            weak_array_t a = base;
            size_t i, length = a->length >> 1; /* utag */
            p = a->dependent;
            MPS_FIX12(ss, &p);
            a->dependent = p;
            for (i = 0; i < length; ++i) {
                p = a->slot[i];
                if (MPS_FIX1(ss, p)) {
                    mps_res_t res = MPS_FIX2(ss, &p);
                    if (res != MPS_RES_OK) return res;
                    if (p == NULL && a->dependent) {
                        /* key/value was splatted: splat value/key too */
                        a->dependent->slot[i] = obj_deleted;
                        a->slot[i] = obj_deleted;
                    } else {
                        a->slot[i] = p;
                    }
                }
                base += offsetof(weak_array_s, slot) + a->length * sizeof a->slot[0];
            }
        }
    } MPS_SCAN_END(ss);
    return MPS_RES_OK;
}
```

Note

The `length` field of the `weak_array_s` structure contains the value `length * 2 + 1` so that it cannot be mistaken for a pointer. See `Caution` below.
3.7.3 Protection faults

AWL has another special power: it enables better handing of protection faults on weak objects (objects containing weak references (1)).

To explain the benefit we first need to describe the problem. The MPS uses a read barrier to perform incremental garbage collection. When the client program tries to read an object containing weak references (1), the MPS may have protected it so that the MPS can process the object before the client gets to see it.

The problem is that the client program may try to access a weak object at a point in the collection cycle when the MPS cannot yet determine the status of the objects that the weak object refers to. What the MPS does in this situation is assume that all the referenced objects are going to live. This assumption is correct but conservative; it may result in objects that are weakly referenced staying alive for longer than they need to. In the worst case this can result in a very large amount of memory being used by objects that are no longer needed.

In order to combat this problem the MPS sometimes does the following: Instead of processing the entire weak object and unprotecting it, so that the client program can access the object, the MPS may emulate the processor instruction. When this happens, the MPS doesn’t process the entire weak object; it only processes the exact location that was being accessed (typically a single word). It emulates the processor instruction, and it keeps the object protected. This happens invisibly from the client program’s perspective: it’s exactly as if the instruction executed as normal.

Naturally this emulation business is delicate and involves staring at the most badly written parts of low-level processor architecture manuals for days.

Emulation of accesses to protected objects happens when all of the following are true:

1. The object is a weak object allocated in an AWL pool.
2. The MPS is running on Linux/IA-32 or Windows/IA-32. Extending this list to new (reasonable) operating systems should be tolerable (for example, OS X/IA-32). Extending this to new processor architectures requires more work.
3. The processor instruction that is accessing the object is of a suitable simple form. The MPS doesn’t contain an emulator for all possible instructions that might access memory, so currently it only recognizes and emulates a simple MOV from memory to a register or vice-versa.

Contact us if you need emulation of access to weak references for new operating systems, processor architectures, or memory access instructions.

3.7.4 Caution

Because of the instruction emulation described in Protection faults above, AWL places the following restriction on the format of objects allocated in it:

- Each slot in an object must either be a valid word-aligned reference, or else the bottom bits of the word must be non-zero so that it does not look like an aligned pointer.

  “Aligned pointer” means a word whose numeric value (that is, its value when treated as an unsigned integer) is a multiple of the size of a pointer. If you’re using a 64-bit architecture, that means that an aligned pointer is a multiple of 8 and its bottom three bits are zero.

  The bottom line is that references from an object in an AWL pool must be untagged and aligned, and integers must be tagged with a non-zero tag.

Normally one would cope with this restriction by allocating the table metadata in a pool belonging to another pool class, and only allocating the arrays of keys and values in an AWL pool. See the example above.
3.7.5 AWL interface

```
#include "mpscawl.h"

mps_pool_class_t mps_class_awl(void)
Return the pool class for an AWL (Automatic Weak Linked) pool.

When creating an AWL pool, mps_pool_create_k() requires one keyword argument:

• MPS_KEY_FORMAT (type mps_fmt_t) specifies the object format for the objects allocated in the pool.
The format must provide a scan method and a skip method.

It accepts three optional keyword arguments:

• MPS_KEY_AWL_FIND_DEPENDENT (type mps_awl_find_dependent_t) is a function that specifies how to find the dependent object for an object in the pool. This defaults to a function that always returns NULL (meaning that there is no dependent object).
• MPS_KEY_CHAIN (type mps_chain_t) specifies the generation chain for the pool. If not specified, the pool will use the arena’s default chain.
• MPS_KEY_GEN (type unsigned) specifies the generation in the chain into which new objects will be allocated. If you pass your own chain, then this defaults to 0, but if you didn’t (and so use the arena’s default chain), then an appropriate generation is used.

Note that AWL does not use generational garbage collection, so blocks remain in this generation and are not promoted.

For example:
```
 MPS_ARGS_BEGIN(args) {
 MPS_ARGS_ADD(args, MPS_KEY_FORMAT, fmt);
 MPS_ARGS_ADD(args, MPS_KEY_AWL_FIND_DEPENDENT, find_dependent);
 res = mps_pool_create_k(&pool, arena, mps_class_awl(), args);
} MPS_ARGS_END(args);
```

When creating an allocation point on an AWL pool, mps_ap_create_k() accepts one optional keyword argument:

• MPS_KEY_RANK (type mps_rank_t, default mps_rank_exact()) specifies the rank of references in objects allocated on this allocation point. It must be mps_rank_exact() (if the objects allocated on this allocation point will contain exact references), or mps_rank_weak() (if the objects will contain weak references (1)).

For example:
```
 MPS_ARGS_BEGIN(args) {
 MPS_ARGS_ADD(args, MPS_KEY_RANK, mps_rank_weak());
 res = mps_ap_create_k(&ap, awl_pool, args);
} MPS_ARGS_END(args);
```

mps_addr_t (*mps_awl_find_dependent_t)(mps_addr_t addr)
The type of functions that find the dependent object for an object in an AWL pool.

addr is the address of an object in an AWL pool.

Returns the address of the corresponding dependent object, or a null pointer if there is none.

The dependent object need not be in memory managed by the MPS, but if it is, then it must be in a non-moving pool in the same arena as addr.
3.8 LO (Leaf Object)

LO is an automatically managed pool class for leaf objects (objects that contain no references). It does not move or protect its objects.

This pool class is intended for unstructured data that needs to be accessed by foreign code. It’s ideal for allocating a buffer that needs to be passed to an operating system I/O function.

Note

A thread that reads or writes from blocks allocated in this pool need not be registered with the arena so long as the liveness of the block is independent of that thread.

This means that you can launch a thread to read or write a buffer allocated in this pool, without having to register the thread, so long as you ensure that the buffer remains alive until the thread has finished (for example, by keeping a reference to the buffer in a root or a scanned object).

If LO is used to allocate large numbers of small objects, the garbage collection performance will degrade. For leaf objects that can move and be protected, it is better to use AMCZ (Automatic Mostly-Copying Zero-rank) instead.

3.8.1 LO properties

- Does not support allocation via mps_alloc() or deallocation via mps_free().
- Supports allocation via allocation points. If an allocation point is created in a LO pool, the call to mps_ap_create_k() takes no keyword arguments.
- Supports allocation frames but does not use them to improve the efficiency of stack-like allocation.
- Does not support segregated allocation caches.
- Garbage collections are scheduled automatically. See Scheduling of collections.
- Does not use generational garbage collection, so blocks are never promoted out of the generation in which they are allocated.
- Blocks may not contain references to blocks in automatically managed pools.
- Allocations may be variable in size.
- The alignment of blocks is configurable.
- Blocks do not have dependent objects.
- Blocks that are not reachable from a root are automatically reclaimed.
- Blocks are not scanned. A consequence of this is that the pool’s object format need not provide a scan method.
- Blocks may only be referenced by base pointers (unless they have in-band headers).
- Blocks are not protected by barriers (1).
- Blocks do not move.
- Blocks may be registered for finalization.
- Blocks must belong to an object format which provides scan and skip methods.
- Blocks may have in-band headers.
3.8.2 LO interface

```c
#include "mpsclo.h"

mps_pool_class_t mps_class_lo(void)

Return the pool class for an LO (Leaf Object) pool.

When creating an LO pool, `mps_pool_create_k()` requires one keyword argument:

• MPS_KEY_FORMAT (type `mps_fmt_t`) specifies the object format for the objects allocated in the pool.
  The format must provide a skip method.

It accepts two optional keyword arguments:

• MPS_KEY_CHAIN (type `mps_chain_t`) specifies the generation chain for the pool. If not specified, the pool will use the arena’s default chain.

• MPS_KEY_GEN (type unsigned) specifies the generation in the chain into which new objects will be allocated. If you pass your own chain, then this defaults to 0, but if you didn’t (and so use the arena’s default chain), then an appropriate generation is used.

Note that LO does not use generational garbage collection, so blocks remain in this generation and are not promoted.

For example:

```c
MPS_ARGS_BEGIN(args) {
  MPS_ARGS_ADD(args, MPS_KEY_FORMAT, fmt);
  res = mps_pool_create_k(&pool, arena, mps_class_lo(), args);
} MPS_ARGS_END(args);
```

3.9 MFS (Manual Fixed Small)

MFS is an manually managed pool class for small objects of fixed size.

Unlike other manual pool classes, it is not subject to internal fragmentation: if the population remains bounded, the memory usage remains bounded too. On the other hand, unlike MVT (Manual Variable Temporal) and MVFF (Manual Variable First Fit) it does not return unused memory to the arena for reuse by other pools.

The implementation is very simple: unlike most other pool classes which store their control structures separately from the allocated blocks, MFS maintains a stack of free blocks using a pointer in the free block. `mps_alloc()` pops this stack and `mps_free()` pushes it.

3.9.1 MFS properties

- Supports allocation via `mps_alloc()` and deallocation via `mps_free()`.
- Does not support allocation via `allocation points`.
- Does not support `allocation frames`.
- Supports segregated allocation caches (but using one would be pointless, since all blocks are the same size).
- There are no garbage collections in this pool.
- Blocks may not contain references to blocks in automatically managed pools (unless these are registered as roots).
- Allocations are fixed in size.
• The alignment of blocks is not configurable: it is the natural alignment of the platform (see MPS_PF_ALIGN).
• Blocks do not have dependent objects.
• Blocks are not automatically reclaimed.
• Blocks are not scanned.
• Blocks are not protected by barriers (1).
• Blocks do not move.
• Blocks may not be registered for finalization.
• Blocks must not belong to an object format.

3.9.2 MFS interface

#include "mpscmfs.h"

mps_pool_class_t mps_class_mfs(void)
Return the pool class for an MFS (Manual Fixed Small) pool.

When creating an MFS pool, mps_pool_create_k() requires one keyword argument:

• MPS_KEY_MFS_UNIT_SIZE (type size_t) is the size of blocks that will be allocated from this pool, in bytes (1). It must be at least one word.

In addition, mps_pool_create_k() accepts one optional keyword argument:

• MPS_KEY_EXTEND_BY (type size_t, default 65536) is the size of block that the pool will request from the arena. It must be at least as big as the unit size specified by the MPS_KEY_MFS_UNIT_SIZE keyword argument. If this is not a multiple of the unit size, there will be wasted space in each block.

For example:

MPS_ARGS_BEGIN(args) {
    MPS_ARGS_ADD(args, MPS_KEY_MFS_UNIT_SIZE, 1024);
    MPS_ARGS_ADD(args, MPS_KEY_EXTEND_BY, 1024 * 1024);
    res = mps_pool_create_k(&pool, arena, mps_class_mfs(), args);
} MPS_ARGS_END(args);

3.10 MV (Manual Variable)

MV is a general-purpose manually managed pool class that manages blocks of variable size.

3.10.1 MV properties

• Supports allocation via mps_alloc() and deallocation via mps_free().
• Does not support allocation via allocation points.
• Does not support allocation frames.
• Supports segregated allocation caches.
• There are no garbage collections in this pool.
• Blocks may not contain references to blocks in automatically managed pools (unless these are registered as roots).
• Allocations may be variable in size.
• The alignment of blocks is configurable.
• Blocks do not have dependent objects.
• Blocks are not automatically reclaimed.
• Blocks are not scanned.
• Blocks are not protected by barriers (1).
• Blocks do not move.
• Blocks may not be registered for finalization.
• Blocks must not belong to an object format.

3.10.2 MV interface

```c
#include "mpscmv.h"

mps_pool_class_t mps_class_mv(void)
Return the pool class for an MV (Manual Variable) pool.

When creating an MV pool, mps_pool_create_k() takes four optional keyword arguments:

- MPS_KEY_ALIGN (type mps_align_t, default is MPS_PF_ALIGN) is the alignment of addresses for allocation (and freeing) in the pool. If an unaligned size is passed to mps_alloc() or mps_free(), it will be rounded up to the pool’s alignment.
- MPS_KEY_EXTEND_BY (type size_t, default 65536) is the size of block that the pool will request from the arena.
- MPS_KEY_MEAN_SIZE (type size_t, default 32) is the predicted mean size of blocks that will be allocated from the pool. This value must be smaller than, or equal to, the value for MPS_KEY_EXTEND_BY.
- MPS_KEY_MAX_SIZE (type size_t, default 65536) is the predicted maximum size of blocks that will be allocated from the pool. This value must be larger than, or equal to, the value for MPS_KEY_EXTEND_BY.

The mean and maximum sizes are hints to the MPS: the pool will be less efficient if these are wrong, but nothing will break.

For example:
```c
MPS_ARGS_BEGIN(args) {
    MPS_ARGS_ADD(args, MPS_KEY_MEAN_SIZE, 32);
    MPS_ARGS_ADD(args, MPS_KEY_MAX_SIZE, 1024);
    MPS_ARGS_ADD(args, MPS_KEY_EXTEND_BY, 1024 * 1024);
    res = mps_pool_create_k(&pool, arena, mps_class_mfs(), args);
} MPS_ARGS_END(args);
```

mps_pool_class_t mps_class_mv_debug(void)
A debugging version of the MV pool class.

When creating a debugging MV pool, mps_pool_create_k() takes five optional keyword arguments: MPS_KEY_ALIGN, MPS_KEY_EXTEND_SIZE, MPS_KEY_MEAN_SIZE, MPS_KEY_MAX_SIZE are as described above, and MPS_KEY_POOL_DEBUG_OPTIONS specifies the debugging options. See mps_pool_debug_option_s.
3.11 MVFF (Manual Variable First Fit)

MVFF manually manages variable-sized, unformatted objects. It uses the first fit allocation policy for blocks allocated via `mps_alloc()`.

Johnstone (1997) found that in his test cases:

No version of best fit had more than 5% actual fragmentation. This is also true for all versions of first fit that used an address-ordered free list, and the two versions of first fit that used a FIFO free list. This strongly suggests that the basic best-fit algorithm and the first-fit algorithm with an address-ordered free list are very robust algorithms.

The MVFF pool class also supports buffered allocation (that is, allocation via allocation points), and in this case, the allocation policy is different: the buffers are filled according to the worst fit policy, and allocation always proceeds upwards from the base.

Buffered and unbuffered allocation can be used at the same time, but the first allocation point must be created before any call to `mps_alloc()`.

It is usually not advisable to use buffered and unbuffered allocation on the same pool, because the worst-fit policy of buffer filling will grab all the large blocks, leading to severe fragmentation. If you need both forms of allocation, use two separate pools.

3.11.1 MVFF properties

- Supports allocation via `mps_alloc()`.
- Supports allocation via allocation points. If an allocation point is created in an MVFF pool, the call to `mps_ap_create_k()` takes no keyword arguments.
- Supports deallocation via `mps_free()`.
- Supports allocation frames but does not use them to improve the efficiency of stack-like allocation.
- Supports segregated allocation caches.
- There are no garbage collections in this pool.
- Blocks may not contain references to blocks in automatically managed pools (unless these are registered as roots).
- Allocations may be variable in size.
- The alignment of blocks is configurable, but may not be smaller than `sizeof(void *)`.
- Blocks do not have dependent objects.
- Blocks are not automatically scanned.
- Blocks are not protected by barriers (1).
- Blocks do not move.
- Blocks may not be registered for finalization.
- Blocks must not belong to an object format.
3.11.2 MVFF interface

```
#include "mpscmvff.h"

mps_pool_class_t mps_class_mvff (void)
Return the pool class for an MVFF (Manual Variable First Fit) pool.

When creating an MVFF pool, mps_pool_create_k() accepts seven optional keyword arguments:

• MPS_KEY_EXTEND_BY (type size_t, default 65536) is the size of block that the pool will request from the arena.

• MPS_KEY_MEAN_SIZE (type size_t, default 32) is the predicted mean size of blocks that will be allocated from the pool. This is a hint to the MPS: the pool will be less efficient if this is wrong, but nothing will break.

• MPS_KEY_ALIGN (type mps_align_t, default is MPS_PF_ALIGN) is the alignment of addresses for allocation (and freeing) in the pool. If an unaligned size is passed to mps_alloc() or mps_free(), it will be rounded up to the pool’s alignment. The minimum alignment supported by pools of this class is sizeof(void *).

• MPS_KEY_SPARE (type double, default 0.75) is the maximum proportion of memory that the pool will keep spare for future allocations. If the proportion of memory that’s free exceeds this, then the pool will return some of it to the arena for use by other pools.

• MPS_KEY_MVFF_ARENA_HIGH (type mps_bool_t, default false) determines whether new blocks are acquired at high addresses (if true), or at low addresses (if false).

• MPS_KEY_MVFF_SLOT_HIGH 12 (type mps_bool_t, default false) determines whether to search for the highest addressed free area (if true) or lowest (if false) when allocating using mps_alloc().

• MPS_KEY_MVFF_FIRST_FIT 1 (type mps_bool_t, default true) determines whether to allocate from the highest address in a found free area (if true) or lowest (if false) when allocating using mps_alloc().

The defaults yield a simple first-fit allocator. Specify MPS_KEY_MVFF_ARENA_HIGH and MPS_KEY_MVFF_SLOT_HIGH true, and MPS_KEY_MVFF_FIRST_FIT false to get a first-fit allocator that works from the top of memory downwards. Other combinations may be useful in special circumstances.

For example:
```
MPS_ARGS_BEGIN(args) {
  MPS_ARGS_ADD(args, MPS_KEY_EXTEND_BY, 1024 * 1024);
  MPS_ARGS_ADD(args, MPS_KEY_MEAN_SIZE, 32);
  MPS_ARGS_ADD(args, MPS_KEY_ALIGN, 8);
  MPS_ARGS_ADD(args, MPS_KEY_MVFF_ARENA_HIGH, 1);
  MPS_ARGS_ADD(args, MPS_KEY_MVFF_SLOT_HIGH, 1);
  MPS_ARGS_ADD(args, MPS_KEY_MVFF_FIRST_FIT, 0);
  res = mps_pool_create_k(&pool, arena, mps_class_mvff(), args);
}
```

mps_pool_class_t mps_class_mvff_debug (void)
A debugging version of the MVFF pool class.

When creating a debugging MVFF pool, mps_pool_create_k() accepts eight optional keyword arguments: MPS_KEY_EXTEND_BY, MPS_KEY_MEAN_SIZE, MPS_KEY_ALIGN, MPS_KEY_SPARE, MPS_KEY_MVFF_ARENA_HIGH, MPS_KEY_MVFF_SLOT_HIGH, and MPS_KEY_MVFF_FIRST_FIT are as described above, and MPS_KEY_POOL_DEBUG_OPTIONS specifies the debugging options. See mps_pool_debug_option_s.

12 Allocation points are not affected by MPS_KEY_MVFF_SLOT_HIGH or MPS_KEY_MVFF_FIRST_FIT. They use a worst-fit policy in order to maximise the number of in-line allocations.
3.12 MVT (Manual Variable Temporal)

MVT manually manages variable-sized, unformatted objects. It uses the temporal fit allocation policy.

3.12.1 Temporal fit

Temporal fit attempts to place consecutive allocations next to each other. It relies on delaying re-use as long as possible to permit freed blocks to coalesce, thus maximizing the number of consecutive allocations that can be co-located. Temporal fit permits a very fast allocator and a deallocator competitive in speed with all other known policies.

Temporal fit is intended to take advantage of knowledge of object lifetimes: either a priori knowledge, or knowledge acquired by profiling. The best performance will be achieved by allocating objects with similar expected death times together.

A simple policy can be implemented to take advantage of MVT. Object size is typically well-correlated with object life-expectancy, and birth time plus lifetime gives death time, so allocating objects of similar size sequentially from the same pool instance should result in objects allocated close to each other dying at about the same time.

An application that has several classes of objects of widely differing life expectancy will best be served by creating a different MVT pool instance for each life-expectancy class. A more sophisticated policy can use either the programmer’s knowledge of the expected lifetime of an object, or any characteristic of objects that correlates with lifetime, to choose an appropriate pool to allocate in.

Allocating objects with unknown or very different death times together will pessimize the space performance of MVT.

3.12.2 MVT properties

- Does not support allocation via mps Alloc() .
- Supports allocation via allocation points only. If an allocation point is created in an MVT pool, the call to mps_ap_create_k() takes no keyword arguments.
- Supports deallocation via mps_free().
- Supports allocation frames but does not use them to improve the efficiency of stack-like allocation.
- Does not support segregated allocation caches.
- There are no garbage collections in this pool.
- Blocks may not contain references to blocks in automatically managed pools (unless these are registered as roots).
- Allocations may be variable in size.
- The alignment of blocks is configurable, but may not be smaller than sizeof(void *).
- Blocks do not have dependent objects.
- Blocks are not automatically reclaimed.
- Blocks are not scanned.
- Blocks are not protected by barriers (1).
- Blocks do not move.
- Blocks may not be registered for finalization.
- Blocks must not belong to an object format.
3.12.3 MVT interface

```c
#include "mpscmvt.h"

mps_pool_class_t mps_class_mvt (void)
Return the pool class for an MVT (Manual Variable Temporal) pool.

When creating an MVT pool, mps_pool_create_k() accepts six optional keyword arguments:

• MPS_KEY_ALIGN (type mps_align_t, default is MPS_PF_ALIGN) is the alignment of addresses for allocation (and freeing) in the pool. If an unaligned size is passed to mps_alloc() or mps_free(), it will be rounded up to the pool's alignment. The minimum alignment supported by pools of this class is sizeof(void *).

• MPS_KEY_MIN_SIZE (type size_t, default is MPS_PF_ALIGN) is the predicted minimum size of blocks that will be allocated from the pool.

• MPS_KEY_MEAN_SIZE (type size_t, default 32) is the predicted mean size of blocks that will be allocated from the pool.

• MPS_KEY_MAX_SIZE (type size_t, default 8192) is the predicted maximum size of blocks that will be allocated from the pool. Partial freeing is not supported for blocks larger than this; doing so will result in the storage of the block never being reused.

The three SIZE arguments above are hints to the MPS: the pool will be less efficient if they are wrong, but the only thing that will break is the partial freeing of large blocks.

• MPS_KEY_MVT_RESERVE_DEPTH (type mps_word_t, default 1024) is the expected hysteresis of the population of the pool. When blocks are freed, the pool will retain sufficient storage to allocate this many blocks of the mean size for near term allocations (rather than immediately making that storage available to other pools).

If a pool has a stable population, or one which only grows over the lifetime of the pool, or one which grows steadily and then shrinks steadily, use a reserve depth of 0.

It is always safe to use a reserve depth of 0, but if the population typically fluctuates in a range (for example, the client program repeatedly creates and destroys a subset of blocks in a loop), it is more efficient for the pool to retain enough storage to satisfy that fluctuation. For example, if a pool has an object population that typically fluctuates between 8,000 and 10,000, use a reserve depth of 2,000.

The reserve will not normally be available to other pools for allocation, even when it is not used by the pool. If this is undesirable, a reserve depth of 0 may be used for a pool whose object population does vary, at a slight cost in efficiency. The reserve does not guarantee any particular amount of allocation.

• MPS_KEY_MVT_FRAG_LIMIT (type double, default 0.3) is a double from 0.0 to 1.0 (inclusive). It sets an upper limit on the space overhead of an MVT pool, in case block death times and allocations do not correlate well. If the free space managed by the pool as a ratio of all the space managed by the pool exceeds the fragmentation limit, the pool falls back to a first fit allocation policy, exploiting space more efficiently at a cost in time efficiency. A fragmentation limit of 0.0 would cause the pool to operate as a first-fit pool, at a significant cost in time efficiency: therefore this is not permitted.

A fragmentation limit of 1.0 causes the pool to always use temporal fit (unless resources are exhausted). If the objects allocated in the pool have similar lifetime expectancies, this mode will have the best time- and space-efficiency. If the objects have widely varying lifetime expectancies, this mode will be time-efficient, but may be space-inefficient. An intermediate setting can be used to limit the space-inefficiency of temporal fit due to varying object life expectancies.

For example:
MPS_ARGS_BEGIN(args) {
    MPS_ARGS_ADD(args, MPS_KEY_MIN_SIZE, 4);
    MPS_ARGS_ADD(args, MPS_KEY_MEAN_SIZE, 32);
    MPS_ARGS_ADD(args, MPS_KEY_MAX_SIZE, 1024);
    MPS_ARGS_ADD(args, MPS_KEY_MVT_RESERVE_DEPTH, 256);
    MPS_ARGS_ADD(args, MPS_KEY_MVT_FRAG_LIMIT, 0.5);
    res = mps_pool_create_k(&pool, arena, mps_class_mvt(), args);
} MPS_ARGS_END(args);

3.13 SNC (Stack No Checking)

Deprecated
starting with version 1.111.
If you need special handling of stack-like allocation, contact us.

SNC is a manually managed pool class that supports a stack-like protocol for allocation and deallocation using allocation frames on allocation points. See Allocation frames.

If mps_ap_frame_pop() is used on an allocation point in an SNC pool (after a corresponding call to mps_ap_frame_push()), then the objects affected by the pop are effectively declared dead, and may be reclaimed by the collector. Extant references to such objects from reachable or de facto alive objects are safe, but such other objects should be dead; that is, such references must never be used.

3.13.1 SNC properties

• Does not support allocation via mps_alloc().
• Supports allocation via allocation points only. If an allocation point is created in an SNC pool, the call to mps_ap_create_k() accepts one optional keyword argument, MPS_KEY_RANK.
• Does not support deallocation via mps_free().
• Supports allocation frames.
• Does not support segregated allocation caches.
• Blocks may contain exact references to blocks in the same or other pools (but may not contain ambiguous references or weak references (1), and may not use remote references).
• There are no garbage collections in this pool.
• Allocations may be variable in size.
• The alignment of blocks is configurable.
• Blocks do not have dependent objects.
• Blocks are not automatically reclaimed.
• Blocks are scanned.
• Blocks may only be referenced by base pointers.
• Blocks are not protected by barriers (1).
• Blocks do not move.
• Blocks may not be registered for finalization.
• Blocks must belong to an object format which provides scan, skip, and padding methods.
• Blocks must not have in-band headers.

3.13.2 SNC interface

```c
#include "mpscsnc.h"

mps_pool_class_t mps_class_snc(void)
    Return the pool class for an SNC (Stack No Check) pool.

    When creating an SNC pool, mps_pool_create_k() requires one keyword argument:
        • MPS_KEY_FORMAT (type mps_fmt_t) specifies the object format for the objects allocated in the pool.
          The format must provide a scan method, a skip method, and a padding method.
    
    For example:

    MPS_ARGS_BEGIN(args) {
        MPS_ARGS_ADD(args, MPS_KEY_FORMAT, fmt);
        res = mps_pool_create_k(&pool, arena, mps_class_snc(), args);
    } MPS_ARGS_END(args);
```

When creating an allocation point on an SNC pool, mps_ap_create_k() accepts one optional keyword argument:

- MPS_KEY_RANK (type mps_rank_t, default mps_rank_exact()) specifies the rank of references in objects allocated on this allocation point.

For example:

```c
MPS_ARGS_BEGIN(args) {
    MPS_ARGS_ADD(args, MPS_KEY_RANK, mps_rank_exact());
    res = mps_ap_create_k(&ap, awl_pool, args);
} MPS_ARGS_END(args);```
CHAPTER 4

Design
Old design

**Warning:** Much of the documentation in this section is very old: some of it dates back to the origin of the MPS in 1995. It has not been brought up to date or checked for correctness, so it is mainly of historical interest. As pieces of documentation are brought up to date, they will be moved to the main *Design* section.
Appendices

6.1 Bibliography


  **Abstract**
  Exact garbage collection for the strongly-typed Java language may seem straightforward. Unfortunately, a single pair of bytecodes in the Java Virtual Machine instruction set presents an obstacle that has thus far not been discussed in the literature. We explain the problem, outline the space of possible solutions, and present a solution utilizing bytecode-preprocessing to enable exact garbage collection while maintaining compatibility with existing compiled Java class files.


  **Abstract**
  Full precision in garbage collection implies retaining only those heap allocated objects that will actually be used in the future. Since full precision is not computable in general, garbage collectors use safe (i.e., conservative) approximations such as reachability from a set of root references. Ambiguous roots collectors (commonly called “conservative”) can be overly conservative because they overestimate the root set, and thereby retain unexpectedly large amounts of garbage. We consider two more precise collection schemes for Java virtual machines (JVMs). One uses a type analysis to obtain a type-precise root set (only those variables that contain references); the other adds a live variable analysis to reduce the root set to only the live reference variables. Even with the Java programming language’s strong typing, it turns out that the JVM specification has a feature that makes type-precise root sets difficult to compute. We explain the problem and ways in which it can be solved.

  Our experimental results include measurements of the costs of the type and liveness analyses at load time, of the incremental benefits at run time of the liveness analysis over the type-analysis alone, and of various map sixes and counts. We find that the liveness analysis often produces little or no improvement in heap size, sometimes modest improvements, and occasionally the improvement is dramatic. While further study is in order, we conclude that the main benefit of the liveness analysis is preventing bad surprises.


  **Abstract**
We’ve designed and implemented a copying garbage-collection algorithm that is efficient, real-time, concurrent, runs on commercial uniprocessors and shared-memory multiprocessors, and requires no change to compilers. The algorithm uses standard virtual-memory hardware to detect references to “from space” objects and to synchronize the collector and mutator threads. We’ve implemented and measured a prototype running on SRC’s 5-processor Firefly. It will be straightforward to merge our techniques with generational collection. An incremental, non-concurrent version could be implemented easily on many versions of Unix.


**Abstract**

*Inside Macintosh: Memory* describes the parts of the Macintosh® Operating System that allow you to directly allocate, release, or otherwise manipulate memory. Everyone who programs Macintosh computers should read this book.

*Inside Macintosh: Memory* shows in detail how your application can manage the memory partition it is allocated and perform other memory-related operations. It also provides a complete technical reference for the Memory Manager, the Virtual Memory Manager, and other memory-related utilities provided by the system software.


**Abstract**

Memory management is a critical issue for many large object-oriented applications, but in C++ only explicit memory reclamation through the delete operator is generally available. We analyse different possibilities for memory management in C++ and present a dynamic memory management framework which can be customised to the need of specific applications. The framework allows full integration and coexistence of different memory management techniques. The Customisable Memory Management (CMM) is based on a primary collector which exploits an evolution of Bartlett’s mostly copying garbage collector. Specialised collectors can be built for separate memory heaps. A Heap class encapsulates the allocation strategy for each heap. We show how to emulate different garbage collection styles or user-specific memory management techniques. The CMM is implemented in C++ without any special support in the language or the compiler. The techniques used in the CMM are general enough to be applicable also to other languages.


**Abstract**

Automatic garbage collection relieves programmers from the burden of managing memory themselves and several techniques have been developed that make garbage collection feasible in many situations, including real time applications or within traditional programming languages. However optimal performance cannot always be achieved by a uniform general purpose solution. Sometimes an algorithm exhibits a predictable pattern of memory usage that could be better handled specifically, delaying as much as possible the intervention of the general purpose collector. This leads to the requirement for algorithm specific customisation of the collector strategies. We present a dynamic memory management framework which can be customised to the needs of an algorithm, while preserving the convenience of automatic collection in the normal case. The Customisable Memory Manager (CMM) organises memory in multiple heaps. Each heap is an instance of a C++ class which abstracts and encapsulates a particular storage discipline. The default heap for collectable objects uses the technique of mostly copying garbage collection, providing good performance and memory compaction. Customisation of the collector is achieved exploiting object orientation by defining specialised versions of the collector methods for each heap class. The object oriented interface to the collector enables coexistence and coordination among the various collectors as well as integration with traditional code unaware of garbage collection. The CMM is implemented in C++ without any special support in the language or the compiler. The techniques used in the CMM are general enough to be applicable also to other languages. The performance of
the CMM is analysed and compared to other conservative collectors for C/C++ in various configurations.


Abstract

We consider the combination of card marking with remembered sets for generational garbage collection as suggested by Hosking and Moss. When more than two generations are used, a naive implementation may cause excessive and wasteful scanning of the cards and thus increase the collection time. We offer a simple data structure and a corresponding algorithm to keep track of which cards need be scanned for which generation. We then extend these ideas for the Train Algorithm of Hudson and Moss. Here, the solution is more involved, and allows tracking of which card should be scanned for which car-collection in the train.


Abstract

This paper investigates some problems associated with an argument evaluation order that we call “future” order, which is different from both call-by-name and call-by-value. In call-by-future, each formal parameter of a function is bound to a separate process (called a “future”) dedicated to the evaluation of the corresponding argument. This mechanism allows the fully parallel evaluation of arguments to a function, and has been shown to augment the expressive power of a language.

We discuss an approach to a problem that arises in this context: futures which were thought to be relevant when they were created become irrelevant through being ignored in the body of the expression where they were bound. The problem of irrelevant processes also appears in multiprocessing problem-solving systems which start several processors working on the same problem but with different methods, and return with the solution which finishes first. This “parallel method strategy” has the drawback that the processes which are investigating the losing methods must be identified, stopped, and reassigned to more useful tasks.

The solution we propose is that of garbage collection. We propose that the goal structure of the solution plan be explicitly represented in memory as part of the graph memory (like Lisp’s heap) so that a garbage collection algorithm can discover which processes are performing useful work, and which can be recycled for a new task. An incremental algorithm for the unified garbage collection of storage and processes is described.


Abstract

A real-time list processing system is one in which the time required by the elementary list operations (e.g. CONS, CAR, CDR, RPLACA, RPLACD, EQ, and ATOM in LISP) is bounded by a (small) constant. Classical implementations of list processing systems lack this property because allocating a list cell from the heap may cause a garbage collection, which process requires time proportional to the heap size to finish. A real-time list processing system is presented which continuously reclaims garbage, including directed cycles, while linearizing and compacting the accessible cells into contiguous locations to avoid fragmenting the free storage pool. The program is small and requires no time-sharing interrupts, making it suitable for microcode. Finally, the system requires the same average time, and not more than twice the space, of a classical implementation, and those space requirements can be reduced to approximately classical proportions by compact list representation. Arrays of different sizes, a program stack, and hash linking are simple extensions to our system, and reference counting is found to be inferior for many applications.

Abstract
MACLISP, unlike some other implementations of LISP, allocates storage for different types of objects in non-
contiguous areas called “spaces”. These spaces partition the active storage into disjoint areas, each of which
holds a different type of object. For example, “list cells” are stored in one space, “full-word integers” reside in
another space, “full-word floating point numbers” in another, and so on.

Allocating space in this manner has several advantages. An object’s type can easily be computed from a pointer
to it, without any memory references to the object itself. Thus, the LISP primitive ATOM(x) can easily compute
its result without even paging in x. Another advantage is that the type of an object does not require any storage
within the object, so that arithmetic with hardware data types such as full-word integers can use hardware
instructions directly.

There are problems associated with this method of storage and type management, however. When all data types
are allocated from the same heap, there is no problem with varying demand for the different data types; all data
types require storage from the same pool, so that only the total amount of storage is important. Once different
data types must be allocated from different spaces, however, the relative sizes of the spaces becomes important.

  Collection.

Abstract
Garbage collectors must minimize the scarce resources of cache space and off-chip communications bandwidth
to optimize performance on modern single-chip computer architectures. Strategies for achieving these goals in
the context of copying garbage collection are discussed. A multi-processor mutator/collector system is analyzed.
Finally, the Intel 80860XP architecture is studied.


Abstract
Linear logic has been proposed as one solution to the problem of garbage collection and providing efficient
“update-in-place” capabilities within a more functional language. Linear logic conserves accessibility, and
hence provides a “mechanical metaphor” which is more appropriate for a distributed-memory parallel processor
in which copying is explicit. However, linear logic’s lack of sharing may introduce significant inefficiencies of
its own.

We show an efficient implementation of linear logic called “Linear Lisp” that runs within a constant factor of
non-linear logic. This Linear Lisp allows RPLACX operations, and manages storage as safely as a non-linear
Lisp, but does not need a garbage collector. Since it offers assignments but no sharing, it occupies a twilight
zone between functional languages and imperative languages. Our Linear Lisp Machine offers many of the
same capabilities as combinator/graph reduction machines, but without their copying and garbage collection
problems.

  SIGPLAN Notices 27, 3 (March 1992), pp. 66–70.

Abstract
A simple real-time garbage collection algorithm is presented which does not copy, thereby avoiding some of the
problems caused by the asynchronous motion of objects. This in-place “treadmill” garbage collection scheme
has approximately the same complexity as other non-moving garbage collectors, thus making it usable in a high-
level language implementation where some pointers cannot be traced. The treadmill is currently being used in a
Lisp system built in Ada.
Abstract

“Lazy allocation” is a model for allocating objects on the execution stack of a high-level language which does not create dangling references. Our model provides safe transportation into the heap for objects that may survive the deallocation of the surrounding stack frame. Space for objects that do not survive the deallocation of the surrounding stack frame is reclaimed without additional effort when the stack is popped. Lazy allocation thus performs a first-level garbage collection, and if the language supports garbage collection of the heap, then our model can reduce the amortized cost of allocation in such a heap by filtering out the short-lived objects that can be more efficiently managed in LIFO order. A run-time mechanism called “result expectation” further filters out unneeded results from functions called only for their effects. In a shared-memory multi-processor environment, this filtering reduces contention for the allocation and management of global memory.

Our model performs simple local operations, and is therefore suitable for an interpreter or a hardware implementation. Its overheads for functional data are associated only with assignments, making lazy allocation attractive for “mostly functional” programming styles. Many existing stack allocation optimizations can be seen as instances of this generic model, in which some portion of these local operations have been optimized away through static analysis techniques.

Important applications of our model include the efficient allocation of temporary data structures that are passed as arguments to anonymous procedures which may or may not use these data structures in a stack-like fashion. The most important of these objects are functional arguments (funargs), which require some run-time allocation to preserve the local environment. Since a funarg is sometimes returned as a first-class value, its lifetime can survive the stack frame in which it was created. Arguments which are evaluated in a lazy fashion (Scheme “delays” or “suspensions”) are similarly handled. Variable-length argument “lists” themselves can be allocated in this fashion, allowing these objects to become “first-class”. Finally, lazy allocation correctly handles the allocation of a Scheme control stack, allowing Scheme continuations to become first-class values.

Abstract

The need to reverse a computation arises in many contexts – debugging, editor undoing, optimistic concurrency undoing, speculative computation undoing, trace scheduling, exception handling undoing, database recovery, optimistic discrete event simulations, subjunctive computing, etc. The need to analyze a reversed computation arises in the context of static analysis – liveness analysis, strictness analysis, type inference, etc. Traditional means for restoring a computation to a previous state involve checkpoints; checkpoints require time to copy, as well as space to store, the copied material. Traditional reverse abstract interpretation produces relatively poor information due to its inability to guess the previous values of assigned-to variables.

We propose an abstract computer model and a programming language – Psi-Lisp – whose primitive operations are injective and hence reversible, thus allowing arbitrary undoing without the overheads of checkpointing. Such a computer can be built from reversible conservative logic circuits, with the serendipitous advantage of dissipating far less heat than traditional Boolean AND/OR/NOT circuits. Unlike functional languages, which have one “state” for all times, Psi-Lisp has at all times one “state”, with unique predecessor and successor states.

Compiling into a reversible pseudocode can have benefits even when targeting a traditional computer. Certain optimizations, e.g., update-in-place, and compile-time garbage collection may be more easily performed, because the information may be elicited without the difficult and time-consuming iterative abstract interpretation required for most non-reversible models.

In a reversible machine, garbage collection for recycling storage can always be performed by a reversed (sub)computation. While this “collection is reversed mutation” insight does not reduce space requirements
when used for the computation as a whole, it does save space when used to recycle at finer scales. This insight also provides an explanation for the fundamental importance of the push-down stack both for recognizing palindromes and for managing storage.

Reversible computers are related to Prolog, linear logic and chemical abstract machines.


Abstract

Generation-based garbage collection has been advocated by appealing to the intuitive but vague notion that “young objects are more likely to die than old objects”. The intuition is, that if a generation-based garbage collection scheme focuses its effort on scanning recently created objects, then its scanning efforts will pay off more in the form of more recovered garbage, than if it scanned older objects. In this note, we show a counterexample of a system in which “infant mortality” is as high as you please, but for which generational garbage collection is ineffective for improving the average mark/cons ratio. Other benefits, such as better locality and a smaller number of large delays, may still make generational garbage collection attractive for such a system, however.


Abstract

We argue that intensional object identity in object-oriented programming languages and databases is best defined operationally by side-effect semantics. A corollary is that “functional” objects have extensional semantics. This model of object identity, which is analogous to the normal forms of relational algebra, provides cleaner semantics for the value-transmission operations and built-in primitive equality predicate of a programming language, and eliminates the confusion surrounding “call-by-value” and “call-by-reference” as well as the confusion of multiple equality predicates.

Implementation issues are discussed, and this model is shown to have significant performance advantages in persistent, parallel, distributed and multilingual processing environments. This model also provides insight into the “type equivalence” problem of Algol-68, Pascal and Ada.


Abstract

“Reference counting” can be an attractive form of dynamic storage management. It recovers storage promptly and (with a garbage stack instead of a free list) it can be made “real-time” – i.e., all accesses can be performed in constant time. Its major drawbacks are its inability to reclaim cycles, its count storage, and its count update overhead. Update overhead is especially irritating for functional (read-only) data where updates may dirty pristine cache lines and pages.

We show how reference count updating can be largely eliminated for functional data structures by using the “linear style” of programming that is inspired by Girard’s linear logic, and by distinguishing normal pointers from “anchored pointers”, which indicate not only the object itself, but also the depth of the stack frame that anchors the object. An “anchor” for a pointer is essentially an enclosing data structure that is temporarily locked from being collected for the duration of the anchored pointer’s existence by a deferred reference count. An “anchored pointer” thus implies a reference count increment that has been deferred until it is either cancelled or performed.

Anchored pointers are generalizations of “borrowed” pointers and “phantom” pointers. Anchored pointers can provide a solution to the “derived pointer problem” in garbage collection.

Abstract
We discuss the principles of statistical thermodynamics and their application to storage management problems. We point out problems which result from imprecise usage of the terms “information”, “state”, “reversible”, “conservative”, etc.


Abstract
Programming languages should have ‘use-once’ variables in addition to the usual ‘multiple-use’ variables. ‘Use-once’ variables are bound to linear (unshared, unaliased, or singly-referenced) objects. Linear objects are cheap to access and manage, because they require no synchronization or tracing garbage collection. Linear objects can elegantly and efficiently solve otherwise difficult problems of functional/mostly-functional systems – e.g., in-place updating and the efficient initialization of functional objects. Use-once variables are ideal for directly manipulating resources which are inherently linear such as freelists and ‘engine ticks’ in reflective languages.

A ‘use-once’ variable must be dynamically referenced exactly once within its scope. Unreferenced use-once variables must be explicitly killed, and multiply-referenced use-once variables must be explicitly copied; this duplication and deletion is subject to the constraint that some linear datatypes do not support duplication and deletion methods. Use-once variables are bound only to linear objects, which may reference other linear or non-linear objects. Non-linear objects can reference other non-linear objects, but can reference a linear object only in a way that ensures mutual exclusion.

Although implementations have long had implicit use-once variables and linear objects, most languages do not provide the programmer any help for their utilization. For example, use-once variables allow for the safe/controlled use of reified language implementation objects like single-use continuations.

Linear objects and use-once variables map elegantly into dataflow models of concurrent computation, and the graphical representations of dataflow models make an appealing visual linear programming language.


From the Preface
The International Workshop on Memory Management 1995 (IWMM’95) is a continuation of the excellent series started by Yves Bekkers and Jacques Cohen with IWMM’92. The present volume assembles the refereed and invited technical papers which were presented during this year’s workshop.


From the notes
There is no well-defined memory-management library API which would allow programmers to easily choose the best memory management implementation for their application.

Some languages allow replacement of their memory management functions, but usually only the program API is specified, hence replacement of the entire program interface is required.
Few languages support multiple memory management policies within a single program. Those that do use proprietary memory management policies.

We believe that the design of an abstract program API is a prerequisite to the design of a “server” API and eventually an API that would permit multiple cooperating memory “servers”. If the interface is simple yet powerful enough to encompass most memory management systems, it stands a good chance of being widely adopted.


Abstract

Dynamic storage allocation is used heavily in many application areas including interpreters, simulators, optimizers, and translators. We describe research that can improve all aspects of the performance of dynamic storage allocation by predicting the lifetimes of short-lived objects when they are allocated. Using five significant, allocation-intensive C programs, we show that a great fraction of all bytes allocated are short-lived (> 90% in all cases). Furthermore, we describe an algorithm for lifetime prediction that accurately predicts the lifetimes of 42–99% of all objects allocated. We describe and simulate a storage allocator that takes advantage of lifetime prediction of short-lived objects and show that it can significantly improve a program’s memory overhead and reference locality, and even, at times, improve CPU performance as well.


Abstract

Generational techniques have been very successful in reducing the impact of garbage collection algorithms upon the performance of programs. However, it is impossible for designers of collection algorithms to anticipate the memory allocation behavior of all applications in advance. Existing generational collectors rely upon the applications programmer to tune the behavior of the collector to achieve maximum performance for each application. Unfortunately, because the many tuning parameters require detailed knowledge of both the collection algorithm and the program allocation behavior in order to be used effectively, such tuning is difficult and error prone. We propose a new garbage collection algorithm that uses just two easily understood tuning parameters that directly reflect the maximum memory and pause time constraints familiar to application programmers and users.

Like generational collectors, ours divides memory into two spaces, one for short-lived, and another for long-lived objects. Unlike previous work, our collector dynamically adjusts the boundary between these two spaces in order to directly meet the resource constraints specified by the user. We describe two methods for adjusting this boundary, compare them with several existing algorithms, and show how effectively ours meets the specified constraints. Our pause time collector saved memory by holding median pause times closer to the constraint than the other pause time constrained algorithm and, when not over-constrained, our memory constrained collector exhibited the lowest CPU overhead of the algorithms we measured yet was capable of maintaining a maximum memory constraint.


Abstract

This paper introduces a copying garbage collection algorithm which is able to compact most of the accessible storage in the heap without having an explicitly defined set of pointers that contain all the roots of all accessible storage. Using “hints” found in the processor’s registers and stack, the algorithm is able to divide heap allocated objects into two groups: those that might be referenced by a pointer in the stack or registers, and those that
are not. The objects which might be referenced are left in place, and the other objects are copied into a more compact representation.

A Lisp compiler and runtime system which uses such a collector need not have complete control of the processor in order to force a certain discipline on the stack and registers. A Scheme implementation has been done for the Digital WRL Titan processor which uses a garbage collector based on this “mostly copying” algorithm. Like other languages for the Titan, it uses the Mahler intermediate language as its target. This simplifies the compiler and allows it to take advantage of the significant machine dependent optimizations provided by Mahler. The common intermediate language also simplifies call-outs from Scheme programs to functions written in other languages and call-backs from functions in other languages.

Measurements of the Scheme implementation show that the algorithm is efficient, as little unneeded storage is retained and only a very small fraction of the heap is left in place.

Simple pointer manipulation protocols also mean that compiler support is not needed in order to correctly handle pointers. Thus it is reasonable to provide garbage collected storage in languages such as C. A collector written in C which uses this algorithm is included in the Appendix.


Abstract

The “mostly-copying” garbage collection algorithm provides a way to perform compacting garbage collection in spite of the presence of ambiguous pointers in the root set. As originally defined, each collection required almost all accessible objects to be moved. While adequate for many applications, programs that retained a large amount of storage spent a significant amount of time garbage collecting. To improve performance of these applications, a generational version of the algorithm has been designed. This note reports on this extension of the algorithm, and its application in collectors for Scheme and C++.


Abstract

In this paper, we present Hoard, a memory allocator for shared-memory multiprocessors. We prove that its worst-case memory fragmentation is asymptotically equivalent to that of an optimal uniprocessor allocator. We present experiments that demonstrate its speed and scalability.


Abstract

Current general-purpose memory allocators do not provide sufficient speed or flexibility for modern high-performance applications. Highly-tuned general purpose allocators have per-operation costs around one hundred cycles, while the cost of an operation in a custom memory allocator can be just a handful of cycles. To achieve high performance, programmers often write custom memory allocators from scratch – a difficult and error-prone process.

In this paper, we present a flexible and efficient infrastructure for building memory allocators that is based on C++ templates and inheritance. This novel approach allows programmers to build custom and general-purpose allocators as “heap layers” that can be composed without incurring any additional runtime overhead or additional
programming cost. We show that this infrastructure simplifies allocator construction and results in allocators that either match or improve the performance of heavily-tuned allocators written in C, including the Kingsley allocator and the GNU obstack library. We further show this infrastructure can be used to rapidly build a general-purpose allocator that has performance comparable to the Lea allocator, one of the best uniprocessor allocators available. We thus demonstrate a clean, easy-to-use allocator interface that seamlessly combines the power and efficiency of any number of general and custom allocators within a single application.


**Abstract**

We describe a technique for storage allocation and garbage collection in the absence of significant co-operation from the code using the allocator. This limits garbage collection overhead to the time actually required for garbage collection. In particular, application programs that rarely or never make use of the collector no longer encounter a substantial performance penalty. This approach greatly simplifies the implementation of languages supporting garbage collection. It further allows conventional compilers to be used with a garbage collector, either as the primary means of storage reclamation, or as a debugging tool.


**Abstract**

We present a method for adapting garbage collectors designed to run sequentially with the client, so that they may run concurrently with it. We rely on virtual memory hardware to provide information about pages that have been updated or “dirtied” during a given period of time. This method has been used to construct a mostly parallel trace-and-sweep collector that exhibits very short pause times. Performance measurements are given.


**Abstract**

Conservative garbage collectors are commonly used in combination with conventional C programs. Empirically, this usually works well. However, there are no guarantees that this is safe in the presence of “improved” compiler optimization. We propose that C compilers provide a facility to suppress optimizations that are unsafe in the presence of conservative garbage collection. Such a facility can be added to an existing compiler at very minimal cost, provided the additional analysis is done in a machine-independent source-to-source prepass. Such a prepass may also check the source code for garbage-collector-safety.


**Abstract**

We call a garbage collector conservative if it has only partial information about the location of pointers, and is thus forced to treat arbitrary bit patterns as though they might be pointers, in at least some cases. We show that some very inexpensive, but previously unused techniques can have dramatic impact on the effectiveness of conservative garbage collectors in reclaiming memory. Our most significant observation is that static data that appears to point to the heap should not result in misidentified reference to the heap. The garbage collector has enough information to allocate around such references. We also observe that programming style has a significantly impact on the amount of spuriously retained storage, typically even if the collector is not terribly conservative. Some fairly common C and C++ programming styles significantly decrease the effectiveness of
any garbage collector. These observations suffice to explain some of the different assessments of conservative collection that have appeared in the literature.


Abstract
Cache misses are currently a major factor in the cost of garbage collection, and we expect them to dominate in the future. Traditional garbage collection algorithms exhibit relatively little temporal locality; each live object in the heap is likely to be touched exactly once during each garbage collection. We measure two techniques for dealing with this issue: prefetch-on-grey, and lazy sweeping. The first of these is new in this context. Lazy sweeping has been in common use for a decade. It was introduced as a mechanism for reducing paging and pause times; we argue that it is also crucial for eliminating cache misses during the sweep phase.

Our measurements are obtained in the context of a non-moving garbage collector. Fully copying garbage collection inherently requires more traffic through the cache, and thus probably also stands to benefit substantially from something like the prefetch-on-grey technique. Generational garbage collection may reduce the benefit of these techniques for some applications, but experiments with a non-moving generational collector suggest that they remain quite useful.


Abstract
Conservative garbage collectors can automatically reclaim unused memory in the absence of precise pointer location information. If a location can possibly contain a pointer, it is treated by the collector as though it contained a pointer. Although it is commonly assumed that this can lead to unbounded space use due to misidentified pointers, such extreme space use is rarely observed in practice, and then generally only if the number of misidentified pointers is itself unbounded. We show that if the program manipulates only data structures satisfying a simple GC-robustness criterion, then a bounded number of misidentified pointers can result at most in increasing space usage by a constant factor. We argue that nearly all common data structures are already GC-robust, and it is typically easy to identify and replace those that are not. Thus it becomes feasible to prove space bounds on programs collected by mildly conservative garbage collectors, such as the one in Barabash et al. (2001). The worst-case space overhead introduced by such mild conservatism is comparable to the worst-case fragmentation overhead for inherent in any non-moving storage allocator. The same GC-robustness criterion also ensures the absence of temporary space leaks of the kind discussed in Rojemo (1995) for generational garbage collectors.


Abstract
We compare two different facilities for running cleanup actions for objects that are about to reach the end of their life. Destructors, such as we find in C++, are invoked synchronously when an object goes out of scope. They make it easier to implement cleanup actions for objects of well-known lifetime, especially in the presence of exceptions. Languages like Java, Modula-3, and C# provide a different kind of “finalization” facility: Cleanup methods may be run when the garbage collector discovers a heap object to be otherwise inaccessible. Unlike C++ destructors, such methods run in a separate thread at some much less well-defined time. We argue that these are fundamentally different, and potentially complementary, language facilities. We also try to resolve some common misunderstandings about finalization in the process. In particular: 1. The asynchronous nature of finalizers is not just an accident of implementation or a shortcoming of tracing collectors; it is necessary for correctness of client code, fundamentally affects how finalizers must be written, and how finalization facilities should be presented to the user. 2. An object may legitimately be finalized while one of its methods are still running. This should and can be addressed by the language specification and client code.

6.1. Bibliography

**Abstract**

An algorithm is presented that searches for the location, “i,” of the first occurrence of a character string, “pat,” in another string, “string.” During the search operation, the characters of pat are matched starting with the last character of pat. The information gained by starting the match at the end of the pattern often allows the algorithm to proceed in large jumps through the text being searched. Thus the algorithm has the unusual property that, in most cases, not all of the first i characters of string are inspected. The number of characters actually inspected (on the average) decreases as a function of the length of pat. For a random English pattern of length 5, the algorithm will typically inspect i/4 characters of string before finding a match at i. Furthermore, the algorithm has been implemented so that (on the average) fewer than i + patlen machine instructions are executed. These conclusions are supported with empirical evidence and a theoretical analysis of the average behavior of the algorithm. The worst case behavior of the algorithm is linear in i + patlen, assuming the availability of array space for tables linear in patlen plus the size of the alphabet.


**Abstract**

The Memory Pool System (MPS) is a very general, adaptable, flexible, reliable, and efficient memory management system. It permits the flexible combination of memory management techniques, supporting manual and automatic memory management, in-line allocation, finalization, weakness, and multiple simultaneous cooperating incremental generational garbage collections. It also includes a library of memory pool classes implementing specialized memory management policies.

Between 1994 and 2001, Harlequin (now part of Global Graphics) invested about thirty person-years of effort developing the MPS. The system contained many innovative techniques and abstractions which were kept secret. In 1997 Richard Brooksby, the manager and chief architect of the project, and Nicholas Barnes, a senior developer, left Harlequin to form their own consultancy company, Ravenbrook, and in 2001, Ravenbrook acquired the MPS technology from Global Graphics. We are happy to announce that we are publishing the source code and documentation under an open source licence. This paper gives an overview of the system.


**Abstract**

Improving the performance of C programs has been a topic of great interest for many years. Both hardware technology and compiler optimization research has been applied in an effort to make C programs execute faster. In many application domains, the C++ language is replacing C as the programming language of choice. In this paper, we measure the empirical behavior of a group of significant C and C++ programs and attempt to identify and quantify behavioral differences between them. Our goal is to determine whether optimization technology that has been successful for C programs will also be successful in C++ programs. We furthermore identify behavioral characteristics of C++ programs that suggest optimizations that should be applied in those programs. Our results show that C++ programs exhibit behavior that is significantly different than C programs. These
results should be of interest to compiler writers and architecture designers who are designing systems to execute object-oriented programs.


Abstract
We describe a new method for determining when an object can be garbage collected. The method does not require marking live objects. Instead, each object \( X \) is dynamically associated with a stack frame \( M \), such that \( X \) is collectable when \( M \) pops. Because \( X \) could have been dead earlier, our method is conservative. Our results demonstrate that the method nonetheless identifies a large percentage of collectable objects. The method has been implemented in Sun’s Java™ Virtual Machine interpreter, and results are presented based on this implementation.


Abstract
A new, high performance Smalltalk-80™ implementation is described which builds directly upon two previous implementation efforts. This implementation supports a large object space while retaining compatibility with previous Smalltalk-80™ images. The implementation utilizes a interpreter which incorporates a generation based garbage collector and which does not have an object table. This paper describes the design decisions which lead to this implementation and reports preliminary performance results.


Abstract
A simple nonrecursive list structure compacting scheme or garbage collector suitable for both compact and LISP-like list structures is presented. The algorithm avoids the need for recursion by using the partial structure as it is built up to keep track of those lists that have been copied.


Abstract
This paper presents two techniques for improving garbage collection performance: generational stack collection and profile-driven pretenuring. The first is applicable to stack-based implementations of functional languages while the second is useful for any generational collector. We have implemented both techniques in a generational collector used by the TIL compiler, and have observed decreases in garbage collection times of as much as 70% and 30%, respectively.

Functional languages encourage the use of recursion which can lead to a long chain of activation records. When a collection occurs, these activation records must be scanned for roots. We show that scanning many activation records can take so long as to become the dominant cost of garbage collection. However, most deep stacks unwind very infrequently, so most of the root information obtained from the stack remains unchanged across successive garbage collections. Generational stack collection greatly reduces the stack scan cost by reusing information from previous scans.

Generational techniques have been successful in reducing the cost of garbage collection. Various complex heap
arrangements and tenuring policies have been proposed to increase the effectiveness of generational techniques by reducing the cost and frequency of scanning and copying. In contrast, we show that by using profile information to make lifetime predictions, pretenuring can avoid copying data altogether. In essence, this technique uses a refinement of the generational hypothesis (most data die young) with a locality principle concerning the age of data: most allocations sites produce data that immediately dies, while a few allocation sites consistently produce data that survives many collections.


Abstract
Processor and memory technology trends show a continual increase in the cost of accessing main memory. Machine designers have tried to mitigate the effect of this trend through a variety of techniques that attempt to reduce or tolerate memory latency. These techniques, unfortunately, have only been partially successful for pointer-manipulating programs. Recent research has demonstrated that these programs can benefit greatly from the complementary approach of reorganizing pointer data structures to improve cache locality. This paper describes how a generational garbage collector can be used to achieve a cache-conscious data layout, in which objects with high temporal affinity are placed next to each other, so they are likely to reside in the same cache block. The paper demonstrates the feasibility of collecting low overhead, real-time profiling information about data access patterns for object-oriented languages, and describes a new copying algorithm that utilizes this information to produce a cache-conscious object layout. Preliminary results indicate that this technique reduces cache miss rates by 21-42%, and improves program performance by 14-37%.


Abstract
If a fixed exponentially decreasing probability distribution function is used to model every object’s lifetime, then the age of an object gives no information about its future life expectancy. This radioactive decay model implies that there can be no rational basis for deciding which live objects should be promoted to another generation. Yet there remains a rational basis for deciding how many objects to promote, when to collect garbage, and which generations to collect.

Analysis of the model leads to a new kind of generational garbage collector whose effectiveness does not depend upon heuristics that predict which objects will live longer than others.

This result provides insight into the computational advantages of generational garbage collection, with implications for the management of objects whose life expectancies are difficult to predict.


Abstract
A concise and unified view of the numerous existing algorithms for performing garbage collection of linked data structures is presented. The emphasis is on garbage collection proper, rather than on storage allocation.

First, the classical garbage collection algorithms and their marking and collecting phases, with and without compacting, are discussed.

Algorithms describing these phases are classified according to the type of cells to be collected: those for collecting single-sized cells are simpler than those for varisized cells. Recently proposed algorithms are presented and compared with the classical ones. Special topics in garbage collection are also covered. A bibliography with topical annotations is included.

Abstract
Mark and sweep garbage collectors (GC) are classical but still very efficient automatic memory management systems. Although challenged by other kinds of systems, such as copying collectors, mark and sweep collectors remain among the best in terms of performance.

This paper describes our implementation of an efficient mark and sweep garbage collector tailored to each program. Compiler support provides the type information required to statically and automatically generate this customized garbage collector. The segregation of object by type allows the production of a more efficient GC code. This technique, implemented in SmallEiffel, our compiler for the object-oriented language Eiffel, is applicable to other languages and other garbage collection algorithms, be they distributed or not.

We present the results obtained on programs featuring a variety of programming styles and compare our results to a well-known and high-quality garbage collector.


Abstract
The automatic reclamation of storage for unreferenced objects is very important in object databases. Existing language system algorithms for automatic storage reclamation have been shown to be inappropriate. In this paper, we investigate methods to improve the performance of algorithms for automatic storage reclamation of object databases. These algorithms are based on a technique called partitioned garbage collection, in which a subset of the entire database is collected independently of the rest. Specifically, we investigate the policy that is used to select what partition in the database should be collected. The new partition selection policies that we propose and investigate are based on the intuition that the values of overwritten pointers provide good hints about where to find garbage. Using trace-driven simulation, we show that one of our policies requires less I/O to collect more garbage than any existing implementable policy and performs close to an impractical-to-implement but near-optimal policy over a wide range of database sizes and connectivities.


Abstract
A fundamental problem in automating object database storage reclamation is determining how often to perform garbage collection. We show that the choice of collection rate can have a significant impact on application performance and that the “best” rate depends on the dynamic behavior of the application, tempered by the particular performance goals of the user. We describe two semi-automatic, self-adaptive policies for controlling collection rate that we have developed to address the problem. Using trace-driven simulations, we evaluate the performance of the policies on a test database application that demonstrates two distinct reclustering behaviors. Our results show that the policies are effective at achieving user-specified levels of I/O operations and database garbage percentage. We also investigate the sensitivity of the policies over a range of object connectivities. The evaluation demonstrates that semi-automatic, self-adaptive policies are a practical means for flexibly controlling garbage collection rate.


6.1. Bibliography
Abstract
We improved the performance of garbage collection in the Standard ML of New Jersey system by using the virtual memory facilities provided by the Mach kernel. We took advantage of Mach’s support for large sparse address spaces and user-defined paging servers. We decreased the elapsed time for realistic applications by as much as a factor of 4.


From the back cover
Using techniques developed in the classroom at America Online’s Programmer’s University, Michael Daconta deftly pilots programmers through the intricacies of the two most difficult aspects of C++ programming: pointers and dynamic memory management. Written by a programmer for programmers, this no-nonsense, nuts-and-bolts guide shows you how to fully exploit advanced C++ programming features, such as creating class-specific allocators, understanding references versus pointers, manipulating multidimensional arrays with pointers, and how pointers and dynamic memory are the core of object-oriented constructs like inheritance, name-mangling, and virtual functions.


From the introduction
A particularly troublesome phenomenon, thrashing, may seriously interfere with the performance of paged memory systems, reducing computing giants (Multics, IBM System 360, and others not necessarily excepted) to computing dwarfs. The term thrashing denotes excessive overhead and severe performance degradation or collapse caused by too much paging. Thrashing inevitably turns a shortage of memory space into a surplus of processor time.


Abstract
The need for automatic storage allocation arises from desires for program modularity, machine independence, and resource sharing. Virtual memory is an elegant way of achieving these objectives. In a virtual memory, the addresses a program may use to identify information are distinguished from the addresses the memory system uses to identify physical storage sites, and program-generated addresses are translated automatically to the corresponding machine addresses. Two principal methods for implementing virtual memory, segmentation and paging, are compared and contrasted. Many contemporary implementations have experienced one or more of these problems: poor utilization of storage, thrashing, and high costs associated with loading information into memory. These and subsidiary problems are studied from a theoretic view, and are shown to be controllable by a proper combination of hardware and memory management policies.


Abstract
A program’s working set $W(t, T)$ at time $t$ is the set of distinct pages among the $T$ most recently referenced pages. Relations between the average working-set size, the missing-page rate, and the interreference-interval
distribution may be derived both from time-average definitions and from ensemble-average (statistical) definitions. An efficient algorithm for estimating these quantities is given. The relation to LRU (least recently used) paging is characterized. The independent-reference model, in which page references are statistically independent, is used to assess the effects of interpage dependencies on working-set size observations. Under general assumptions, working-set size is shown to be normally distributed.


From the introduction

Automatic storage management, or garbage collection, is a feature that can ease program development and enhance program reliability. Many high-level languages other than C++ provide garbage collection. This paper proposes the use of “smart pointer” template classes as an interface for the use of garbage collection in C++. Template classes and operator overloading are techniques allowing language extension at the level of user code; I claim that using these techniques to create smart pointer classes provides a syntax for manipulating garbage-collected storage safely and conveniently. Further, the use of a smart-pointer template class offers the possibility of implementing the collector at the user-level, without requiring support from the compiler. If such a compiler-independent implementation is possible with adequate performance, then programmers can start to write code using garbage collection without waiting for language and compiler modifications. If the use of such a garbage collection interface becomes widespread, then C++ compilation systems can be built to specially support the garbage collection interface, thereby allowing the use of collection algorithms with enhanced performance.


Abstract

Dynamic storage allocation is an important part of a large class of computer programs written in C and C++. High-performance algorithms for dynamic storage allocation have been, and will continue to be, of considerable interest. This paper presents detailed measurements of the cost of dynamic storage allocation in 11 diverse C and C++ programs using five very different dynamic storage allocation implementations, including a conservative garbage collection algorithm. Four of the allocator implementations measured are publicly-available on the Internet. A number of the programs used in these measurements are also available on the Internet to facilitate further research in dynamic storage allocation. Finally, the data presented in this paper is an abbreviated version of more extensive statistics that are also publicly-available on the Internet.


Abstract

This paper describes a new way of solving the storage reclamation problem for a system such as Lisp that allocates storage automatically from a heap, and does not require the programmer to give any indication that particular items are no longer useful or accessible. A reference count scheme for reclaiming non-self-referential structures, and a linearizing, compacting, copying scheme to reorganize all storage at the users discretion are proposed. The algorithms are designed to work well in systems which use multiple levels of storage, and large virtual address space. They depend on the fact that most cells are referenced exactly once, and that reference counts need only be accurate when storage is about to be reclaimed. A transaction file stores changes to reference counts, and a multiple reference table stores the count for items which are referenced more than once.


Abstract
As an example of cooperation between sequential processes with very little mutual interference despite frequent manipulations of a large shared data space, a technique is developed which allows nearly all of the activity needed for garbage detection and collection to be performed by an additional processor operating concurrently with the processor devoted to the computation proper. Exclusion and synchronization constraints have been kept as weak as could be achieved; the severe complexities engendered by doing so are illustrated.


Abstract

We consider the problem of supporting compacting garbage collection in the presence of modern compiler optimizations. Since our collector may move any heap object, it must accurately locate, follow, and update all pointers and values derived from pointers. To assist the collector, we extend the compiler to emit tables describing live pointers, and values derived from pointers, at each program location where collection may occur. Significant results include identification of a number of problems posed by optimizations, solutions to those problems, a working compiler, and experimental data concerning table sizes, table compression, and time overhead of decoding tables during collection. While gc support can affect the code produced, our sample programs show no significant changes; the table sizes are a modest fraction of the size of the optimized code, and stack tracing is a small fraction of total gc time. Since the compiler enhancements are also modest, we conclude that the approach is practical.


Abstract

Heap allocation with copying garbage collection is a general storage management technique for modern programming languages. It is believed to have poor memory subsystem performance. To investigate this, we conducted an in-depth study of the memory subsystem performance of heap allocation for memory subsystems found on many machines. We studied the performance of mostly-functional Standard ML programs which made heavy use of heap allocation. We found that most machines support heap allocation poorly. However, with the appropriate memory subsystem organization, heap allocation can have good performance. The memory subsystem property crucial for achieving good performance was the ability to allocate and initialize a new object into the cache without a penalty. This can be achieved by having subblock placement with a subblock size of one word with a write allocate policy, along with fast page-mode writes or a write buffer. For caches with subblock placement, the data cache overhead was under 9% for a 64k or larger data cache; without subblock placement the overhead was often higher than 50%.


Abstract

Heap allocation with copying garbage collection is believed to have poor memory subsystem performance. We conducted a study of the memory subsystem performance of heap allocation for memory subsystems found on many machines. We found that many machines support heap allocation poorly. However, with the appropriate memory subsystem organization, heap allocation can have good memory subsystem performance.


Abstract
This paper presents the design and implementation of a “quasi real-time” garbage collector for Concurrent Caml Light, an implementation of ML with threads. This two-generation system combines a fast, asynchronous copying collector on the young generation with a non-disruptive concurrent marking collector on the old generation. This design crucially relies on the ML compile-time distinction between mutable and immutable objects.


Abstract
We describe and prove the correctness of a new concurrent mark-and-sweep garbage collection algorithm. This algorithm derives from the classical on-the-fly algorithm from Dijkstra et al. A distinguishing feature of our algorithm is that it supports multiprocessor environments where the registers of running processes are not readily accessible, without imposing any overhead on the elementary operations of loading a register or reading or initializing a field. Furthermore our collector never blocks running mutator processes except possibly on requests for free memory; in particular, updating a field or creating or marking or sweeping a heap object does not involve system-dependent synchronization primitives such as locks. We also provide support for process creation and deletion, and for managing an extensible heap of variable-sized objects.


Abstract
This paper describes a new language feature that allows dynamically allocated objects to be saved from deallocation by an automatic storage management system so that clean-up or other actions can be performed using the data stored within the objects. The program has full control over the timing of clean-up actions, which eliminates several potential problems and often eliminates the need for critical sections in code that interacts with clean-up actions. Our implementation is “generation-friendly” in the sense that the additional overhead within the mutator is proportional to the number of clean-up actions actually performed.


From the introduction
This paper shows how the behaviour of smart pointers diverges from that of pointers in certain common C++ constructs. Given this, we conclude that the C++ programming language does not support seamless smart pointers: smart pointers cannot transparently replace raw pointers in all ways except declaration syntax. We show that this conclusion also applies to accessors.


Abstract
Our research is concerned with compiler- independent, tag-free garbage collection for the C++ programming language. This paper presents a mark-and-sweep collector, and explains how it ameliorates shortcomings of a previous copy collector. The new collector, like the old, uses C++’s facilities for creating abstract data types to define a tracked reference type, called roots, at the level of the application program. A programmer wishing to utilize the garbage collection service uses these roots in place of normal, raw pointers. We present a detailed study of the cost of using roots, as compared to both normal pointers and reference counted pointers, in terms of instruction counts. We examine the efficiency of a small C++ application using roots, reference counting, manual reclamation, and conservative collection. Coding the application to use garbage collection, and analyzing the resulting efficiency, helped us identify a number of memory leaks and inefficiencies in the original, manually
reclaimed version. We find that for this program, garbage collection using roots is much more efficient than reference counting, though less efficient than manual reclamation. It is hard to directly compare our collector to the conservative collector because of the differing efficiencies of their respective memory allocators.


Our summary
(This short memo doesn’t have an abstract. Basically, it describes the plan for the LISP II Relocating Garbage Collector. It has four phases: marking, collection, relocation and moving. Marking is by recursive descent using a bit table. The remaining phases are linear sweeps through the bit table. The collection phase calculates how much everything needs to move, storing this information in the free blocks. The relocation phase updates all relocatable addresses. The moving phase moves the surviving objects into one contiguous block.)


Abstract
We propose adding safe, efficient garbage collection to C++, eliminating the possibility of storage-management bugs and making the design of complex, object-oriented systems much easier. This can be accomplished with almost no change to the language itself and only small changes to existing implementations, while retaining compatibility with existing class libraries.


Abstract
The model of Larchant is that of a Shared Address Space (spanning every site in a network including secondary storage) with Persistence By Reachability. To provide the illusion of a shared address space across the network, despite the fact that site memories are disjoint, Larchant implements a distributed shared memory mechanism. Reachability is accessed by tracing the pointer graph, starting from the persistent root, and reclaiming unreachable objects. This is the task of Garbage Collection (GC).

GC was until recently thought to be intractable in a large-scale system, due to problems of scale, incoherence, asynchrony, and performance. This thesis presents the solutions that Larchant proposes to these problems.

The GC algorithm in Larchant combines tracing and reference-listing. It traces whenever economically feasible, i.e., as long as the memory subset being collected remains local to a site, and counts references that would cost I/O traffic to trace. GC is orthogonal to coherence, i.e., makes progress even if only incoherent replicas are locally available. The garbage collector runs concurrently and asynchronously to applications. The reference-listing boundary changes dynamically and seamlessly, and independently at each site, in order to collect cycles of unreachable objects.

We prove formally that our GC algorithm is correct, i.e., it is safe and live. The performance results from our Larchant prototype show that our design goals (scalability, coherence orthogonality, and good performance) are fulfilled.


Abstract
Caching and persistence support efficient, convenient and transparent distributed data sharing. The most natural model of persistence is persistence by reachability, managed automatically by a garbage collector (GC). We propose a very general model of such a system (based on distributed shared memory) and a scalable, asynchronous
distributed GC algorithm. Within this model, we show sufficient and widely applicable correctness conditions for the interactions between applications, store, memory, coherence, and GC.

The GC runs as a set of processes (local to each participating machine) communicating by asynchronous messages. Collection does not interfere with applications by setting locks, polluting caches, or causing I/O; this requirement raised some novel and interesting challenges which we address in this article. The algorithm is safe and live; it is not complete, i.e. it collects some distributed cycles of garbage but not necessarily all.


**Abstract**

A new algorithm is introduced for garbage collecting a heap which contains shared data structures accessed from a scatter table. The scheme provides for the purging of useless entries from the scatter table with no traversals beyond the two required by classic collection schemes. For languages which use scatter tables to sustain unique existence of complex structures, like natural variables of SNOBOL, it indirectly allows liberal use of a single scatter table by ensuring efficient deletion of useless entries. Since the scatter table is completely restructured during the course of execution, the hashing scheme itself is easily altered during garbage collection whenever skewed loading of the scatter table warrants abandonment of the old hashing. This procedure is applicable to the maintenance of dynamic structures such as those in information retrieval schemes or in languages like LISP and SNOBOL.


**Abstract**

Deutsch and Bobrow propose a storage reclamation scheme for a heap which is a hybrid of garbage collection and reference counting. The point of the hybrid scheme is to keep track of very low reference counts between necessary invocation of garbage collection so that nodes which are allocated and rather quickly abandoned can be returned to available space, delaying necessity for garbage collection. We show how such a scheme may be implemented using the mark bit already required in every node by the garbage collector. Between garbage collections that bit is used to distinguish nodes with a reference count known to be one. A significant feature of our scheme is a small cache of references to nodes whose implemented counts “ought to be higher” which prevents the loss of logical count information in simple manipulations of uniquely referenced structures.


**From the introduction**

In this note we advance reference counting as a storage management technique viable for implementing recursive languages like ISWIM or pure LISP with the *labels* construct for implementing mutual recursion from SCHEME. *Labels* is derived from *letrec* and displaces the *label* operator, a version of the paradoxical Y-combinator. The key observation is that the requisite circular structure (which ordinarily cripples reference counts) occurs only within the language–rather than the user–structure, and that the references into this structure are well-controlled.


**Abstract**

The allocation and disposal of memory is a ubiquitous operation in most programs. Rarely do programmers concern themselves with details of memory allocators; most assume that memory allocators provided by the system
perform well. This paper presents a performance evaluation of the reference locality of dynamic storage allocation algorithms based on trace-driven simulation of five large allocation-intensive C programs. In this paper, we show how the design of a memory allocator can significantly affect the reference locality for various applications. Our measurements show that poor locality in sequential-fit algorithms reduces program performance, both by increasing paging and cache miss rates. While increased paging can be debilitating on any architecture, cache misses rates are also important for modern computer architectures. We show that algorithms attempting to be space-efficient, by coalescing adjacent free objects show poor reference locality, possibly negating the benefits of space efficiency. At the other extreme, algorithms can expend considerable effort to increase reference locality yet gain little in total execution performance. Our measurements suggest an allocator design that is both very fast and has good locality of reference.


Abstract
The allocation and disposal of memory is a ubiquitous operation in most programs. Rarely do programmers concern themselves with details of memory allocators; most assume that memory allocators provided by the system perform well. Yet, in some applications, programmers use domain-specific knowledge in an attempt to improve the speed or memory utilization of memory allocators. In this paper, we describe a program (CustoMalloc) that synthesizes a memory allocator customized for a specific application. Our experiments show that the synthesized allocators are uniformly faster than the common binary-buddy (BSD) allocator, and are more space efficient. Constructing a custom allocator requires little programmer effort. The process can usually be accomplished in a few minutes, and yields results superior even to domain-specific allocators designed by programmers. Our measurements show the synthesized allocators are from two to ten times faster than widely used allocators.


Abstract
This report is a discussion of various techniques for representing type information in dynamically typed languages, as implemented on general-purpose machines (and costs are discussed in terms of modern RISC machines). It is intended to make readily available a large body of knowledge that currently has to be absorbed piecemeal from the literature or re-invented by each language implementor. This discussion covers not only tagging schemes but other forms of representation as well, although the discussion is strictly limited to the representation of type information. It should also be noted that this report does not purport to contain a survey of the relevant literature. Instead, this report gathers together a body of folklore, organizes it into a logical structure, makes some generalizations, and then discusses the results in terms of modern hardware.


Abstract
This article presents a technique for allowing the early recovery of storage space occupied by garbage data. The idea is similar to that of generational garbage collection, except that the heap is partitioned based on a static analysis of data type definitions rather than on the approximate age of allocated objects. A prototype implementation is presented, along with initial results and ideas for future work.


Abstract
This paper presents a new model for scheduling the work of an incremental garbage collector in a system with hard real time requirements. The method utilizes the fact that just some of the processes in the system have to meet hard real time requirements and that these processes typically run periodically, a fact that we can make use of when scheduling the garbage collection. The work of the collector is scheduled to be performed in the pauses between the critical processes and is suspended when the processes with hard real time requirements run. It is shown that this approach is feasible for many real time systems and that it leaves the time-critical parts of the system undisturbed from garbage collection induced delays.


Abstract
Incremental algorithms are often used to interleave the work of a garbage collector with the execution of an application program, the intention being to avoid long pauses. However, overestimating the worst-case storage needs of the program often causes all the garbage collection work to be performed in the beginning of the garbage collection cycles, slowing down the application program to an unwanted degree. This paper explores an approach to distributing the work more evenly over the garbage collection cycle.


Abstract
The complexity of systems for automatic control and other safety-critical applications grows rapidly. Computer software represents an increasing part of the complexity. As larger systems are developed, we need to find scalable techniques to manage the complexity in order to guarantee high product quality. Memory management is a key quality factor for these systems. Automatic memory management, or garbage collection, is a technique that significantly reduces the complex problem of correct memory management. The risk of software errors decreases and development time is reduced.

Garbage collection techniques suitable for interactive and soft real-time systems exist, but few approaches are suitable for systems with hard real-time requirements, such as control systems (embedded systems). One part of the problem is solved by incremental garbage collection algorithms, which have been presented before. We focus on the scheduling problem which forms the second part of the problem, i.e. how the work of a garbage collector should be scheduled in order to disturb the application program as little as possible. It is studied how a priori scheduling analysis of systems with automatic memory management can be made. The field of garbage collection research is thus joined with the field of scheduling analysis in order to produce a practical synthesis of the two fields.

A scheduling strategy is presented that employs the properties of control systems to ensure that no garbage collection work is performed during the execution of critical processes. The hard real-time part of the system is thus never disturbed by garbage collection work. Existing incremental garbage collection algorithms are adapted to the presented strategy. Necessary modifications of the algorithms and the real-time kernel are discussed. A standard scheduling analysis technique, rate monotonic analysis, is extended in order to make a priori analysis of the schedulability of the garbage collector possible.

The scheduling algorithm has been implemented in an industrially relevant real-time environment in order to show that the strategy is feasible in practice. The experimental evaluation shows that predictable behaviour and sub-millisecond worst-case delays can be achieved on standard hardware even by a non-optimized prototype garbage collector.


Abstract
Reachability-based persistence imposes new requirements for main memory management in general, and garbage collection in particular. After a brief introduction to the characteristics and requirements of reachability-based persistence, we present the design of a run-time storage manager for Persistent Smalltalk and Persistent Modula-3, which allows the reclamation of storage from both temporary objects and buffered persistent objects.


**Abstract**

Generational garbage collectors are able to achieve very small pause times by concentrating on the youngest (most recently allocated) objects when collecting, since objects have been observed to die young in many systems. Generational collectors must keep track of all pointers from older to younger generations, by “monitoring” all stores into the heap. This write barrier has been implemented in a number of ways, varying essentially in the granularity of the information observed and stored. Here we examine a range of write barrier implementations and evaluate their relative performance within a generation scavenging garbage collector for Smalltalk.


**Abstract**

Remembered sets and dirty bits have been proposed as alternative implementations of the write barrier for garbage collection. There are advantages to both approaches. Dirty bits can be efficiently maintained with minimal, bounded overhead per store operation, while remembered sets concisely, and accurately record the necessary information. Here we present evidence to show that hybrids can combine the virtues of both schemes and offer competitive performance. Moreover, we argue that a hybrid can better avoid the devils that are the downfall of the separate alternatives.


**Abstract**

Many operating systems allow user programs to specify the protection level (inaccessible, read-only, read-write) of pages in their virtual memory address space, and to handle any protection violations that may occur. Such page-protection techniques have been exploited by several user-level algorithms for applications including generational garbage collection and persistent stores. Unfortunately, modern hardware has made efficient handling of page protection faults more difficult. Moreover, page-sized granularity may not match the natural granularity of a given application. In light of these problems, we reevaluate the usefulness of page-protection primitives in such applications, by comparing the performance of implementations that make use of the primitives with others that do not. Our results show that for certain applications software solutions outperform solutions that rely on page-protection or other related virtual memory primitives.


**Abstract**

We describe a memory management toolkit for language implementors. It offers efficient and flexible generation scavenging garbage collection. In addition to providing a core of language-independent algorithms and data structures, the toolkit includes auxiliary components that ease implementation of garbage collection for programming languages. We have detailed designs for Smalltalk and Modula-3 and are confident the toolkit
can be used with a wide variety of languages. The toolkit approach is itself novel, and our design includes a number of additional innovations in flexibility, efficiency, accuracy, and cooperation between the compiler and the collector.


Abstract
We present a garbage collection algorithm that extends generational scavenging to collect large older generations (mature objects) non-disruptively. The algorithm’s approach is to process bounded-size pieces of mature object space at each collection; the subtleties lie in guaranteeing that it eventually collects any and all garbage. The algorithm does not assume any special hardware or operating system support, e.g., for forwarding pointers or protection traps. The algorithm copies objects, so it naturally supports compaction and reclustering.


Abstract
A new garbage collection algorithm for distributed object systems, called DMOS (Distributed Mature Object Space), is presented. It is derived from two previous algorithms, MOS (Mature Object Space), sometimes called the train algorithm, and PMOS (Persistent Mature Object Space). The contribution of DMOS is that it provides the following unique combination of properties for a distributed collector: safety, completeness, non-disruptiveness, incrementality, and scalability. Furthermore, the DMOS collector is non-blocking and does not use global tracing.


Abstract
Dynamic memory use has been widely recognized to have profound effects on program performance, and has been the topic of many research studies over the last forty years. In spite of years of research, there is considerable confusion about the effects of dynamic memory allocation. Worse, this confusion is often unrecognized, and memory allocators are widely thought to be fairly well understood.

In this research, we attempt to clarify many issues for both manual and automatic non-moving memory management. We show that the traditional approaches to studying dynamic memory allocation are unsound, and develop a sound methodology for studying this problem. We present experimental evidence that fragmentation costs are much lower than previously recognized for most programs, and develop a framework for understanding these results and enabling further research in this area. For a large class of programs using well-known allocation policies, we show that fragmentation costs are near zero. We also study the locality effects of memory allocation on programs, a research area that has been almost completely ignored. We show that these effects can be quite dramatic, and that the best allocation policies in terms of fragmentation are also among the best in terms of locality at both the cache and virtual memory levels of the memory hierarchy.

We extend these fragmentation and locality results to real-time garbage collection. We have developed a hard real-time, non-copying generational garbage collector which uses a write-barrier to coordinate collection work only with modifications of pointers, therefore making coordination costs cheaper and more predictable than previous approaches. We combine this write-barrier approach with implicit non-copying reclamation, which has most of the advantages of copying collection (notably avoiding both the sweep phase required by mark-sweep collectors, and the referencing of garbage objects when reclaiming their space), without the disadvantage of having to actually copy the objects. In addition, we present a model for non-copying implicit-reclamation garbage collection. We use this model to compare and contrast our work with that of others, and to discuss the tradeoffs that must be made when developing such a garbage collector.

Abstract
We show that for 8 real and varied C and C++ programs, several conventional dynamic storage allocators provide near-zero fragmentation, once overheads due to implementation details (headers, alignment, etc.) are properly accounted for. This substantially strengthens our previous results showing that the memory fragmentation problem has generally been misunderstood, and that good allocator policies can provide good memory usage for most programs. The new results indicate that for most programs, excellent allocator policies are readily available, and efficiency of implementation is the major challenge. While we believe that our experimental results are state-of-the-art and our methodology is superior to most previous work, more work should be done to identify and study unusual problematic program behaviors not represented in our sample.


Abstract
The G-machine is a compiled graph reduction machine for lazy functional languages. The G-machine compiler contains many optimisations to improve performance. One set of such optimisations is designed to improve the performance of tail recursive functions. Unfortunately the abstract machine is subject to a space leak–objects are unnecessarily preserved by the garbage collector.

This paper analyses why a particular form of space leak occurs in the G-machine, and presents some ideas for fixing this problem. This phenomena in other abstract machines is also examined briefly.


Abstract
Weighted Reference Counting is a low-communication distributed storage reclamation scheme for loosely-coupled multiprocessors. The algorithm we present herein extends weighted reference counting to allow the collection of cyclic data structures. To do so, the algorithm identifies candidate objects that may be part of cycles and performs a tricolour mark-scan on their subgraph in a lazy manner to discover whether the subgraph is still in use. The algorithm is concurrent in the sense that multiple useful computation processes and garbage collection processes can be performed simultaneously.


From the back cover
The memory storage requirements of complex programs are extremely difficult to manage correctly by hand. A single error may lead to indeterminate and inexplicable program crashes. Worse still, failures are often unrepeatable and may surface only long after the program has been delivered to the customer. The eradication of memory errors typically consumes a substantial amount of development time. And yet the answer is relatively easy – garbage collection; removing the clutter of memory management from module interfaces, which then frees the programmer to concentrate on the problem at hand rather than low-level book-keeping details. For this reason, most modern object-oriented languages such as Smalltalk, Eiffel, Java and Dylan, are supported by garbage collection. Garbage collecting libraries are even available for such uncooperative languages as C and C++.

This book considers how dynamic memory can be recycled automatically to guarantee error-free memory management. There is an abundant but disparate literature on the subject, largely confined to research papers. This
book sets out to pool this experience in a single accessible and unified framework.

Each of the important algorithms is explained in detail, often with illustrations of its characteristic features and animations of its use. Techniques are described and compared for declarative and imperative programming styles, for sequential, concurrent and distributed architectures.

For professionals developing programs from simple software tools to complex systems, as well as for researchers and students working in compiler construction, functional, logic and object-oriented programming design, this book will provide not only a clear introduction but also a convenient reference source for modern garbage collection techniques.


From the Preface

The International Symposium on Memory Management is a forum for research in several related areas of memory management, especially garbage collectors and dynamic storage allocators. [...] The nineteen papers selected for publication in this volume cover a remarkably broad range of memory management topics from explicit malloc-style allocation to automatic memory management, from cache-conscious data layout to efficient management of distributed references, from conservative to type-accurate garbage collection, for applications ranging from user application to long-running servers, supporting languages as different as C, C++, Modula-3, Java, Eiffel, Erlang, Scheme, ML, Haskell and Prolog.


Abstract

The C++?? Critique is an analysis of some of the flaws of C++. It is by no means exhaustive, nor does it attempt to document every little niggle with C++, rather concentrating on main themes. The critique uses Java and Eiffel as comparisons to C++ to give a more concrete feel to the criticisms, viewing conceptual differences rather than syntactic ones as being more important. Some C++ authors realising there are glaring deficiencies in C++ have chosen to defend C++ by also being critical within their own work. Most notable are Bjarne Stroustup’s “Design and Evolution of C++,” and Scott Meyers’ “Effective” and “More Effective C++.” These warn of many traps and pitfalls, but reach the curious conclusion that since “good” C++ programmers are aware of these problems and know how to avoid them, C++ is alright.

The C++ critique makes many of the same criticisms, but comes to the different conclusion that these pitfalls are not acceptable, and should not be in a language used for modern large scale software engineering. Clean design is more important than after the fact warnings, and it is inconceivable that purchasers of end user software would tolerate this tactic on the part of vendors. The critique also takes a look at C, and concludes that many of the features of C should be left out of modern languages, and that C is a flawed base for a language.


Abstract

This paper studies a representative of an important class of emerging applications, a parallel data mining workload. The application, extracted from the IBM Intelligent Miner, identifies groups of records that are mathematically similar based on a neural network model called self-organizing map. We examine and compare in
details two implementations of the application: (1) temporal locality or working set sizes; (2) spatial locality and memory block utilization; (3) communication characteristics and scalability; and (4) TLB performance.

First, we find that the working set hierarchy of the application is governed by two parameters, namely the size of an input record and the size of prototype array; it is independent of the number of input records. Second, the application shows good spatial locality, with the implementation optimized for sparse data sets having slightly worse spatial locality. Third, due to the batch update scheme, the application bears very low communication. Finally, a 2-way set associative TLB may result in severely skewed TLB performance in a multiprocessor environment caused by the large discrepancy in the amount of conflict misses. Increasing the set associativity is more effective in mitigating the problem than increasing the TLB size.


Abstract

This paper studies the memory system behavior of Java programs by analyzing memory reference traces of several SPECjvm98 applications running with a Just-In-Time (JIT) compiler. Trace information is collected by an exception-based tracing tool called JTRACE, without any instrumentation to the Java programs or the JIT compiler. First, we find that the overall cache miss ratio is increased due to garbage collection, which suffers from higher cache misses compared to the application. We also note that going beyond 2-way cache associativity improves the cache miss ratio marginally. Second, we observe that Java programs generate a substantial amount of short-lived objects. However, the size of frequently-referenced long-lived objects is more important to the cache performance, because it tends to determine the application’s working set size. Finally, we note that the default heap configuration which starts from a small initial heap size is very inefficient since it invokes a garbage collector frequently. Although the direct costs of garbage collection decrease as we increase the available heap size, there exists an optimal heap size which minimizes the total execution time due to the interaction with the virtual memory performance.


Abstract

A stable heap is a storage that is managed automatically using garbage collection, manipulated using atomic transactions, and accessed using a uniform storage model. These features enhance reliability and simplify programming by preventing errors due to explicit deallocation, by masking failures and concurrency using transactions, and by eliminating the distinction between accessing temporary storage and permanent storage. Stable heap management is useful for programming language for reliable distributed computing, programming languages with persistent storage, and object-oriented database systems. Many applications that could benefit from a stable heap (e.g., computer-aided design, computer-aided software engineering, and office information systems) require large amounts of storage, timely responses for transactions, and high availability. We present garbage collection and recovery algorithms for a stable heap implementation that meet these goals and are appropriate for stock hardware. The collector is incremental: it does not attempt to collect the whole heap at once. The collector is also atomic: it is coordinated with the recovery system to prevent problems when it moves and modifies objects. The time for recovery is independent of heap size, and can be shortened using checkpoints.


Abstract

Prior work on dynamic memory allocation has largely neglected long-running server applications, for example, web servers and mail servers. Their requirements differ from those of one-shot applications like compilers or
text editors. We investigated how to build an allocator that is not only fast and memory efficient but also scales well on SMP machines. We found that it is not sufficient to focus on reducing lock contention. Only limited improvement can be achieved this way; higher speedups require a reduction in cache misses and cache invalidation traffic. We then designed and prototyped a new allocator, called Lkmalloc, targeted for both traditional applications and server applications. Lkmalloc uses several subheaps, each one with a separate set of free lists and memory arena. A thread always allocates from the same subheap but can free a block belonging to any subheap. A thread is assigned to a subheap by hashing on its thread ID. We compared its performance with several other allocators on a server-like, simulated workload and found that it indeed scales well and is quite fast but could use memory more efficiently.


Abstract

In previous heap storage systems, the cost of creating objects and garbage collection is independent of the lifetime of the object. Since objects with short lifetimes account for a large portion of storage use, it is worth optimizing a garbage collector to reclaim storage for these objects more quickly. The garbage collector should spend proportionately less effort reclaiming objects with longer lifetimes. We present a garbage collection algorithm that (1) makes storage for short-lived objects cheaper than storage for long-lived objects, (2) that operates in real-time—object creation and access times are bounded, (3) increases locality of reference, for better virtual memory performance, (4) works well with multiple processors and a large address space.

- J. McCarthy, M. L. Minsky. 1959. “Artificial Intelligence, Quarterly Progress Report no. 53”. Research Laboratory of Electronics at MIT.

Abstract

A programming system called LISP (for LISt Processor) has been developed for the IBM 704 computer by the Artificial Intelligence group at M.I.T. The system was designed to facilitate experiments with a proposed system called the Advice Taker, whereby a machine could be instructed to handle declarative as well as imperative sentences and could exhibit “common sense” in carrying out its instructions. The original proposal for the Advice Taker was made in November 1958. The main requirement was a programming system for manipulating expressions representing formalized declarative and imperative sentences so that the Advice Taker could make deductions.

In the course of its development the LISP system went through several stages of simplification and eventually came to be based on a scheme for representing the partial recursive functions of a certain class of symbolic expressions. This representation is independent of the IBM 704 computer, or of any other electronic computer, and it now seems expedient to expound the system by starting with the class of expressions called S-expressions and the functions called S-functions.


From the publisher’s catalog

Presents a survey of both distributed shared memory (DSM) efforts and commercial DSM systems. The book discusses relevant issues that make the concept of DSM one of the most attractive approaches for building large-scale, high-performance multiprocessor systems. Its text provides a general introduction to the DSM field as well as a broad survey of the basic DSM concepts, mechanisms, design issues, and systems.
Distributed Shared Memory concentrates on basic DSM algorithms, their enhancements, and their performance evaluation. In addition, it details implementations that employ DSM solutions at the software and the hardware level. The book is a research and development reference that provides state-of-the-art information that will be useful to architects, designers, and programmers of DSM systems.


Abstract

This paper presents an algorithm for reclaiming unused free storage memory cells in LISP. It depends on availability of a fast secondary storage device, or a large block of available temporary storage. For this price, we get
1. Packing of free-storage into a solidly packed block. 2. Smooth packing of arbitrary linear blocks and arrays. 3. The collector will handle arbitrarily complex re-entrant list structure with no introduction of spurious copies. 4. The algorithm is quite efficient; the marking pass visits words at most twice and usually once, and the loading pass is linear. 5. The system is easily modified to allow for increase in size of already fixed consecutive blocks, provide one can afford to initiate a collection pass or use a modified array while waiting for such a pass to occur.


Abstract

This paper discusses garbage collection techniques used in a high-performance Lisp implementation with a large virtual memory, the Symbolics 3600. Particular attention is paid to practical issues and experience. In a large system problems of scale appear and the most straightforward garbage-collection techniques do not work well. Many of these problems involve the interaction of the garbage collector with demand-paged virtual memory. Some of the solutions adopted in the 3600 are presented, including incremental copying garbage collection, approximately depth-first copying, ephemeral objects, tagged architecture, and hardware assists. We discuss techniques for improving the efficiency of garbage collection by recognizing that objects in the Lisp world have a variety of lifetimes. The importance of designing the architecture and the hardware to facilitate garbage collection is stressed.


Abstract

Massively distributed computing is a challenging problem for garbage collection algorithm designers as it raises the issue of scalability. The high number of hosts involved in a computation can require large tables for reference listing, whereas the lack of information sharing between hosts in a same locality can entail redundant GC traffic. In this paper, we argue that a conceptual hierarchical organisation of massive distributed computations can solve this problem. By conceptual hierarchical organisation, we mean that processors are still able to communicate in a peer to peer manner using their usual communication mechanism, but GC messages will be routed as if processors were organised in hierarchy. We present an extension of a distributed reference counting algorithm that uses such a hierarchical organisation. It allows us to bound table sizes by the number of hosts in a domain,
and it allows us to share GC information between hosts in a same locality in order to reduce cross-network GC traffic.


Abstract
Most specifications of garbage collectors concentrate on the low-level algorithmic details of how to find and preserve accessible objects. Often, they focus on bit-level manipulations such as “scanning stack frames,” “marking objects,” “tagging data,” etc. While these details are important in some contexts, they often obscure the more fundamental aspects of memory management: what objects are garbage and why?

We develop a series of calculi that are just low-level enough that we can express allocation and garbage collection, yet are sufficiently abstract that we may formally prove the correctness of various memory management strategies. By making the heap of a program syntactically apparent, we can specify memory actions as rewriting rules that allocate values on the heap and automatically dereference pointers to such objects when needed. This formulation permits the specification of garbage collection as a relation that removes portions of the heap without affecting the outcome of evaluation.

Our high-level approach allows us to specify in a compact manner a wide variety of memory management techniques, including standard trace-based garbage collection (i.e., the family of copying and mark/sweep collection algorithms), generational collection, and type-based, tag-free collection. Furthermore, since the definition of garbage is based on the semantics of the underlying language instead of the conservative approximation of inaccessibility, we are able to specify and prove the idea that type inference can be used to collect some objects that are accessible but never used.


Abstract
PMOS is an incremental garbage collector designed specifically to reclaim space in a persistent object store. It is complete in that it will, after a finite number of invocations, reclaim all unreachable storage. PMOS imposes minimum constraints on the order of collection and offers techniques to reduce the I/O traffic induced by the collector. Here we present the first implementation of the PMOS collector called PMOS#1. The collector has been incorporated into the stable heap layer of the generic persistent object store used to support a number of languages including Napier88. Our main design goals are to maintain the independence of the language from the store and to retain the existing store interface. The implementation has been completed and tested using a Napier88 system. The main results of this work show that the PMOS collector is implementable in a persistent store and that it can be built without requiring changes to the language interpreter. Initial performance measurements are reported. These results suggest however, that effective use of PMOS requires greater cooperation between language and store.


Abstract
We introduce a new replication-based copying garbage collection technique. We have implemented one simple variation of this method to provide incremental garbage collection on stock hardware with no special operating system or virtual memory support. The performance of the prototype implementation is excellent: major garbage collection pauses are completely eliminated with only a slight increase in minor collection pause times.

Unlike the standard copying algorithm, the replication-based method does not destroy the original replica when
a copy is created. Instead, multiple copies may exist, and various standard strategies for maintaining consistency may be applied. In our implementation for Standard ML of New Jersey, the mutator continues to use the from-space replicas until the collector has achieved a consistent replica of all live data in to-space.

We present a design for a concurrent garbage collector using the replication-based technique. We also expect replication-based GC methods to be useful in providing services for persistence and distribution, and briefly discuss these possibilities.


Abstract

Garbage collection (GC) is an important part of many language implementations. One of the most important garbage collection techniques is copying GC. This paper consists of an informal but abstract description of copying collection, a formal specification of copying collection written in the Larch Shared Language and the Larch/C Interface Language, a simple implementation of a copying collector written in C, an informal proof that the implementation satisfies the specification, and a discussion of how the specification applies to other types of copying GC such as generational copying collectors. Limited familiarity with copying GC or Larch is needed to read the specification.


Abstract

Orthogonal persistence provides a safe and convenient model of object persistence. We have implemented a transaction system which supports orthogonal persistence in a garbage-collected heap. In our system, replicating collection provides efficient concurrent garbage collection of the heap. In this paper, we show how replicating garbage collection can also be used to reduce commit operation latencies in our implementation.

We describe how our system implements transaction commit. We explain why the presence of non-persistent objects can add to the cost of this operation. We show how to eliminate these additional costs by using replicating garbage collection. The resulting implementation of orthogonal persistence should provide transaction performance that is independent of the quantity of non-persistent data in use. We expect efficient support for orthogonal persistence to be valuable in operating systems applications which use persistent data.


Abstract

We have implemented the first copying garbage collector that permits continuous unimpeded mutator access to the original objects during copying. The garbage collector incrementally replicates all accessible objects and uses a mutation log to bring the replicas up-to-date with changes made by the mutator. An experimental implementation demonstrates that the costs of using our algorithm are small and that bounded pause times of 50 milliseconds can be readily achieved.


Abstract

This paper investigates the performance of 35 dynamic memory allocation algorithms when used to service simulation programs as represented by 18 test cases. Algorithm performance was measured in terms of processing time, memory usage, and external memory fragmentation. Algorithms maintaining separate free space lists for each size of memory block used tended to perform quite well compared with other algorithms. Simple algorithms operating on memory ordered lists (without any free list) performed surprisingly well. Algorithms
employing power-of-two block sizes had favorable processing requirements but generally unfavorable memory usage. Algorithms employing LIFO, FIFO, or memory ordered free lists generally performed poorly compared with others.

• James O’Toole. 1990. “Garbage Collecting Locally”.

Abstract
Generational garbage collection is a simple technique for automatic partial memory reclamation. In this paper, I present the basic mechanics of generational collection and discuss its characteristics. I compare several published algorithms and argue that fundamental considerations of locality, as reflected in the changing relative speeds of processors, memories, and disks, strongly favor a focus on explicit optimization of I/O requirements during garbage collection. I show that this focus on I/O costs due to memory hierarchy debunks a well-known claim about the relative costs of garbage collection and stack allocation. I suggest two directions for future research in this area and discuss some simple architectural changes in virtual memory interfaces which may enable efficient garbage collector utilization of standard virtual memory hardware.


Abstract
We have implemented a concurrent copying garbage collector that uses replicating garbage collection. In our design, the client can continuously access the heap during garbage collection. No low-level synchronization between the client and the garbage collector is required on individual object operations. The garbage collector replicates live heap objects and periodically synchronizes with the client to obtain the client’s current root set and mutation log. An experimental implementation using the Standard ML of New Jersey system on a shared-memory multiprocessor demonstrates excellent pause time performance and moderate execution time speedups.


Abstract
For a compiler writer, generating good machine code for a variety of platforms is hard work. One might try to reuse a retargetable code generator, but code generators are complex and difficult to use, and they limit one’s choice of implementation language. One might try to use C as a portable assembly language, but C limits the compiler writer’s flexibility and the performance of the resulting code. The wide use of C, despite these drawbacks, argues for a portable assembly language. C– is a new language designed expressly for this purpose. The use of a portable assembly language introduces new problems in the support of such high-level run-time services as garbage collection, exception handling, concurrency, profiling, and debugging. We address these problems by combining the C– language with a C– run-time interface. The combination is designed to allow the compiler writer a choice of source-language semantics and implementation techniques, while still providing good performance.


Abstract
Software caching, automatic algorithm blocking, and data overlays are different names for the same problem: compiler management of data movement throughout the memory hierarchy. Modern high-performance architectures often omit hardware support for moving data between levels of the memory hierarchy: iWarp does not include a data cache, and Cray supercomputers do not have virtual memory. These systems have effectively traded a more complicated programming model for performance by replacing a hardware-controlled memory
hierarchy with a simple fast memory. The simpler memories have less logic in the critical path, so the cycle time of the memories is improved.

For programs which fit in the resulting memory, the extra performance is great. Unfortunately, the driving force behind supercomputing today is a class of very large scientific problems, both in terms of computation time and in terms of the amount of data used. Many of these programs do not fit in the memory of the machines available. When architects trade hardware support for data migration to gain performance, control of the memory hierarchy is left to the programmer. Either the program size must be cut down to fit into the machine, or every loop which accesses more data than will fit into memory must be restructured by hand. This thesis describes how a compiler can relieve the programmer of this burden, and automate data motion throughout the memory hierarchy without direct hardware support.

This works develops a model of how data is accessed within a nested loop by typical scientific programs. It describes techniques which can be used by compilers faced with the task of managing data motion. The concentration is on nested loops which process large data arrays using linear array subscripts. Because the array subscripts are linear functions of the loop indices and the loop indices form an integer lattice, linear algebra can be applied to solve many compilation problems.

The approach it to tile the iteration space of the loop nest. Tiling allows the compiler to improve locality of reference. The tiling basis matrix is chosen from a set of candidate vectors which neatly divide the data set. The execution order of the tiles is selected to maximize locality between tiles. Finally, the tile sizes are chosen to minimize execution time.

The approach has been applied to several common scientific loop nests: matrix-matrix multiplication, QR-decomposition, and LU-decomposition. In addition, an illustrative example from the Livermore Loop benchmark set is examined. Although more compiler time can be required in some cases, this technique produces better code at no cost for most programs.


Abstract
This paper presents a classification of barrier techniques for interleaving tracing with mutator operation during an incremental garbage collection. The two useful tricolour invariants are derived from more elementary considerations of graph traversal. Barrier techniques for maintaining these invariants are classified according to the action taken at the barrier (such as scanning an object or changing its colour), and it is shown that the algorithms described in the literature cover all the possibilities except one. Unfortunately, the new technique is impractical. Ways of combining barrier techniques are also discussed.


Abstract
This paper presents work currently in progress on Disk Garbage Collection issues for PJava, an orthogonally persistent version of Java. In particular, it concentrates on the initial Prototype of the Disk Garbage Collector of PJava0 which has already been implemented. This Prototype was designed to be very simple and modular in order to be easily changed, evolved, improved, and allow experimentation. Several experiments were performed in order to test possible optimisations; these experiments concentrated on the following four areas: a) efficient access to the store; b) page-replacement algorithms; c) efficient discovery of live objects during compaction; and d) dealing with forward references. The paper presents a description of the Prototype’s architecture, the results of these experiments and related discussion, and some future directions based on the experience gained from this work.

- Tony Printezis & Quentin Cutts. 1996. “Measuring the Allocation Rate of Napier88”. Department of Computing Science at University of Glasgow. TR ?.
Abstract

As processor speeds continue to improve relative to main-memory access times, cache performance is becoming an increasingly important component of program performance. Prior work on the cache performance of garbage-collected programming languages has either assumed or argued that conventional garbage-collection methods will yield poor performance, and has therefore concentrated on new collection algorithms designed specifically to improve cache-level reference locality. This dissertation argues to the contrary: Many programs written in garbage-collected languages are naturally well-suited to the direct-mapped caches typically found in modern computer systems.

Using a trace-driven cache simulator and other analysis tools, five nontrivial, long-running Scheme programs are studied. A control experiment shows that the programs have excellent cache performance without any garbage collection at all. A second experiment indicates that the programs will perform well with a simple and infrequently-run generational compacting collector.

An analysis of the test programs’ memory usage patterns reveals that the mostly-functional programming style typically used in Scheme programs, in combination with simple linear storage allocation, causes most data objects to be dispersed in time and space so that references to them cause little cache interference. From this it follows that other Scheme programs, and programs written in similar styles in different languages, should perform well with a simple generational compacting collector; sophisticated collectors intended to improve cache performance are unlikely to be effective. The analysis also suggests that, as locality becomes ever more important to program performance, programs written in garbage-collected languages may turn out to have significant performance advantage over programs written in more conventional languages.

Abstract

It is well accepted that automatic garbage collection simplifies programming, promotes modularity, and reduces development effort. However it is commonly believed that these advantages do not counteract the perceived price: excessive overheads, possible long pause times while garbage collections occur, and the need to modify existing code. Even though there are publically available garbage collector implementations that can be used in existing programs, they do not guarantee short pauses, and some modification of the application using them is still required. In this paper we describe a snapshot-at-beginning concurrent garbage collector algorithm and its implementation. This algorithm guarantees short pauses, and can be easily implemented on stock UNIX-like operating systems. Our results show that our collector performs comparable to other garbage collection implementations on uniprocessor machines and outperforms similar collectors on multiprocessor machines. We also show our collector to be competitive in performance with explicit deallocation. Our collector has the added advantage of being non-intrusive. Using a dynamic linking technique and effective root set inferencing, we have been able to successfully run our collector even in commercial programs where only the binary executable and no source code is available. In this paper we describe our algorithm, its implementation, and provide both an algorithmic and a performance comparison between our collector and other similar garbage collectors.

Self-compiling implementations of Haskell, i.e., those written in Haskell, have been and, except one, are still space consuming monsters. Object code size for the compilers themselves are 3-8Mb, and they need 12-20Mb
to recompile themselves. One reason for the huge demands for memory is that the main goal for these compilers is to produce fast code. However, the compiler described in this paper, called “nhc” for “Nearly a Haskell Compiler”, is the one above mentioned exception. This compiler concentrates on keeping memory usage down, even at a cost in time. The code produced is not fast, but nhc is usable, and the resulting programs can be run on computers with small memory.

This paper describes some of the implementation choices done, in the Haskell part of the source code, to reduce memory consumption in nhc. It is possible to use these also in other Haskell compilers with no, or very small, changes to their run-time systems.

Time is neither the main focus of nhc nor of this paper, but there is nevertheless a small section about the speed of nhc. The most notable observation concerning speed is that nhc spends approximately half the time processing interface files, which is much more than needed in the type checker. Processing interface files is also the most space consuming part of nhc in most cases. It is only when compiling source files with large sets of mutually recursive functions that more memory is needed to type check than to process interface files.


**Abstract**

Generational garbage collection is an established method for creating efficient garbage collectors. Even a simple implementation where all nodes that survive one garbage collection are *tenured*, i.e., moved to an old generation, works well in strict languages. In lazy languages, however, such an implementation can create severe *temporary space leaks*. The temporary space leaks appear in programs that traverse large lazily built data structures, e.g., a lazy list representing a large file, where only a small part is needed at any time. A simple generational garbage collector cannot reclaim the memory, used by the lazily built list, at minor collections. The reason is that at least one of the nodes in the list belongs to the old generation, after the first minor collection, and will hold on to the rest of the nodes in the list until the next major collection.


**Abstract**

The context for this paper is functional computation by graph reduction. Our overall aim is more efficient use of memory. The specific topic is the detection of dormant cells in the live graph – those retained in heap memory though not actually playing a useful role in computation. We describe a profiler that can identify heap consumption by such ‘useless’ cells. Unlike heap profilers based on traversals of the live heap, this profiler works by examining cells post-mortem. The new profiler has revealed a surprisingly large proportion of ‘useless’ cells, even in some programs that previously seemed space-efficient such as the bootstrapping Haskell compiler “nhc”.


**Abstract**

Stoye’s one-bit reference tagging scheme can be extended to local counts of two or more via two strategies. The first, suited to pure register transactions, is a cache of referents to two shared references. The analog of Deutch’s and Bobrow’s multiple-reference table, this cache is sufficient to manage small counts across successive assignment statements. Thus, accurate reference counts above one can be tracked for short intervals, like that bridging one function’s environment to its successor’s.

The second, motivated by runtime stacks that duplicate references, avoids counting any references from the stack. It requires a local pointer-inversion protocol in the mutator, but one still local to the referent and the stack frame. Thus, an accurate reference count of one can be maintained regardless of references from the recursion stack.
Abstract

Enough is known now about garbage collection, runtime types, strong-typing, static-checking and concurrency that it is possible to explore what happens when they are combined in a real programming system.

Storage management is one of a few central issues through which one can get a good view of the design of an entire system. Tensions between ease of integration and the need for protection; between generality, simplicity, flexibility, extensibility and efficiency are all manifest when assumptions and attitudes about managing storage are studied. And deep understanding follows best from the analysis of systems that people use to get real work done.

This paper is not for those who seek arguments pro or con about the need for these features in programming systems; such issues are for other papers. This one assumes these features to be good and describes how they combine and interact in Cedar, a programming language and environment designed to help programmers build moderate-sized experimental systems for moderate numbers of people to test and use.


Abstract

We describe the design, implementation, and use of a new kind of profiling tool that yields valuable information about the memory use of lazy functional programs. The tool has two parts: a modified functional language implementation which generated profiling implementation during the execution of programs, and a separate program which converts this information to graphical form. With the aid of profile graphs, one can make alterations to a functional program which dramatically reduce its space consumption. We demonstrate that this is the case of a genuine example – the first to which the tool has been applied – for which the results are strikingly successful.


Abstract

First-generation heap profilers for lazy functional languages have proved to be effective tools for locating some kinds of space faults, but in other cases they cannot provide sufficient information to solve the problem. This paper describes the design, implementation and use of a new profiler that goes beyond the two-dimensional “who produces what” view of heap cells to provide information about their more dynamic and structural attributes. Specifically, the new profiler can distinguish between cells according to their eventual lifetime, or on the basis of the closure retainers by virtue of which they remain part of the live heap. A bootstrapping Haskell compiler (nhc) hosts the implementation: among examples of the profiler’s use we include self-application to nhc. Another example is the original heap-profiling case study “clausify”, which now consumes even less memory and is much faster.


Abstract

A heap profile is a chart showing the contents of heap memory throughout a computation. Contents are depicted abstractly by showing how much space is occupied by memory cells in each of several classes. A good heap profiler can use a variety of attributes of memory cells to define a classification. Effective profiling usually involves a combination of attributes. The ideal profiler gives full support for combination in two ways. First, a section of the heap of interest to the programmer can be specified by constraining the values of any combination...
of cell attributes. Secondly, no matter what attributes are used to specify such a section, a heap profile can be obtained for that section only, and any other attribute can be used to define the classification.

Achieving this ideal is not simple. For some combinations of attributes, A heap profile is derived by interpolation of a series of censuses of heap contents at different stages. The obvious way to obtain census data is to traverse the live heap at intervals throughout the computation. This is fine for static attributes (e.g., What type of value does this memory cell represent?), and for dynamic attributes that can be determined for each cell by examining the heap at any given moment (e.g., From which function closures can this cell be reached?). But some attributes of cells can only be determined retrospectively by post-mortem inspection as a cell is overwritten or garbage-collected (e.g., Is this cell ever used again?). Now we see the problem: if a profiler supports both live and post-mortem attributes, how can we implement the ideal of unrestricted combinations? That is the problem we solve in this paper. We give techniques for profiling a heap section specified in terms of both live and post-mortem attributes, and vice versa.


Abstract
We present an implementation of the Train Algorithm, an incremental collection scheme for reclamation of mature garbage in generation-based memory management systems. To the best of our knowledge, this is the first Train Algorithm implementation ever. Using the algorithm, the traditional mark-sweep garbage collector employed by the Mjølner run-time system for the object-oriented BETA programming language was replaced by a non-disruptive one, with only negligible time and storage overheads.


Abstract
Memory is the performance bottleneck of modern architectures. Keeping memory consumption as low as possible enables fast and unobtrusive applications. But it is not easy to estimate the memory use of programs implemented in functional languages, due to both the complex translations of some high level constructs, and the use of automatic memory managers. To help understand memory allocation behavior of Scheme programs, we have designed two complementary tools. The first one reports on frequency of allocation, heap configurations and on memory reclamation. The second tracks down memory leaks. We have applied these tools to our Scheme compiler, the largest Scheme program we have been developing. This has allowed us to drastically reduce the amount of memory consumed during its bootstrap process, without requiring much development time. Development tools will be neglected unless they are both conveniently accessible and easy to use. In order to avoid this pitfall, we have carefully designed the user interface of these two tools. Their integration into a real programming environment for Scheme is detailed in the paper.


Abstract
Larchant-RDOSS is a distributed shared memory that persists on reliable storage across process lifetimes. Memory management is automatic: including consistent caching of data and of locks, collecting objects unreachable from the persistent root, writing reachable objects to disk, and reducing store fragmentation. Memory management is based on a novel garbage collection algorithm, that approximates a global trace by a series of local traces, with no induced I/O or locking traffic, and no synchronization between the collector and the application processes. This results in a simple programming model, and expected minimal added application latency. The
algorithm is designed for the most unfavorable environment (uncontrolled programming language, reference by pointers, distributed system, non-coherent shared memory) and should work well also in more favorable settings.


Abstract

Texas is a persistent storage system for C++, providing high performance while emphasizing simplicity, modularity and portability. A key component of the design is the use of pointer swizzling at page fault time, which exploits existing virtual memory features to implement large address spaces efficiently on stock hardware, with little or no change to existing compilers. Long pointers are used to implement an enormous address space, but are transparently converted to the hardware-supported pointer format when pages are loaded into virtual memory.

Runtime type descriptors and slightly modified heap allocation routines support pagewise pointer swizzling by allowing objects and their pointer fields to be identified within pages. If compiler support for runtime type identification is not available, a simple preprocessor can be used to generate type descriptors.

This address translation is largely independent of issues of data caching, sharing, and checkpointing; it employs operating systems’ existing virtual memories for caching, and a simple and flexible log-structured storage manager to improve checkpointing performance.

Pagewise virtual memory protections are also used to detect writes for logging purposes, without requiring any changes to compiled code. This may degrade checkpointing performance for small transactions with poor locality of writes, but page diffing and sub-page logging promise to keep performance competitive with finer-grained checkpointing schemes.

Texas presents a simple programming interface; an application creates persistent objects by simply allocating them on the persistent heap. In addition, the implementation is relatively small, and is easy to incorporate into existing applications. The log-structured storage module easily supports advanced extensions such as compressed storage, versioning, and adaptive reorganization.


Abstract

Garbage collector performance in LISP systems on custom hardware has been substantially improved by the adoption of lifetime-based garbage collection techniques. To date, however, successful lifetime-based garbage collectors have required special-purpose hardware, or at least privileged access to data structures maintained by the virtual memory system. I present here a lifetime-based garbage collector requiring no special-purpose hardware or virtual memory system support, and discuss its performance.


Abstract

Algorithms for a multiprocessing compactifying garbage collector are presented and discussed. The simple case of two processors, one performing LISP-like list operations and the other performing garbage collection continuously, is thoroughly examined. The necessary capabilities of each processor are defined, as well as interprocessor communication and interlocks. Complete procedures for garbage collection and for standard list
processing primitives are presented and thoroughly explained. Particular attention is given to the problems of marking and relocating list cells while another processor may be operating on them. The primary aim throughout is to allow the list processor to run unimpeded while the other processor reclaims list storage. The more complex case involving several list processors and one or more garbage collection processors are also briefly discussed.


Abstract
The internal representations of the various MacLISP data types are presented and discussed. Certain implementation tradeoffs are considered. The ultimate decisions on these tradeoffs are discussed in the light of MacLISP’s prime objective of being an efficient high-level language for the implementation of large systems such as MACSYMA. The basic strategy of garbage collection is outlined, with reference to the specific representations involved. Certain “clever tricks” are explained and justified. The “address space crunch” is explained and some alternative solutions explored.


Abstract
A high-performance implementation of a Java Virtual Machine requires a compiler to translate Java bytecodes into native instructions, as well as an advanced garbage collector (e.g., copying or generational). When the Java heap is exhausted and the garbage collector executes, the compiler must report to the garbage collector all live object references contained in physical registers and stack locations. Typical compilers only allow certain instructions (e.g., call instructions and backward branches) to be GC-safe; if GC happens at some other instruction, the compiler may need to advance execution to the next GC-safe point. Until now, no one has ever attempted to make every compiler-generated instruction GC-safe, due to the perception that recording this information would require too much space. This kind of support could improve the GC performance in multithreaded applications. We show how to use simple compression techniques to reduce the size of the GC map to about 20% of the generated code size, a result that is competitive with the best previously published results. In addition, we extend the work of Agesen, Detlefs, and Moss, regarding the so-called “JSR Problem” (the single exception to Java’s type safety property), in a way that eliminates the need for extra runtime overhead in the generated code.


Abstract
The SKIM II processor is a microcoded hardware machine for the rapid evaluation of functional languages. This paper gives details of some of the more novel methods employed by SKIM II, and resulting performance measurements. The authors conclude that combinator reduction can still form the basis for the efficient implementation of a functional language.


Abstract
We study the cost of storage management for garbage-collected programs compiled with the Standard ML of New Jersey compiler. We show that the cost of storage management is not the same as the time spent garbage
collecting. For many of the programs, the time spent garbage collecting is less than the time spent doing other
storage-management tasks.


Abstract

Shared environment closure reducers such as Fairbairn and Wray’s TIM incur a comparatively low cost when
creating a suspension, and so provide an elegant method for implementing lazy functional evaluation. However,
comparatively little attention has been given to the problems involved in identifying which portions of a shared
environment are needed (and ignoring those which are not) during a garbage collection. Proper consideration
of this issue has subtle consequences when implementing a storage manager in a TIM-like system. We describe
the problem and illustrate the negative consequences of ignoring it.

We go on to describe a solution in which the compiler determines statically which portions of that code’s
environment are required for each piece of code it generates, and emits information to assist the run-time storage
manager to scavenge environments selectively. We also describe a technique for expressing this information
directly as executable code, and demonstrate that a garbage collector implemented in this way can perform
significantly better than an equivalent, table-driven interpretive collector.


Abstract

Implementations of abstract machines such as the OP-TIM and the PG-TIM need to use a tailored garbage
collector which seems to require an auxiliary stack, with a potential maximum size that is directly proportional
to the amount of live data in the heap. However, it turns out that it is possible to build a recursive copying
collector that does not require additional space by reusing already-scavenged space. This paper is a description
of this technique.


Abstract

This paper describes a memory management discipline for programs that perform dynamic memory allocation
and de-allocation. At runtime, all values are put into regions. The store consists of a stack of regions. All
points of region allocation and de-allocation are inferred automatically, using a type and effect based program
analysis. The scheme does not assume the presence of a garbage collector. The scheme was first presented in
Programming Languages, pp. 188–201); subsequently, it has been tested in the ML Kit with Regions, a region-
based, garbage-collection free implementation of the Standard ML Core Language, which includes recursive
datatypes, higher-order functions and updatable references (L. Birkedal, M. Tofte, and M. Vejlstrup, (1996), in
171–183). This paper defines a region-based dynamic semantics for a skeletal programming language extracted
from Standard ML. We present the inference system which specifies where regions can be allocated and de-
allocated and a detailed proof that the system is sound with respect to a standard semantics. We conclude by
giving some advice on how to write programs that run well on a stack of regions, based on practical experience
with the ML Kit.

- Dave Ungar. 1984. “Generation Scavenging: A Non-disruptive High Performance Storage Reclamation Algo-
rithm”. ACM, SIGSOFT, SIGPLAN. Practical Programming Environments Conference.
Abstract

Many interactive computing environments provide automatic storage reclamation and virtual memory to ease the burden of managing storage. Unfortunately, many storage reclamation algorithms impede interaction with distracting pauses. *Generation Scavenging* is a reclamation algorithm that has no noticeable pauses, eliminates page faults for transient objects, compacts objects without resorting to indirection, and reclaims circular structures, in one third the time of traditional approaches.


Abstract

One of the most promising automatic storage reclamation techniques, generation-based storage reclamation, suffers poor performance if many objects live for a fairly long time and then die. We have investigated the severity of the problem by simulating Generation Scavenging automatic storage reclamation from traces of actual four-hour sessions. There was a wide variation in the sample runs, with garbage-collection overhead ranging from insignificant, during interactive runs, to sever, during a single non-interactive run. All runs demonstrated that performance could be improved with two techniques: segregating large bitmaps and strings, and mediating tenuring with demographic feedback. These two improvements deserve consideration for any generation-based storage reclamation strategy.


Abstract

On C/Unix systems, the malloc interface is standard for dynamic memory allocation. Despite its popularity, malloc’s shortcomings frequently cause programmers to code around it. The new library Vmalloc generalizes malloc to give programmers more control over memory allocation. Vmalloc introduces the idea of organizing memory into separate regions, each with a discipline to get raw memory and a method to manage allocation. Applications can write their own disciplines to manipulate arbitrary type of memory or just to better organize memory in a region by creating new regions out of its memory. The provided set of allocation methods include general purpose allocations, fast special cases and aids for memory debugging or profiling. A compatible malloc interface enables current applications to select allocation methods using environment variables so they can tune for performance or perform other tasks such as profiling memory usage, generating traces of allocation calls or debugging memory errors. A performance study comparing Vmalloc and currently popular malloc implementations shows that Vmalloc is competitive to the best of these allocators. Applications can gain further performance improvement by using the right mixture of regions with different Vmalloc methods.


Abstract

Garwick’s algorithm, for repacking LIFO lists stored in a contiguous block of memory, bases the allocation of remaining space upon both sharing and previous stack growth. A system whereby the weight applied to each method can be adjusted according to the current behaviour of the stacks is discussed.

We also investigate the problem of determining during memory repacking that the memory is used to saturation and the driving program should therefore be aborted. The tuning parameters studied here seem to offer no new grasp on this problem.

Abstract
GC systems allocate and reuse memory cyclically; this imposes a cyclic pattern on memory accesses that has its own distinctive locality characteristics. The cyclic reuse of memory tends to defeat caching strategies if the reuse cycle is too large to fit in fast memory. Generational GCs allow a smaller amount of memory to be reused more often. This improves VM performance, because the frequently-reused area stays in main memory. The same principle can be applied at the level of high-speed cache memories, if the cache is larger than the youngest generation. Because of the repeated cycling through a fixed amount of memory, however, generational GC interacts with cache design in unusual ways, and modestly set-associative caches can significantly outperform direct-mapped caches.

While our measurements do not show very high miss rates for GCed systems, they indicate that performance problems are likely in faster next-generation systems, where second-level cache misses may cost scores of cycles. Software techniques can improve cache performance of garbage-collected systems, by decreasing the cache “footprint” of the youngest generation; compiler techniques that reduce the amount of heap allocation also improve locality. Still, garbage-collected systems with a high rate of heap allocation require somewhat more cache capacity and/or main memory bandwidth than conventional systems.


Abstract
Pointer swizzling at page fault time is a novel address translation mechanism that exploits conventional address translation hardware. It can support huge address spaces efficiently without long hardware addresses; such large address spaces are attractive for persistent object stores, distributed shared memories, and shared address space operating systems. This swizzling scheme can be used to provide data compatibility across machines with different word sizes, and even to provide binary code compatibility across machines with different hardware address sizes.

Pointers are translated (“swizzled”) from a long format to a shorter hardware-supported format at page fault time. No extra hardware is required, and no continual software overhead is incurred by presence checks of indirection of pointers. This pagewise technique exploits temporal and spatial locality in much the same way as normal virtual memory; this gives it many desirable performance characteristics, especially given the trend toward larger main memories. It is easy to implement using common compilers and operating systems.


Abstract
We survey basic garbage collection algorithms, and variations such as incremental and generational collection; we then discuss low-level implementation considerations and the relationships between storage management systems, languages, and compilers. Throughout, we attempt to present a unified view based on abstract traversal strategies, addressing issues of conservatism, opportunism, and immediacy of reclamation; we also point out a variety of implementation details that are likely to have a significant impact on performance.


Abstract
Dynamic memory allocation has been a fundamental part of most computer systems since roughly 1960, and memory allocation is widely considered to be either a solved problem or an insoluble one. In this survey, we describe a variety of memory allocator designs and point out issues relevant to their design and evaluation. We then chronologically survey most of the literature on allocators between 1961 and 1995. (Scores of papers are discussed, in varying detail, and over 150 references are given.)
We argue that allocator designs have been unduly restricted by an emphasis on mechanism, rather than policy, while the latter is more important; higher-level strategic issues are still more important, but have not been given much attention.

Most theoretical analyses and empirical allocator evaluations to date have relied on very strong assumptions of randomness and independence, but real program behavior exhibits important regularities that must be exploited if allocators are to perform well in practice.


Abstract

A new buddy system is described in which the region of storage being managed is partitioned into two sub-regions, each managed by a fairly standard “binary” buddy system. Like the weighted buddy systems of Shen and Peterson, the block sizes are of sizes $2^{n+1}$ or $3\cdot2^n$, but unlike theirs there is no extra overhead for typing information or for buddy calculation, and an allocation which requires splitting an extant available block only rarely creates a block smaller than the one being allocated. Such smaller blocks are carved out only when the boundary between the two subregions floats; the most interesting property of this system is that the procedures for allocation and deallocation are designed to keep blocks immediately adjacent to the subregion boundary free, so that the boundary may be moved within a range of unused space without disturbing blocks in use. This option is attained with a minimum of extra computation beyond that of a binary buddy system, and provides this scheme with a new approach to the problem of external fragmentation.


Abstract

The two-pass compaction algorithm of F.L. Morris, which follows upon the mark phase in a garbage collector, may be modified to recover reference counts for a hybrid storage management system. By counting the executions of two loops in that algorithm where upward and downward references, respectively, are forwarded to the relocation address of one node, we can initialize a count of active references and then update it but once. The reference count may share space with the mark bit in each node, but it may not share the additional space required in each pointer by Morris’s algorithm, space which remains unused outside the garbage collector.


Abstract

A project to design a pair of memory chips with a modicum of intelligence is described. Together, the two allow simple fabrication of a small memory bank, a heap of binary (LISP-like) nodes that offers the following features: 64-bit nodes; two pointer fields per node up to 29 bits each; reference counts implicitly maintained on writes; 2 bits per node for marking (uncounted) circular references; 4 bits per node for conditional-store testing at the memory; provision for processor-driven, recounting garbage collection.


Abstract

A stop-and-copy garbage collector updates one-bit reference counting with essentially no extra space and minimal memory cycles beyond the conventional collection algorithm. Any object that is uniquely referenced during a collection becomes a candidate for cheap recovery before the next one, or faster recopying then if it remains...
uniquely referenced. Since most objects stay uniquely referenced, subsequent collections run faster even if none are recycled between garbage collections. This algorithm extends to generation scavenging, it admits uncounted references from roots, and it corrects conservatively stuck counters, that result from earlier uncertainty whether references were unique.


Abstract

Identifying some pointers as invisible threads, for the purposes of storage management, is a generalization from several widely used programming conventions, like threaded trees. The necessary invariant is that nodes that are accessible (without threads) emit threads only to other accessible nodes. Dynamic tagging or static typing of threads ameliorates storage recycling both in functional and imperative languages.

We have seen the distinction between threads and links sharpen both hardware- and software-supported storage management in SCHEME, and also in C. Certainly, therefore, implementations of languages that already have abstract management and concrete typing, should detect and use this as a new static type.


Abstract

A hardware self-managing heap memory (RCM) for languages like LISP, SMALLTALK, and JAVA has been designed, built, tested and benchmarked. On every pointer write from the processor, reference-counting transactions are performed in real time within this memory, and garbage cells are reused without processor cycles. A processor allocates new nodes simply by reading from a distinguished location in its address space. The memory hardware also incorporates support for off-line, multiprocessing, mark-sweep garbage collection.

Performance statistics are presented from a partial implementation of SCHEME over five different memory models and two garbage collection strategies, from main memory (no access to RCM) to a fully operational RCM installed on an external bus. The performance of the RCM memory is more than competitive with main memory.


Abstract

A group at Symbolics is developing a Lisp runtime kernel, derived from its Genera operating system, to support real-time control applications. The first candidate application has strict response-time requirements (so strict that it does not permit the use of paged virtual memory). Traditionally, Lisp’s automatic storage-management mechanism has made it unsuitable to real-time systems of this nature. A number of garbage collector designs and implementations exist (including the Genera garbage collector) that purport to be “real-time”, but which actually have only mitigated the impact of garbage collection sufficiently that it usually goes unnoticed by humans. Unfortunately, electro-mechanical systems are not so forgiving. This paper examines the limitations of existing real-time garbage collectors and describes the avenues that we are exploring in our work to develop a CLOS-based garbage collector that can meet the real-time requirements of real real-time systems.


Abstract
The thesis of this project is that incremental collection can be done feasibly and efficiently in an architecture and compiler independent manner. The design and implementation of an incremental, generational mostly-copying garbage collector for C++ is presented. The collector achieves, simultaneously, real-time performance (from incremental collection), low total garbage collection delay (from generational collection), and the ability to function without hardware and compiler support (from mostly-copying collection).

The incremental collector runs on commercially-available uniprocessors, such as the DECStation 3100, without any special hardware support. It uses UNIX’s user controllable page protection facility (mprotect) to synchronize between the scanner (of the collector) and the mutator (of the application program). Its implementation does not require any modification to the C++ compiler. The maximum garbage collection pause is well within the 100-millisecond limit imposed by real-time applications executing on interactive workstations. Compared to its non-incremental version, the total execution time of the incremental collector is not adversely affected.


**Abstract**

An algorithm for real-time garbage collection is presented, proved correct, and evaluated. This algorithm is intended for list-processing systems on general-purpose machines, i.e., Von Neumann style serial computers with a single processor. On these machines, real-time garbage collection inevitably causes some overhead on the overall execution of the list-processing system, because some of the primitive list-processing operations must check the status of garbage collection. By removing such overhead from frequently used primitives such as pointer references (e.g., Lisp car and cdr) and stack manipulations, the presented algorithm reduces the execution overhead to a great extent. Although the algorithm does not support compaction of the whole data space, it efficiently supports partial compaction such as array relocation.


**Abstract**

This paper describes inprof, a tool used to study the memory allocation behavior of programs. mprof records the amount of memory each function allocates, breaks down allocation information by type and size, and displays a program’s dynamic call graph so that functions indirectly responsible for memory allocation are easy to identify. mprof is a two-phase tool. The monitor phase is linked into executing programs and records information each time memory is allocated. The display phase reduces the data generated by the monitor and displays the information to the user in several tables. mprof has been implemented for C and Kyoto Common Lisp. Measurements of these implementations are presented.


**Abstract**

This thesis shows that object-level, trace-driven simulation can facilitate evaluation of language runtime systems and reaches new conclusions about the relative performance of important garbage collection algorithms. In particular, I reach the unexpected conclusion that mark-and-sweep garbage collection, when augmented with generations, shows comparable CPU performance and much better reference locality than the more widely used copying algorithms. In the past, evaluation of garbage collection algorithms has been limited by the high cost of implementing the algorithms. Substantially different algorithms have rarely been compared in a systematic way.

With the availability of high-performance, low-cost workstations, trace-driven performance evaluation of these algorithms is now economical. This thesis describes MARS, a runtime system simulator that is driven by op-
operations on program objects, and not memory addresses. MARS has been attached to a commercial Common Lisp system and eight large Lisp applications are used in the thesis as test programs. To illustrate the advantages of the object-level tracing technique used by MARS, this thesis compares the relative performance of stop-and-copy, incremental, and mark-and-sweep collection algorithms, all organized with multiple generations. The comparative evaluation is based on several metrics: CPU overhead, reference locality, and interactive availability.

Mark-and-sweep collection shows slightly higher CPU overhead than stop-and-copy ability (5 percent), but requires significantly less physical memory to achieve the same page fault rate (30-40 percent). Incremental collection has very good interactive availability, but implementing the read barrier on stock hardware incurs a substantial CPU overhead (30-60 percent). In the future, I will use MARS to investigate other performance aspects of sophisticated runtime systems.


Abstract
Stop-and-copy garbage collection has been preferred to mark-and-sweep collection in the last decade because its collection time is proportional to the size of reachable data and not to the memory size. This paper compares the CPU overhead and the memory requirements of the two collection algorithms extended with generations, and finds that mark-and-sweep collection requires at most a small amount of additional CPU overhead (3-6%) but requires an average of 20% (and up to 40%) less memory to achieve the same page fault rate. The comparison is based on results obtained using trace-driven simulation with large Common Lisp programs.


Abstract
Garbage collection algorithms have been enhanced in recent years with two methods: generation-based collection and Baker incremental copying collection. Generation-based collection requires special actions during certain store operations to implement the “write barrier”. Incremental collection requires special actions on certain load operations to implement the “read barrier”. This paper evaluates the performance of different implementations of the read and write barriers and reaches several important conclusions. First, the inlining of barrier checks results in surprisingly low overheads, both for the write barrier (2%-6%) and the read barrier (< 20%). Contrary to previous belief, these results suggest that a Baker-style read barrier can be implemented efficiently without hardware support. Second, the use of operating system traps to implement garbage collection methods results in extremely high overheads because the cost of trap handling is so high. Since this large overhead is completely unnecessary, operating system memory protection traps should be reimplemented to be as fast as possible. Finally, the performance of these approaches on several machine architectures is compared to show that the results are generally applicable.


Abstract
Cache performance is an important part of total performance in modern computer systems. This paper describes the use of trace-driven simulation to estimate the effect of garbage collection algorithms on cache performance. Traces from four large Common Lisp programs have been collected and analyzed with an all-associativity cache simulator. While previous work has focused on the effect of garbage collection on page reference locality, this evaluation unambiguously shows that garbage collection algorithms can have a profound effect on cache performance as well. On processors with a direct-mapped cache, a generation stop-and-copy algorithm exhibits a miss rate up to four times higher than a comparable generation mark-and-sweep algorithm. Furthermore, two-
way set-associative caches are shown to reduce the miss rate in stop-and-copy algorithms often by a factor of two and sometimes by a factor of almost five over direct-mapped caches. As processor speeds increase, cache performance will play an increasing role in total performance. These results suggest that garbage collection algorithms will play an important part in improving that performance.


Abstract
Dynamic memory management is an important part of a large class of computer programs and high-performance algorithms for dynamic memory management have been, and will continue to be, of considerable interest. This paper presents empirical data from a collection of six allocation-intensive C programs. Extensive statistics about the allocation behavior of the programs measured, including the distributions of object sizes, lifetimes, and interarrival times, are presented. This data is valuable for the following reasons: first, the data from these programs can be used to design high-performance algorithms for dynamic memory management. Second, these programs can be used as a benchmark test suite for evaluating and comparing the performance of different dynamic memory management algorithms. Finally, the data presented gives readers greater insight into the storage allocation patterns of a broad range of programs. The data presented in this paper is an abbreviated version of more extensive statistics that are publicly available on the internet.


Abstract
Because dynamic memory management is an important part of a large class of computer programs, high-performance algorithms for dynamic memory management have been, and will continue to be, of considerable interest. Experience indicates that for many programs, dynamic storage allocation is so important that programmers feel compelled to write and use their own domain-specific allocators to avoid the overhead of system libraries. Conservative garbage collection has been suggested as an important algorithm for dynamic storage management in C programs. In this paper, I evaluate the costs of different dynamic storage management algorithms, including domain-specific allocators; widely-used general-purpose allocators; and a publicly available conservative garbage collection algorithm. Surprisingly, I find that programmer enhancements often have little effect on program performance. I also find that the true cost of conservative garbage collection is not the CPU overhead, but the memory system overhead of the algorithm. I conclude that conservative garbage collection is a promising alternative to explicit storage management and that the performance of conservative collection is likely to be improved in the future. C programmers should now seriously consider using conservative garbage collection instead of malloc/free in programs they write.


Abstract
Because dynamic memory management is an important part of a large class of computer programs, high-performance algorithms for dynamic memory management have been, and will continue to be, of considerable interest. We evaluate and compare models of the memory allocation behavior in actual programs and investigate how these models can be used to explore the performance of memory management algorithms. These models, if accurate enough, provide an attractive alternative to algorithm evaluation based on trace-driven simulation using actual traces. We explore a range of models of increasing complexity including models that have been used by other researchers. Based on our analysis, we draw three important conclusions. First, a very simple model, which generates a uniform distribution around the mean of observed values, is often quite accurate. Second, two new models we propose show greater accuracy than those previously described in the literature. Finally, none of the models investigated appear adequate for generating an operating system workload.
6.2 Memory Management Glossary

6.2.1 Memory Management Glossary: A

| A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z |

absolute address

See physical address.

activation frame

See activation record.

activation record

Also known as function record, activation frame.

An activation or function record is a data structure, associated with the invocation of a function, procedure, or control block that stores the variables, temporaries, and fixed-sized data that are local to the block, and the information required to return to the invoking context. It is often stored on a control stack.

In a register-based hardware architecture, the current activation record is typically partially stored in registers.

Closures and continuations are specializations of activation records in support of particular language features of LISP, Scheme and related languages.

Relevance to memory management

The current activation record is part of the state of the mutator, and is therefore a root to the collector (2). In languages that permit recursion, activation records have dynamic extent. In languages that permit closures or continuations, activation records may have indefinite extent. Although they may not be visible to the programmer, their memory (1) must be managed by the language run-time support. Because they are usually not visible to the programmer, they may be a source of inexplicable memory overhead.

See also:

stack frame.

activation stack

See control stack.
address  An address is a specification of a memory location in an address space.

An address is almost always represented as an unsigned integer stored in a single machine word. The address is decoded by the hardware in order to access a location on a physical memory (2) device (such as a RAM) or some memory-mapped resource.

Fig. 6.1: A simplified view of addresses, address space, and locations on a 32-bit architecture.

In the MPS
An address is represented by a value of the type mps_addr_t.

address space  An address space is the set of possible addresses. It can also be considered to be a partial function from addresses to locations.

Typically, addresses start at zero and run to \(2^n\), where \(n\) is the address width (for example, 15, 16, 24, 32, 64), which is usually the same as the width of the address bus. This may not be true for segmented architectures.

In modern systems, large parts of the whole address space may be reserved by the operating system or architecture, or not mapped at any given time. The mapped part of the address space may be discontiguous or sparse.

See also:

virtual address space, physical address space.

address space layout randomization

Also known as

ASLR.

The random placement in address space of the stack, data segment, heap, and so on, of a process.

The purpose of ASLR is to make it harder for an attacker to exploit buffer overflow bugs, by making it harder to determine the addresses of data structures.

In the MPS

ASLR also makes it hard to prepare a repeatable test case for a program that performs computation based on the addresses of objects, for example, hashing objects by their address. See Address space layout randomization for techniques to deal with this.

address translation cache

See

translation lookaside buffer.
address-ordered first fit  The allocation policy that always uses the suitable free block with the lowest address. One of the most common allocation policies in use. Commonly implemented by first fit on a single address-ordered free block chain. Sometimes just called “first fit”.

See also:
FIFO-ordered first fit, LIFO-ordered first fit.

Related publication

aging space  In some generational garbage collection systems, when generations are divided into buckets, the aging space is where objects which survive a collection cycle stay until they are old enough to be promoted.

Opposite term
creation space.

algebraic data type  Algebraic data types aggregate or alternate a number of dissimilarly-typed objects. They are termed algebraic because they can be expressed using the sum and product operators, for example (a and b and c) or d.

Examples of algebraic data types include: structures, records, tuples, and unions.

Relevance to memory management
Algebraic data types are usually represented using a heap. Because of their non-uniformity, algebraic data types are more difficult to scan.

See also:
scalar data type, vector data type, heap.

alignment  Alignment is a constraint on the address of an object in memory (2).

The constraint is usually that the object’s address must be a multiple of a power of two, \(2^n\), and therefore that the least significant \(n\) bits of the address must be zero.

The bus hardware of many modern processors cannot access multi-byte (2) objects at any memory address. Often word-sized objects must be aligned to word boundaries, double-words to double-word boundaries, double-floats to 8-byte boundaries, and so on. If a program attempts to access an object that is incorrectly aligned, a bus error occurs.

Relevance to memory management
A memory manager must take care to allocate memory with an appropriate alignment for the object that is going to be stored there. Implementations of malloc have to allocate all blocks at the largest alignment that the processor architecture requires. Other reasons for aligning objects include using the least significant bits of the address for a tag.

Opposite term
unaligned.

See also:
natural alignment.
In the MPS

An alignment is represented by the unsigned integral type `mps_align_t`. It must be a power of 2. The alignment of objects allocated in a pool may be specified by passing the MPS_KEY_ALIGN keyword argument when calling `mps_pool_create_k()`.

alive

See

`live`.

allocate

Also known as

`cons`.

Allocation is the process of assigning resources. When requested to by the program, an application memory manager or allocator allocates a block of memory for the program to store its data in. Allocation is also known as consing, from `cons`.

When faced with a request for memory from the program, a memory manager must choose a suitable block and hand it over, or fail. The choices made by the memory manager at this point can have a significant effect on the future efficiency of the program.

Allocation is rarely a simple issue. For example, programs usually allocate activation records (automatic variables, and so on) for functions from a processor stack simply by subtracting a number from their stack pointer. However, in a virtual memory system, this may extend the stack onto a previously unused page, in which case the operating system memory manager must carry out some quite complex operations in order to supply the program with backing store for the stack so that the program can continue.

Historical note

The term reserved was often used to mean “allocated”.

Similar term

`malloc`.

Opposite term

`free (1)`.

See also:

`constructor (1)`.

Related publication


In the MPS

See Allocation.
allocation frame

**In the MPS**

An allocation frame is a marker that can pushed onto an allocation point by calling `mps_ap_frame_push()`, and then popped by calling `mps_ap_frame_pop()` to indicate that all blocks allocated on the allocation point are **dead** (in the case of manual pools), or very likely dead (in the case of automatic pools). Allocation frames can be used by the client program to efficiently implement stack-like patterns of allocation.

**allocation mechanism** The algorithm by which an allocator chooses a free block from which to satisfy an allocation request. An allocation mechanism is the implementation of an allocation policy.

A common mechanism is "first fit on an address-ordered free block chain, with eager coalescing". This implements the address-ordered first fit policy, and commonly inherits that name, although there are many other mechanisms for implementing that policy, for example, “leftmost fit on a balanced tree of free blocks ordered by address”.

**Related publication**


**allocation point**

**In the MPS**

An allocation point is an interface to a pool which provides fast buffered allocation, and defers the need for synchronization in a multi-threaded environment. Allocation points belong to the type `mps_ap_t`.

**allocation point protocol**

**In the MPS**

The protocol that ensures safe inline allocation on an allocation point. See Allocation point protocol.

**allocation policy**

**Also known as**

*placement policy.*

The concrete policy used by an allocator for choosing a free block to satisfy an allocation request. For instance, “always allocate from the largest free block” (worst fit) or “use the most recently freed block suitable” (LIFO-ordered first fit).

Each allocation policy is motivated by an allocation strategy and implemented by an allocation mechanism.

**See also:**

*address-ordered first fit, best fit.*

**Related publication**
allocation strategy The high-level design motivation or strategy, of an allocator, which uses observations or theories about patterns of allocation requests to justify an allocation policy.

For instance, “do not allow small long-lived objects to fragment large free areas”, “allocate consecutive objects close together”, and so on. The allocation strategy motivates an allocation policy, which is implemented by an allocation mechanism.

Related publication

allocator The term allocator is often used to refer to the memory manager, usually when it is a simple manual one.

Similar term
memory manager.

See also:
allocation.

ambiguous reference

Also known as
unsure reference.

An ambiguous or unsure reference is a value that is potentially a reference, but the collector cannot prove that it is.

The presence of ambiguous references in a garbage-collected system requires the use of conservative garbage collection.

Opposite term
exact reference.

See also:
floating garbage.

ambiguous root An ambiguous root is a root containing ambiguous references.

Opposite term
exact root.

In the MPS
An ambiguous root has rank mps_rank_ambig().

arena The area of memory used by malloc for allocation.

So named from a semi-mythical “malloc: corrupted arena” message supposedly emitted when some early versions became terminally confused.

See also:
In the MPS
An arena is the data structure responsible for requesting memory from the operating system, making it available to pools, and for garbage collection. Arenas belong to the type mps_arena_t. See Arenas.

arena class

In the MPS
A value of type mps_arena_class_t describing a class of arenas. Arena classes include client arenas and virtual memory arenas.

ASLR

See address space layout randomization.

assertion
A declaration in a program of a condition that is expected always to be true, or which must be true in order for the program to continue to execute correctly.

In the MPS
Memory management mistakes often lead to overwriting errors that corrupt the data structures used by the memory manager to maintain memory. Except in the rash variety, most MPS functions assert the validity of the data structures they operate on. This means that memory management mistakes are detected as early as possible, when there may still be enough evidence in the heap to debug them. See Error handing.

asynchronous garbage collector
A collector is asynchronous with respect to the mutator if it cannot be (easily) predicted when the collector will run.

This means that the mutator must ensure that formatted objects are always scannable.

Opposite term
synchronous garbage collector.

ATC

See translation lookaside buffer.

atomic object

See leaf object.

automatic memory management
Automatic memory management is a general term for techniques that automatically recycle unused memory.

It is not possible, in general, to automatically determine which objects are still live. Even if it didn’t depend on future input, there can be no general algorithm to prove that an object is live (cf. the Halting Problem). However, effective approximations are possible.
In tracing garbage collection, the approximation is that an object can’t be live unless it is reachable. In reference counting, the approximation is that an object can’t be live unless it is referenced. Analysis of the program text can reveal where objects die; A notable technique in this vein is region inference.

Hybrid algorithms are also possible.

**Similar term**
garbage collection.

**Opposite term**
manual memory management.

**In the MPS**
The MPS provides automatic memory management through pool classes such as AMC (Automatic Mostly-Copying), AMS (Automatic Mark and Sweep), and AWL (Automatic Weak Linked).

**automatic storage duration**  In C, objects that are declared with automatic storage duration are live for the duration of a block of code.

In most implementations of C, objects with automatic storage duration are allocated on the stack when a function is entered, and deallocated when it returns.

**Similar terms**
stack allocation, dynamic extent.

**Opposite term**
static storage duration.

**6.2.2 Memory Management Glossary: B**

**backing store**  Backing store (2) is typically part of a hard disk that is used by a paging or swapping system to store information not currently in main memory. Backing store is slower and cheaper than main memory.

Other storage may, less commonly, be used in place of a hard disk (for instance, magnetic tape, floppy disk, or historically, magnetic drum).

In general, backing store may mean any locations used to store information when its preferred or natural location is otherwise being used: for example, memory used by a graphical interface to keep a copy of the contents of obscured windows.

**Similar term**
swap space.

**barrier**(1)  A barrier is a block on reading from or writing to certain memory (2) locations by certain threads or processes.

Barriers can be implemented in either software or hardware. Software barriers involve additional instructions around load or store (1) operations, which would typically be added by a cooperative compiler. Hardware bar-
Barriers don’t require compiler support, and may be implemented on common operating systems by using memory protection.

**Relevance to memory management**

Barriers are used for incremental or concurrent garbage collection.

**See also:**

*read barrier, write barrier.*

**Related publications**

*Pirinen (1998), Zorn (1990).*

**barrier**

A memory barrier is an instruction on certain processor architectures that will ensure certain guarantees about the order of accesses to memory.

Some processor architectures make very few guarantees about the relative orders of *load* and *store (1)* operations in the instruction stream and the actual order of accesses to main memory. These architectures will often have special instructions that make stronger guarantees.

For example, the ARM has the *DMB* (Data Memory Barrier) instruction:

It ensures that all explicit memory accesses that appear in program order before the DMB instruction are observed before any explicit memory accesses that appear in program order after the DMB instruction.

These instructions are vital for certain synchronization operations.

**barrier hit**

**See**

*protection fault.*

**base pointer**

A *base pointer* is a *pointer* to the base or start of an *object*.

This term is commonly used in opposition to *derived pointer*.

Note that *Boehm & Chase (1992)* define “base pointer” to be “any pointer value directly recognizable by the collector (1)”, and this may well include *interior pointers*.

**Opposite term**

*derived pointer.*

**In the MPS**

For objects with *in-band headers*, the MPS distinguishes between the base pointer, which points to the start of the header, and the *client pointer*, which points to the first word after the end of the header.

**best fit**

The allocation policy that always allocates from the smallest suitable *free block*. Suitable allocation mechanisms include *sequential fit* searching for a *perfect fit*, *first fit* on a size-ordered *free block chain*, segregated fits, and *indexed fits*. Many good fit allocators are also described as *best fit*.

In theory, best fit may exhibit bad *fragmentation*, but in practice this is not commonly observed.

**See also:**

*allocation policy, first fit, sequential fit.*
Related publication


BIBOP

Also known as

*big bag of pages.*

BIBOP, or *B1g Bag Of Pages*, is a technique that encodes object type in the high-order bits of their address, by using a lookup table that maps from those bits to a type.

Despite the name, the blocks involved need not be the size of a *page*.

BIBOP requires storing only objects of the same type in a block, but this has the same advantages as segregated fits in general.

Historical note

This technique was invented for the PDP-10 MACLISP by JonL White and Stavros Macrakis. It was an advance on earlier techniques that divided the address space into contiguous blocks for each type.

Related publications


big bag of pages

See

*BIBOP.*

*binary buddies* The most common buddy system allocation mechanism, in which all block sizes are a power of two. Finding a block’s buddy is then a matter of flipping the appropriate bit in the block’s address.

*Internal fragmentation* is usually high, because objects are often not a good fit for power-of-two sized blocks.

See also:

buddy system, allocation mechanism.

Related publication


bit array

See

bitmap.

bit table

See

bitmap.
bit vector

See

bitmap.

bitmap

Also known as

bit array, bit table, bit vector, bitset.

A table of bits.

Relevance to memory management

Bitmaps are sometimes used to represent the marks in a mark-sweep collector (see bitmap marking), or the used memory in a bitmapped fits allocator.

bitmap marking In mark-sweep collectors, bitmap marking is a technique for marking objects that stores the mark bits for the objects in a contiguous range of memory in a separate bitmap. This improves the collector’s locality of reference and cache performance, because it avoids setting the dirty bit on the pages containing the marked objects.

Related publication


bitmapped fit A class of allocation mechanisms that use a bitmap to represent the usage of the heap. Each bit in the map corresponds to a part of the heap, typically a word, and is set if that part is in use. Allocation is done by searching the bitmap for a run of clear bits.

Bitmapped fit mechanisms have good locality of reference, as they avoid examining in-band headers when allocating.

See also:

allocation mechanism, sequential fit, indexed fit.

Related publication


bitmask A bitmap used to select or exclude a set of bits in another bitmap.

bitset

See

bitmap.

black In a tri-color marking scheme, black objects are objects that have been scanned.

More precisely, black objects have been noted reachable and the collector (2) has finished with them and need not visit them again (for the purposes of tracing).

Opposite terms

white, gray.
blacklisting, black-listing  A conservative garbage collector can be made more effective by blacklisting values which resemble addresses that may be allocated in the future, but are known not to be pointers. This list is then used to avoid allocation at those addresses.

For example, such values can be gathered by scanning the roots before any objects have been allocated.

Related publication

block  Block is a vague term for an (often contiguous) area of memory. Often used to describe memory allocated by an allocator such as malloc.

In the MPS
The term block is used as a general term for a unit of allocation, with object being reserved for formatted objects.

bounds error

See
overwriting error.

boxed  Boxed objects are represented by a pointer to a block of memory that contains the object data. Sometimes the pointer is tagged to distinguish it from an unboxed object, or to represent its type. Only the pointer is duplicated when the object is passed around, so updates to the object are reflected everywhere.

Opposite term
unboxed.

See also:
tag, BIBOP.

Related publication

break-table  A break-table is a data structure used by a mark-compact collector to store the relocation information.

See also:
mark-compact.

brk  brk is a Unix system call that sets the limit of the data segment. This limit is known as the break.

The C library implementation of malloc usually allocates memory for the heap by extending the data segment using brk or sbrk.

Most implementations of malloc never shrink the data segment, so the memory usage of a process never decreases. In most Unix systems, the data segment resides immediately above the program code (text segment) in the address space.

Fig. 6.2: A simplified view of the address space of a Unix process.
broken heart  

*Copying garbage collectors move reachable objects* into another *semi-space*. They leave a *forwarding pointer* in the old *location*, pointing to the new. The object at the old location is known as a broken heart.

**Similar term**

*forwarding pointer.*

bucket  

In a *generational garbage collector*, it is often desirable to divide *generations* by the age of the *object*. These divisions are known as buckets.

**See also:**

*generational garbage collection, aging space, creation space.*

buddy system  

Buddy systems are a subclass of *strict segregated fit allocation mechanisms* which make *splitting* and *coalescing* fast by pairing each block with a unique adjacent *buddy* block.

There is an array of *free lists*, one for each allowable block size. Allocation rounds up the requested size to an allowable size and allocates from the corresponding free list. If the free list is empty, a larger block is selected and split. A block may only be split into a pair of buddies.

A block may only be coalesced with its buddy, and this is only possible if the buddy has not been split into smaller blocks.

The advantage of buddy systems is that the buddy of a block being freed can be quickly found by a simple address computation. The disadvantage of buddy systems is that the restricted set of block sizes leads to high *internal fragmentation*, as does the limited ability to coalesce.

Different sorts of buddy system are distinguished by the available block sizes and the method of splitting. They include *binary buddies* (the most common), *Fibonacci buddies, weighted buddies*, and *double buddies*.

**See also:**

*allocation mechanism, segregated free lists, segregated fit, strict segregated fit.*

**Related publication**

*Wilson et al. (1995).*

buffer  

A *buffer* is a large block of *memory (2)* from which blocks are *allocated* contiguously, as a simple technique for fast *allocation*.

By keeping only a *high-water* mark (that is, a *pointer* to the start of unused memory), the buffer technique avoids expensive *in-band headers* and the searching of *free block chains*. Buffers tend to, however, lead to *external fragmentation*.

**Related publication**

*Appel et al. (1988).*

**In the MPS**

Buffers are implemented using *allocation points* attached to *pools*.

bus error  

Strictly speaking, a *bus error* is a fault on a hardware bus, such as when an invalid *address* is issued.

Generally, any hardware exception caused by a *memory (2)* access (for example, *loading* an *unaligned word*) is termed a *bus error*. The term is often used more loosely as a synonym for any memory access error.

**See also:**
segmentation violation.

byte\(^{1}\) A unit of storage measurement, equal to 8 bits.

It does not matter how the bits are arranged: a byte is just a quantity.

This is the sense of byte used in the terms kilobyte, megabyte, gigabyte, terabyte, etc. The prefixes in these terms derive from the SI prefixes for powers of 1000, but since powers of two are much more common in binary computers, they are used to denote powers of 1024 (\(2^{10}\)).

See also:

word.

byte\(^{2}\) A data type defined by a processor architecture.

For example, the smallest addressable memory location on the Intel x86 family is the 8-bit byte.

Historical note

The PDP-10 had 36-bit words, and defined “byte” to be a general sub-word bit-field: compare byte (3). On this machine it was commonplace for characters to be packed four or five to a word using 9- or 7-bit bytes respectively.

See also:

word.

byte\(^{3}\) A contiguous set of bits used to represent a range of values compactly.

The number of bits in a byte is a measure of the information content of the byte. An \(n\)-bit byte can represent \(2^n\) distinct values.

Bytes may be packed into (or otherwise stored in bit-fields of) integers, words, or other aligned values for space efficiency.

byte\(^{4}\) A data type or storage unit defined by a programming language.

In ANSI/ISO C, “the unit of data storage large enough to hold the basic character set of the execution environment”. In this sense, it is often used synonymously with the C type char. C defines sizeof(char) to be 1. Many architectures that run C programs equate this sense of byte and byte (2).

6.2.3 Memory Management Glossary: C

A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z

C89

See

C90.

C90

Also known as

C89.

A revision of the ANSI/ISO Standard for the C programming language. Although more than twenty years old, it remains the only form of Standard C that is supported by all the major compilers, including Microsoft Visual C.
In the MPS
The public interface conforms to this standard. See Interface conventions.

Related publication

C99
A revision of the ANSI/ISO Standard for C the C programming language.

In the MPS
Keyword arguments can be conveniently passed to functions using C99’s compound literal syntax. See Keyword arguments.

Related publication

cache(1)

Also known as
memory cache, cache memory.

A processor’s memory cache is a small piece of fast, but more expensive memory, usually static memory (1), used for copies of parts of main memory. The cache is automatically used by the processor for fast access to any data currently resident there. Access to the cache typically takes only a few processor clock cycles, whereas access to main memory may take tens or even hundreds of cycles.

What part of main memory is resident in a cache, and the mechanisms by which it is kept consistent, are quite varied. See cache policy.

Some systems have more than one level of cache. “Level 1 cache” is the fastest, smallest storage level, “level 2” the next fastest, and so on.

See also:
storage hierarchy, cache (2).

cache(2) A cache is any small, fast piece of memory (1), used for copies of data that normally reside in a larger, slower piece of storage. The cache is used to speed up access to data resident in the slower storage.

In a typical cache, recently used data is resident in the cache (although the details of this depend on the cache policy). A cache (1) is the most common example of a cache(2).

See also:
storage hierarchy.

cache memory

See
cache (1).

cache policy Any cache (3) uses a cache policy to decide which data to store. A cache policy is an attempt to predict the future, so that the cache will provide swift responses to future requests.
Cache policy may be implemented in hardware, software, or a combination of both. Some systems allow programs to influence cache policy, by giving hints or directions about future use of data.

There are three main aspects of cache behavior which the cache policy can affect:

1. Fetch policy. This determines which data is fetched into the cache, usually as a result of receiving a request for data that isn’t cached.
2. Eviction policy. This determines which data is discarded from the cache to provide space for newly fetched data.
3. Write policy. This determines how and when modifications to cached data are synchronized with the underlying storage.

See also:

*cache (1)*, *cache (2)*, *cache (3)*.

Related publications


---

**caching**(3)

Also known as

*memoization*, *tabling*.

*Caching* is a heuristic that stores answers to questions asked in the past in a cache or a table, in order that they may be more quickly answered in the future. This process is also called memoization and tabling (by the Prolog community).

A “look-ahead cache” attempts to store answers to questions that will be asked soon. A *cache (2)* is a common example of a cache(3).

---

**cactus stack**

Also known as

*spaghetti stack*.

A cactus stack is a stack with branches. When diagrammed, its shape resembles that of a saguaro cactus.

In languages that support continuations, activation records can have indefinite extent. One technique for implementing continuations is not to copy the activation records that are captured, but rather to create a fork in the stack below the captured stack frames, so that new frames appear as a parallel branch. Often the process of forking is done lazily: captured frames are only duplicated if they are modified.

**card**

A card is a division of memory, all cards being of equal size (in a particular area of discourse). A card is usually bigger than a word and smaller than a page. Cards are used in a technique called card marking whereby dirty bits (which record which portions of old generations have been written into) are maintained for each card. Often the use of cards will also entail the use of a crossing map.

**card marking**

A technique for managing pointer stores (1) into old generations (which in turn is used to track inter-generational pointers). Each generation is divided into a number of equal-sized cards, and when a generation is written into, the particular card written to is recorded (often by using a bitmap). Subsequently, when scanning an older generation in order to collect a younger generation, only the recorded cards (in the old generation) need to be scanned.

See also:
generational garbage collection.

Related publications


cell

See

object.

Cheney collector

Also known as

Cheney scan.

A Cheney collector uses the tospace of a two-space collector as a queue of objects remaining to be scanned, thus eliminating the need for recursion when tracing the graph of objects.

Related publication


Cheney scan

See

Cheney collector.

clamped state

In the MPS

One of the three states an arena can be in (the others being the unclamped state and the parked state). In the clamped state, no object motion occurs and the staleness of location dependencies does not change. However, a garbage collection may be in progress. Call mps_arena_clamp() to put an arena into the clamped state.

client arena

In the MPS

An arena class that gets its memory (2) from the client program. See Client arenas.

client object

In the MPS

A formatted object that contains data from the client program. One of three types of formatted objects, the other two being forwarding objects and padding objects.

client pointer

In the MPS

A pointer to the first word in an object that’s not part of the in-band header. See In-band headers.
closure A closure is a function or procedure that is saved along with the current bindings from enclosing blocks for later invocation.

Some programming languages, such as ALGOL, permit nested blocks to access the local variables of enclosing blocks. Lisp-like languages further permit such an inner block (in particular a function or procedure) to be saved for later invocation. The act of saving such an inner block along with the current bindings of variables in the enclosing blocks that are referenced by the inner block, is called closing over or capturing those variables. The object created is termed a closure. A closure is invoked just like the function from which it was built, passing whatever parameters the function accepts, but when the function executes, the variables that belong to enclosing blocks will have the bindings that were in effect when the closure was created.

Relevance to memory management

A closure is typically implemented by saving both the function and any activation records that contain variables referenced by the function. The closure creates additional implicit references to the bindings closed over and hence must be accounted for in any memory management scheme. The closure itself is an object that must be managed and may have either dynamic extent or indefinite extent depending on whether it is only used by inner blocks of the creating block or passed out of the creating block.

See also:

continuation.

coalescope Coalescing is the act of merging two adjacent free blocks.

Coalescing reduces external fragmentation, but is not totally effective.

Coalescing can be done as soon as blocks are freed, or it can be deferred until some time later (known as deferred coalescing), or it might not be done at all.

Wilson et al. (1995) has details about fragmentation, and which coalescing strategies are effective under what circumstances.

collect An object is collected when it is reclaimed by a garbage collector.

Similar term

reclaim.

collection

See

collection cycle.

collection cycle

Also known as

collection.
A collection cycle is a single complete execution of a tracing garbage collection algorithm.

Each collection cycle includes (not necessarily in strict order) choosing a condemned set; scanning roots and objects that have not been condemned; tracing the object graph to find all condemned objects that are reachable; and reclaiming those that were not reachable.

In non-incremental garbage collection, the mutator pauses at the start of a collection cycle and cannot continue until it is complete. In incremental and parallel garbage collection, a collection cycle can be interleaved with, or simultaneous to, mutator activity.

collector\(^{(1)}\)

See
garbage collector.

collector\(^{(2)}\) In a garbage-collected system, the part that executes the garbage collection code, which discovers unused memory\(^{(1)}\) and reclaims it.

For purposes of describing incremental garbage collection, the system is divided into the mutator and the collector. These can be separate threads of computation, or interleaved within the same thread.

Historical note
This term is due to Dijkstra et al. (1976).

Opposite term
mutator.

color, colour In a tri-color marking scheme, each node has a one of three colors: black, white, or gray. In a treadmill, nodes may also be colored off-white.

commit limit

In the MPS
The commit limit is a limit on the committed memory\(^{(2)}\) that the arena will obtain from the operating system. It can be changed by calling mps_arena_commit_limit_set().

committed\(^{(1)}\)

See
mapped.

committed\(^{(2)}\)

In the MPS
A block has been committed if it is fully initialized and is under the management of the MPS, as opposed to a block that is merely reserved. See Allocation point protocol.

compactifying

See
compaction.
compaction

Also known as
compactifying.

Compaction is the process of moving live objects to eliminate dead space between them. Some people call this compactifying, to distinguish it from techniques for compressing data structures.

Compaction is used to avoid external fragmentation and to increase locality of reference.

composite object  In the PostScript language, composite objects are the boxed objects.

Unlike a simple object, the main data (what PostScript calls the value) in a composite object are stored separately, in VM (2). Several composite objects can share the same value.

Similar term
boxed.

Opposite term
simple object.

comprehensive  A collector (1) is comprehensive if all garbage (or, all unreachable objects) is reclaimed in one collection cycle.

See also:
garbage collection.

concurrent garbage collection

See
parallel garbage collection.

condemned set

Also known as
threatened set.

Condemned objects are those which are candidates for recycling within a collection cycle.

At the start of a collection cycle, the collector (1) may choose to condemn some objects (the condemned set or threatened set) but not to condemn others (the immune set). Objects that are not condemned are assumed to be live and behave as roots for the purposes of that collection cycle.

Many simple tracing garbage collection algorithms begin by condemning all objects, but generational garbage collectors will condemn individual generations or combinations of generations. Often young generations are condemned but older ones are not, because objects in older generations are less likely to have become unreachable.

In collectors using tri-color marking, at the start of a collection cycle the condemned set is exactly the set of objects that the collector colors white.

Opposite term
immune set.
connected  

*Objects* are connected if and only if one contains a *reference* to the other.

See also:

*graph*.

cons$^{(1)}$  In *Lisp*, *cons* is a primitive operation creating a list element (from English “CONStruct”). By extension, a *cons* is the element created.

**Related link**

Function CONS in the Common Lisp HyperSpec.

cons$^{(2)}$

See

*allocate*.

**conservative garbage collection**  In conservative *garbage collection*, the layout of *objects* and *roots* is not known, instead the *collector* (1) assumes that any field that looks like a *pointer* might be a *reference*.

Conservative collectors can work with programs where information about the *memory* (2) layout is not available, because, for example, the language doesn’t support *garbage collection*.

A conservative collector doesn’t need to know the *format* of the objects, it just needs some idea of where the object boundaries are. It regards any field value that looks like a pointer to an object (or, sometimes, into the middle of one), as preventing the *recycling* of that object. It can’t move objects, because then the references to the moved objects would need to be updated, and such *ambiguous references* must not be modified, in case they weren’t pointers after all. Therefore, conservative collectors are usually *mark-sweep collectors*.

Because references are ambiguous, some objects may be retained despite being actually *unreachable*. In practice, this happens rarely, and refinements such as *black-listing* can further reduce the odds.

**Opposite term**

*exact garbage collection*.

See also:

*ambiguous root*, *semi-conservative garbage collection*, *interior pointer*.

**Related publications**

*Boehm & Weiser (1988), Boehm (1993).*

constant root

**In the MPS**

A *root* that the *client program* promises not change after it is registered, by specifying the *root mode* *MPS_RM_CONST* when calling a registration function such as *mps_root_create()*.

**constructor$^{(1)}$**  A constructor is a function or method that *allocates* and initializes an *object*.

**Opposite term**

*destructor* (1).
**constructor** In C++, a *constructor* is a member function that is used to initialize a newly-allocated object.

The actual allocation of *memory* is performed by `operator new` or the compiler (for *static* and *stack allocation*), and the new *block* is then passed to the appropriate constructor.

See also: `destructor (2)`.

**continuation** A continuation is the data required to restore an execution context after invocation of another context, typically as a subroutine.

**Relevance to memory management**

If continuations can be represented as first-class objects, as in *Scheme*, the execution contexts can no longer be stored on a *stack*, instead, (at least some) *activation records* have to be *heap-allocated*.

See also: `closure`.

**control stack**

Also known as

`activation stack`, `execution stack`.

A *stack* that stores *activation records*, particularly subroutine return information, is known as a *control stack*.

Typically the control stack is supported and used by the hardware architecture and the operating system, limiting the types and sizes of *objects* that can be stored on it. Often, only one type of object, a *stack frame*, is permitted, and the layout of that is defined by the hardware architecture.

**Relevance to memory management**

Theoretically, a control stack is simply an array of activation records, and hence just another object managed by the *memory manager*. In practice, the control stack is central to the performance of the hardware architecture and may require special treatment. In particular, it may not be accessible as ordinary *memory* (2), or it may have its own *cache* (2) with specific updating requirements.

**Similar term**

`stack`.

See also:

`data stack`.

**cool**

*In the MPS*

A *variety* in which most MPS functions *assert* that their data structures are valid, even functions on the *critical path*. See *Building the Memory Pool System*. Compare *hot* and *rash*.

**copying garbage collection**

Also known as

`scavenging garbage collection`.
Copying garbage collection is a kind of tracing garbage collection that operates by relocating reachable objects (this is sometimes called scavenging) and then reclaiming objects that are left behind, which must be unreachable and therefore dead.

A copying garbage collection relies on being able to find and correct all references to copied objects.

Fig. 6.3: Copying garbage collection.

Similar term
moving.

See also:
broken heart, forwarding pointer, two-space collector.

In the MPS
The AMC (Automatic Mostly-Copying) pool class implements copying garbage collection (more precisely, mostly-copying garbage collection).

core  A historical synonym for main memory, deriving from the cores or ferrite rings which were once the main technology used to implement it.

creation space  In generational garbage collection, when generations are divided into buckets, the creation space is where new objects are created in each generation. This term is sometimes used as a synonym for nursery space.

Opposite term
aging space.

See also:
generational garbage collection.

critical path

In the MPS
The sequence of operations on which the MPS spends the majority of its time, consisting of scanning, fixing, marking and copying. See design-critical-path.

crossing map  Where memory (2) has already been divided into some fixed-sized unit (for example, pages or cards), a crossing map records where objects lie across the boundaries of the fixed-sized units. In other words, which fixed-sized units do not start with the beginning of an object.

A system which implements remembered sets by page marking or card marking needs to scan all the pointers in the page or card. If the system can not scan partial objects (or requires information in the object header in order to scan a partial object), a crossing map is necessary to find the beginning of the first object in the unit.
Relevance to memory management

In a sense, a crossing map is an optimization of tagged architecture. It represents the minimum information necessary to determine how to interpret any word of memory.

cyclic data structure  A data structure is cyclic if some of its references form a loop; that is, there’s an object that can be reached by following references from itself.

6.2.4 Memory Management Glossary: D

A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z

dangling pointer  A dangling pointer is a surviving reference to an object that no longer exists at that address.

   In manual memory management, dangling pointers typically arise from one of:
   1. A premature free, where an object is freed, but a reference is retained;
   2. Retaining a reference to a stack-allocated object, after the relevant stack frame has been popped.

   Dangling pointers can occur under automatic memory management, because of a garbage collection bug (such as premature collection, or moving without updating all references), but this is much rarer because garbage collection code is usually a single common core of reused code in which these bugs can be fixed systematically.

data stack  A stack used to manage the storage of stack-allocated objects, other than activation records, often under program control.

   Because of the limitations that may be imposed on the control stack, or to support stack-like semantics for certain data structures, some language implementations manage additional data stacks in software for storing objects that have dynamic extent but that do not fit within the constraints of the control stack.

   See also:
   control stack.

dead  An object is dead if it is not live; that is, when the mutator cannot reach any state in which it accesses the object.

   It is not possible, in general, for garbage collectors to determine exactly which objects are dead and which are live. Instead, they use some approximation to detect objects that are provably dead, such as those that are unreachable.

   Opposite term
   live.

   See also:
   garbage, undead, free (3).

deallocate

   See free (1).

debugging pool

   In the MPS
A pool that performs extra checking in order to find errors in the client program. It uses fenceposts to detect overwriting errors and it writes patterns over reclaimed blocks in order to detect use after free or missing references during scanning.

**deferred coalescing**  Deferred coalescing is a policy which coalesces free blocks some time after the blocks are freed, as opposed to coalescing free blocks immediately as they are freed.

Adjacent free blocks can be coalesced to form larger free blocks; deferred coalescing is a catch-all for policies which perform this coalescing sometime after the blocks were freed.

Given this rather flexible definition there are a number of choices for when to coalesce: as the free list is traversed during allocation, when the allocation cannot be satisfied from the free list, periodically, and so on. In addition there are choices to be made regarding how much coalescing to perform at any one time.

**deferred reference counting**  Deferred reference counting reduces the cost of maintaining reference counts by avoiding adjustments when the reference is stored on the stack.

On many systems, the majority of stores are made into local variables, which are kept on the stack. Deferred reference counting leaves those out and counts only references stored in heap objects. This requires compiler support, but can lead to substantial performance improvements.

Objects cannot be reclaimed as soon as their reference count becomes zero, because there might still be references to them from the stack. Such objects are added to a zero count table (ZCT) instead. If a reference to an object with a count of zero is stored into the heap, then the object is removed from the ZCT. Periodically the stack is scanned, and any objects in the ZCT which were not referenced from the stack are reclaimed.

Deferred reference counting has been used successfully with several languages, notably Smalltalk. However, since it fails to collect objects with cyclic references, it is often used alongside a tracing garbage collector.

**Related publication**  Deutsch & Bobrow (1976).

**dependent object**

**In the MPS**

In AWL (Automatic Weak Linked), each object in the pool can have a dependent object. While scanning an object, the MPS ensures that the dependent object is unprotected so that it can be updated. This feature supports the implementation of weak-key and weak-value hash tables. See Dependent objects.

**derived pointer**

See

*interior pointer*.

**derived type**

**In the MPS**

In the MPS interface, a derived type is a type that is neither an opaque type nor a transparent type, but is instead a structure or function type based on transparent and opaque types and on built-in C types. See Interface conventions.

**destructor**  A destructor is a function or a method that performs the explicit deallocation of an object. It may also perform clean-up actions.
Opposite term
constructor (1).

destructor (2) In C++, a destructor is a member function that is used to clean up when an object is being deallocated.
When an object is being destroyed (by delete or automatically), the appropriate destructor is called, and then the actual deallocation of memory (2) is performed by operator delete or the run-time system (for static and stack allocation).

See also:
constructor (2).

DGC

See
distributed garbage collection.

direct method Direct methods of automatic memory management maintain information about the liveness of each object, detecting garbage directly.

Such bits of information, for example, reference counts, are typically stored within the objects themselves.

Direct garbage collection can allow memory (2) to be reclaimed as soon as it becomes unreachable. However, the stored information must be updated as the graph of objects changes; this may be an expensive operation, especially in distributed garbage collection where it can lead to intensive communication between processors, and make garbage collection less robust to network failures.

Opposite term
indirect method.

Related publication
Jones et al. (2012).

dirty bit A dirty bit is a flag indicating that a page (or similar) has been written to since it was last examined.

Dirty bits are used by cache (2) to determine which pages must be written out, and by garbage collectors in conjunction with write barriers.

distributed garbage collection

Also known as
DGC.

Distributed garbage collection is garbage collection in a system where objects might not reside in the same address space or even on the same machine.

Distributed garbage collection is difficult to achieve in widely-distributed systems (over wide-area networks) because of the costs of synchronization and communication between processes. These costs are particularly high for a tracing garbage collector, so other techniques, including weighted reference counting, are commonly used instead.

double buddies A buddy system allocation mechanism using a pair of binary buddy systems with staggered size classes.
One system is a pure binary buddy, with powers-of-two classes (2, 4, 8, \ldots). The other uses some fixed multiple of powers-of-two (for example, 3, 6, 12, \ldots). This resembles weighted buddies, but the two buddy systems are treated independently: blocks cannot be split or coalesced from one to the other.

**Related publication**

*Wise (1978).*

double free  A double free is when an attempt is made to free (1) a memory (2) block that has already been freed.

This usually occurs in manual memory management when two parts of a program believe they are responsible for the management of the same block.

Many manual memory managers have great trouble with double frees, because they cannot cheaply determine that deallocated blocks were already free. Instead, they corrupt their free block chain, which leads to mysterious problems when the same block is subsequently allocated.

See also:

*premature free.*

doubleword  

Also known as

longword.

A doubleword is a unit of memory consisting of two adjacent words.

**Historical note**

On the Intel 80386, 80486, and Pentium processors, the doubleword of 32 bits is actually the natural word size, but the term *word* is still used for the 16-bit unit, as it was on earlier processors of this series.

See also:

*quadword.*

doubly weak hash table  A hash table that is both weak-key and weak-value.

DRAM

See

*dynamic memory.*

dynamic allocation

See

*heap allocation.*

dynamic extent  An object has dynamic extent if its lifetime is bounded by the execution of a function or some other block construct.

Objects of dynamic extent are usually stack-allocated.

Similar term

*automatic storage duration.*
Opposite term

indefinite extent.

dynamic memory

Also known as
dynamic RAM, DRAM.

Dynamic memory, or dynamic RAM (DRAM, pronounced “dee ram”), is a type of RAM.

Dynamic memory requires periodic refreshing to avoid losing its contents (as opposed to static memory, the contents of which are preserved without any need for refreshing). The refreshing is performed by additional “refresh hardware” usually external to the dynamic memory package itself, sometimes by the main CPU. Dynamic memory is cheap and compact and is the choice for large amounts of relatively fast memory, such as the main memory of PCs. Dynamic memory often comes packaged in SIMMs or DIMMs.

See also:
static memory, SDRAM.

dynamic RAM

See
dynamic memory.

6.2.5 Memory Management Glossary: E

A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z

ecreu

See
off-white.

digraph

In a graph, an edge is a connection between two nodes.

In a directed graph (digraph), edges have a direction; otherwise the start and end nodes are interchangeable. By convention, two directed edges between the same two nodes, but in different directions, are depicted as a bi-directional edge.

Typically an edge represents some relation between nodes.

Relevance to memory management

In memory management, edges normally represent the fact that an object holds a reference to another object.

See also:
graph.

entry table

An entry table is a table of references into a set of objects used to indirect references from the outside.
The Lieberman-Hewitt collector (1) represented references from older generations to younger ones by indirect pointers through an entry table in the younger generation that contained the actual address of the young object. This is fairly expensive without special hardware; other generational collectors generally use remembered sets.

See also:
* generational garbage collection, exit table.

**Related publication**
*Lieberman & Hewitt (1983).*

**entry table** (2) An entry table is an implementation of a remembered set, where, for a given generation, there is a list of objects in older generations which contain references into that generation. One could also store the actual locations of the references, which would save time when scanning, but incur other costs.

**Similar term**
rememberset.

See also:
* generational garbage collection, exit table.

**exact garbage collection**

Also known as
precise garbage collection, type-accurate garbage collection.

*Garbage collection* is exact (or precise) if it deals only with exact references. An exact collector (1) needs to know the format of the objects and the roots, so that it can tell which fields are references.

**Opposite term**
conservative garbage collection.

**exact reference**

Also known as
precise reference, sure reference.

An exact or precise or sure reference is a value the collector (1) knows is a reference. This is the usual sort of reference. The term is used to draw a contrast with ambiguous reference.

**Opposite term**
ambiguous reference.

**exact root**

Also known as
precise root.
An exact or precise root is a root that contains only exact references.

Opposite term
ambiguous root.

See also:
exact reference.

In the MPS
An exact root has rank mps_rank_exact().

exact segregated fit  A segregated fit allocation mechanism which has a separate free list for each possible block size. The array of free lists may be represented sparsely. Large blocks may be treated separately.

See also:
segregated fit, segregated free list, allocation mechanism.

Related publication

execution stack
See
control stack.

exit table  An exit table is a table of all references from a set of objects to objects outside the set.

See also:
entry table (1), entry table (2).

Related publication

extent
See
lifetime.

external fragmentation  External fragmentation is the inability to use memory because free memory is divided into many small blocks.
If live objects are scattered, the free blocks cannot be coalesced, and hence no large blocks can be allocated.
Common solutions to external fragmentation include:
1. Moving garbage collection;
2. Handles;
3. Making all your objects the same size.
See also:

*internal fragmentation.*

Related publication


6.2.6 Memory Management Glossary: F

A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z

fencepost, fence post  A fencepost is spare *memory (1)* between *allocated blocks* for checking purposes.

Some *memory management* systems leave spare memory between allocated blocks and store special values in it. If a checking routine finds that these memory *locations* have been modified, this probably indicates an *overwriting error* in the application that was allocated the adjacent block.

Such checking can help application programmers to find bugs that would otherwise be difficult to reproduce and track down.

Similar term

in-band header.

In the MPS

*Debugging pools* use fenceposts. See *Debugging pools*.

fencepost error, fence post error  The term *fencepost error* refers to errors arising from the fact that, to enclose *n* consecutive intervals, you need *n* + 1 end-points, from the number of posts required to support fence rails.

An example of a fencepost error would be, in *C*:

```c
void f(void)
{
  int i;
  int a[10];
  for(i = 0; i <= 10; i++)
    a[i] = 0;
}
```

because the declaration *int a[10];* creates an array of ten integers, with indices from 0 to 9, but the *for* loop index *i* runs from 0 to 10.

Fibonacci buddies  A common *buddy system allocation mechanism*, in which block sizes form a Fibonacci series (each block size is the sum of the two previous sizes). Each block can therefore be *split* to form two blocks of valid sizes, and the sizes are more closely spaced than in *binary buddies*. However, if the same size is allocated repeatedly, performance may suffer as the remainder blocks may have to be split again (or become fragments).

See also:

*buddy system, allocation mechanism.*

Related publication

FIFO-ordered first fit  The allocation policy that always uses the least-recently freed suitable free block. Commonly implemented by adding freed blocks to the end of a free block chain, and then using first fit allocation on this chain. free can be very quick, depending on the coalescing policy.

According to Wilson et al. (1995), this policy controls fragmentation quite well, better than LIFO-ordered first fit and as well as address-ordered first fit in some cases, although locality may be worse.

file mapping

See memory mapping.

finalization

Also known as termination.

In garbage-collected languages, it is often necessary to perform actions on some objects after they are no longer in use and before their memory can be recycled. These actions are known as finalization or termination.

A common use of finalization is to release resources when the corresponding “proxy” object dies. For example, an open file might be represented by a stream object. When this object has been proven dead by the collector, it is certain that the file is no longer in use by the program, and it can and should be closed before the stream is recycled.

Note that finalization is not, in general, guaranteed to be prompt, and this can cause problems if it is used to manage scarce operating system resources such as file descriptors.

Many object-oriented languages provide support for finalization, for example, Cedar, Java, Perl 5, and Smalltalk.

The term finalization is sometimes used to refer to the use of destructors, for example in Ada.

In the MPS

See Finalization.

classified block

In the MPS

A block that has been registered for finalization using mps_finalize(), and which the MPS has determined is dead, but whose finalization message has not been discarded. See mps_message_type_finalization().

first fit  First fit is a sequential fit allocation mechanism.

To quote Wilson et al. (1995):

First fit simply searches the free list from the beginning, and uses the first free block large enough to satisfy the request. If the block is larger than necessary, it is split and the remainder is put on the free list.

The first fit mechanism provides a class of first fit allocation policies, depending on the order in which the free list is stored. Address-ordered first fit stores the list in order of (usually increasing) address. LIFO-ordered first fit puts blocks on the front of the free list when they are freed. FIFO-ordered first fit puts blocks on the end of the free list when they are freed.
fix

In the MPS

To fix a reference from one block to another is to declare it to the MPS by calling MPS_FIX1() and MPS_FIX2() within a scan method. In a moving pool, fixing a reference may also update it to point to the new location of the block. See Scanning.

flip

The instant in a two-space collector when the roles of the two semi-spaces are reversed. What was the tospace is now marked as fromspace and condemned. What was the fromspace becomes the site for all new allocations. Also used in a more general sense to mean the initiation of a new collection cycle.

Fig. 6.4: In a two-space collector, the flip occurs just before the fromspace is condemned and copying starts.

floating garbage

Floating garbage is garbage that is not recycled promptly due to some approximation or optimization in the garbage collector.

Floating garbage results from conservatively estimating an object that is really unreachable to be reachable for the purposes of a particular collection cycle. Using estimates can have considerable performance benefits but also result in higher memory consumption.

Typical estimates that cause floating garbage are:

1. Every register or activation frame slot holds a reachable value: this is not always true, as objects stored in dead registers or slots may be otherwise unreachable. This estimate can simplify the compiler as well as the interface between the compiler and the garbage collector.

2. Every object in a remembered set is reachable: this is not always true, because remembered objects can have become unreachable since they were added to the remembered set. This estimate allows remembered sets to be effective; the alternative—determining whether each remembered object is reachable—is equivalent to a full garbage collection.

3. Anything that looks like a reference is one: this is not generally true, because random data can have the same bit pattern as a pointer. Conservative garbage collectors use this estimate.

4. Any object referenced from another is reachable: this is not generally true, because garbage can reference other garbage. Reference counting collectors use this estimate, resulting in their not being able to reclaim self-referential structures.

5. Any object reached during collection remains live until the next collection: this may not be true when the garbage collector runs interleaved with the mutator, as do incremental and concurrent collectors.

A more subtle kind of floating garbage is an unreachable data structure that spans multiple regions that are never condemned together.

foreign code

In the MPS

From the point of view of the client program, foreign code is external code (not part of the client program, or the MPS), which is not aware of and does not co-operate with the MPS. The client program must take care when passing the address of a block in a moving pool to foreign code.

The LO (Leaf Object) pool class is designed for this use case: blocks allocated from this pool do not move and are never protected, and so may be passed safely to foreign code.
format  A format describes the representation of an object; that is, how the object is laid out in memory.

A format usually specifies where the fields of the objects are located and what their type is.

Relevance to memory management

If formats are provided by a language or the application program, exact garbage collection can be used, because the collector (1) can determine which fields are references.

See also:

conservative garbage collection.

format method

In the MPS

One of the methods in an object format, defined by the client program in order to describe its objects to the MPS. May be a scan method, skip method, forward method, is-forwarded method, or padding method.

formatted object

In the MPS

An allocated block that belongs to an object format and may be scanned by the garbage collector. See Object formats.

forward method

In the MPS

A format method that is called by a moving pool when it has moved an object. The forward method replaces the old object with a forwarding marker that points to the new location of the object. See mps_fmt_fwd_t.

forwarding marker, forwarding object, forwarding pointer  Some garbage collectors move reachable objects into another space. They leave a forwarding pointer, a special reference pointing to the new location, in the old location.

Similar term

broken heart.

See also:

copying garbage collection, two-space collector.

In the MPS

The term forwarding object is used. This is a formatted object that has been replaced by a forwarding marker. One of three types of formatted objects, the other two being client objects and padding objects.

fragmentation  Fragmentation is the inability to use memory (1) because of the arrangement of memory already in use. It is usually divided into external fragmentation and internal fragmentation.

Related publication

frame

See

in-band header.

free

Also known as
deallocate.

In manual memory management, to free or deallocate an object is to tell the memory manager that it is no longer needed. The memory (1) may then be recycled by being used for subsequent allocation, or by being returned to the operating system.

Opposite term
allocate.

See also:
free (2), destructor (1).

free

In C, the system function used for explicit deallocation is called free.

free

Memory (2) is free if it is not currently allocated.

Historical note
The term available was commonly used to mean “free”.

Opposite term
allocated.

See also:
free (1).

free

See
unmapped.

free block
A single contiguous area of memory (2) available to satisfy an allocation request.

For the purpose of discussing allocation mechanisms, two adjacent free blocks are not considered to be a single free block, until they are coalesced. Free blocks may be split.

See also:
allocation mechanism, free list.

Related publication
Some systems store the *free list* as a linked list, or chain. Usually the links are stored within the *free (3) blocks*. This means that all *allocated* blocks must be large enough to store these, and implies a minimum size.

Sometimes, the free block chain is ordered by *address*. This makes *coalescence* considerably cheaper, but *deallocation* more expensive.

**See also:**

- *free list*.

### free list

The free list is the set of *free blocks*.

Originally this term meant the single linked list of all free blocks, but as *allocation mechanisms* have become more varied, it has become more generic, and now may be implemented as a tree or other data structure rather than a linked list. If the implementation actually is a linked list of free blocks, this is called a *free block chain* to distinguish it from the abstract term.

There may be several free lists, classed by size or other characteristic. For instance, *segregated free list* systems classify free lists by block size.

**See also:**

- *free block*, *free block chain*.

### freestanding

In the C programming language as defined by *C90*, a freestanding implementation “accepts any strictly conforming program in which the use of the features specified in the library section is confined to the contents of the standard headers `<float.h>`, `<limits.h>`, `<stdarg.h>`, and `<stddef.h>`.” The *C99* standard adds `<iso646.h>`, `<stdbool.h>`, and `<stdint.h>` to this list

In particular, a freestanding implementation need not provide the other features of the standard C library, including I/O, time, and string operations.

*Opposite term*

- *hosted*.

### In the MPS

The MPS is designed to be portable to a freestanding implementation, by restricting the use of other features either to *platform*-specific modules or to the replaceable *plinth* modules.

*Related publications*


### free store, freestore

**See**

- *heap*.

### from space, fromspace

*Also known as*

- *old space, oldspace*. 

264 Chapter 6. Appendices
In copying garbage collection, the space containing a mixture of live and dead objects, out of which the former are copied.

Opposite term
tospace.

function pointer  In the C programming language, a pointer to a function, as distinct from a object pointer. The C programming language does not guarantee that function and object pointers are the same size, or that a pointer of one type can be cast to a pointer of the the other type without losing information (but on every mainstream C implementation, including all those supported by the MPS, they are in fact the same).

Opposite term
object pointer.

function record

See
activation record.

6.2.7 Memory Management Glossary: G

A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z

garbage  Garbage consists of objects that are dead.

In tracing garbage collection, the term is sometimes used to mean objects that are known to be dead; that is, objects that are unreachable.

garbage collection

Also known as
GC.

Garbage collection (GC), also known as automatic memory management, is the automatic recycling of dynamically allocated memory. Garbage collection is performed by a garbage collector which recycles memory that it can prove will never be used again. Systems and languages which use garbage collection can be described as garbage-collected.

Garbage collection is a tried and tested memory management technique that has been in use since its invention in the 1950s. It avoids the need for the programmer to deallocate memory blocks explicitly, thus avoiding a number of problems: memory leaks, double frees, and premature frees. The burden on the programmer is reduced by not having to investigate such problems, thereby increasing productivity.

Garbage collection can also dramatically simplify programs, chiefly by allowing modules to present cleaner interfaces to each other: the management of object storage between modules is unnecessary.

It is not possible, in general, for a garbage collector to determine exactly which objects are still live. Even if it didn’t depend on future input, there can be no general algorithm to prove that an object is live (cf. the Halting Problem). All garbage collectors use some efficient approximation to liveness. In tracing garbage collection, the approximation is that an object can’t be live unless it is reachable. In reference counting, the approximation is that an object can’t be live unless it is referenced. Hybrid algorithms are also possible. Often the term garbage collection is used narrowly to mean only tracing garbage collection.

There is a large body of published work on particular and general garbage collection algorithms.
Historical note
Garbage collection was first invented by John McCarthy in 1958 as part of the implementation of *Lisp.*

Other significant languages offering garbage collection include *Java, ML, Modula-3, Perl, Prolog,* and *Smalltalk.* Major applications using garbage collection include Emacs and AutoCAD; usually, you can’t tell whether an application does or not, but these have extension languages that expose the fact.

Similar term
*automatic memory management.*

Opposite term
*manual memory management.*

See also:
*conservative garbage collection, copying garbage collection, distributed garbage collection, generational garbage collection, incremental garbage collection, parallel garbage collection.*

Related publication
*McCarthy (1960).*

garbage collector

Also known as
*collector.*

A (garbage) collector is (an implementation of) a *garbage collection* algorithm.

This term is often used when referring to particular implementations or algorithms, for example, “the Boehm–Demers–Weiser *collector*”.

GB

See
*gigabyte.*

GC

See
*garbage collection.*

General Protection Fault

Also known as
*GPF.*

A General Protection Fault on the Windows platforms is the equivalent of a *segmentation violation* on Unix.
generation  A generation is a set of objects of similar age.

A generational garbage collector will typically divide the set of all objects into generations, and condemn all the objects in a generation together. Rather than allowing whole generations to age, the collector can promote objects into older generations as they survive successive collection cycles.

New objects are usually allocated in the youngest or nursery generation, but if we know that particular objects will be long-lived, we might want to allocate them directly in an older generation. Thus, more loosely, a generation is a set of objects which have similar expected lifetimes.

See also:

* bucket.

In the MPS

The client program specifies the generational structure of a pool (or group of pools) using a generation chain. See Garbage collection.

---

**generation chain**

In the MPS

A data structure that specifies the structure of the generations in a pool (or group of pools). See Garbage collection.

---

**generation scavenging**

See

* generational garbage collection.

---

**generational garbage collection**

Also known as

generation scavenging.

Generational garbage collection is tracing garbage collection that makes use of the generational hypothesis. Objects are gathered together in generations. New objects are allocated in the youngest or nursery generation, and promoted to older generations if they survive. Objects in older generations are condemned less frequently, saving CPU time.

It is typically rare for an object to refer to a younger object. Hence, objects in one generation typically have few references to objects in younger generations. This means that the scanning of old generations in the course of collecting younger generations can be done more efficiently by means of remembered sets.

In some purely functional languages (that is, without update), all references are backwards in time, in which case remembered sets are unnecessary.

See also:

* remembered set.

In the MPS

The AMC (Automatic Mostly-Copying) and AMCZ (Automatic Mostly-Copying Zero-rank) pool classes support generational garbage collection.
generational hypothesis

Also known as

*infant mortality.*

Infant mortality or the generational hypothesis is the observation that, in most cases, young *objects* are much more likely to *die* than old objects.

Strictly, the hypothesis is that the probability of death as a function of age falls faster than exponential decay (inverse hyper-exponential), but this strict condition is not always required for techniques such as *generational garbage collection* to be useful.

gigabyte

Also known as

*GB.*

A gigabyte is 1024 *megabytes*, or 1073741824 *bytes* (1).

See *byte* (1) for general information on this and related quantities.

good fit The class of *allocation policies* which approximate *best fit*. Strict best fit may be costly to implement (depending on the details of the *allocation mechanism*), so some implementors approximate it, choosing a block which is close in size to the allocation request.

See also:

*best fit, allocation policy, next fit, worst fit.*

Related publication


GPF

See

*General Protection Fault.*

grain The grain of a platform is the smallest *alignment* that is sufficient to accommodate all data accesses on that platform. Often this is a *word* or a small multiple of a word. Double precision floating point numbers often have the strictest alignment requirements.

See also:

*alignment, word.*

graph A graph is a set of *nodes* together with a set of *edges* connecting nodes.

If the edges have direction like arrows (for example, *references* in a graph of *objects*), then the graph is said to be a *directed graph*.

Fig. 6.5: Directed graph.

Relevance to memory management

Graphs are used to model *reachability* for *tracing garbage collection*. The *objects* are considered to form a graph, with the nodes of the graph being the objects and the edges of the graph being the references from one
object to another. Usually, there is a single, distinguished root to which the mutator has direct access, and the nodes strongly connected to it are the reachable modes.

**gray, grey**  In a tri-color marking scheme, gray objects are objects that are proved or assumed (see generational and condemn) to be reachable, but have not yet been scanned.

More precisely, gray objects have been noted reachable, but must still be visited by the collector (2) in order to process their children.

**Similar term**

gray list.

**Opposite terms**

black, white.

**gray list, grey list**  The gray list is the set of objects that a tracing garbage collector has noted reachable, but hasn’t scanned yet.

The gray list is so called because it corresponds to the set of gray objects in the tri-color marking model of graph tracing. The gray list changes as the garbage collector progresses.

Each gray object is scanned, and all white objects referred to by it become gray and are added to the list. Scanning a gray object turns it black. When the gray list is empty, the tracing is finished, and white objects may be reclaimed.

The representation of the gray list is a key part of garbage collector design. The size of the list is potentially proportional to the size of the heap, and the operation of finding the next gray object to scan must be cheap.

**See also:**

Cheney scan.

### 6.2.8 Memory Management Glossary: H

**A** | **B** | **C** | **D** | **E** | **F** | **G** | **H** | **I** | **J** | **K** | **L** | **M** | **N** | **O** | **P** | **Q** | **R** | **S** | **T** | **U** | **V** | **W** | **X** | **Y** | **Z**

**handle**  A handle is an object that represents a resource.

Handles are used when the resource cannot be represented directly. For example, a file handle is an object passed between a process and the OS in order to access a file, because the file itself cannot be represented.

**Relevance to memory management**

In memory management, a handle is an object that represents another object. Handles are usually used because the object itself needs to be moved in memory (2), or even swapped out to disk. The program therefore cannot know the address of the object.

For example, Apple’s Classic Mac OS made extensive use of handles in its heap management to avoid problems due to fragmentation. If the Classic Mac OS Memory Manager could not satisfy a request for memory, it tried compacting the heap: moving all the relocatable objects together to squeeze out gaps. It could do this because the program only had handles on the objects, and not their actual addresses.

Fig. 6.6: Handle-based heap.
Similar term

pointer.

header

See

in-band header.

heap

Also known as

free store, freestore.

The heap or free store is the memory area managed by dynamic allocation. This use of heap is unconnected with the data structure used by the heapsort algorithm.

heap allocation

Also known as

dynamic allocation.

Heap allocation or dynamic allocation means run-time allocation and deallocation of memory in arbitrary order.

Dynamic allocation is usually for objects whose size, quantity, or lifetime could not be determined at compile-time. It is necessary to implement modern data structures, such as recursive trees and full closures.

Objects on the heap can be managed manually, as in C, or automatically, as in Lisp and Java.

Opposite terms

stack allocation, static allocation.

See also:

indefinite extent.

hit A hit is a successful lookup in any form of cache, most commonly at some level of a storage hierarchy, such as a cache or virtual memory system.

Opposite term

miss.

hit rate At any level of a storage hierarchy, the hit rate is the proportion of accesses which hit.

Opposite term

miss rate.

hosted In the C programming language, a hosted implementation is one that provides all the features of the standard C library.

Opposite term
freestanding.

Related publications

hot

In the MPS
A variety in which many MPS functions assert that their data structures are valid, but functions on the critical path do not. Select it by defining CONFIG_VAR_HOT. Compare cool and rash.

huge page

Also known as
large page, superpage.

Some processor architectures support multiple page sizes. This allows operating systems to better match the page size to the granularity of memory usage and so reduce the size of the page table.

6.2.9 Memory Management Glossary: I

A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z

Immediate data
Immediate data is the representation of a value object as one or more machine words, as a register, or as a field in an instruction.

Immediate data takes its name from the value of the object being immediately available, rather than requiring a load or indirect through a reference.

Similar term
unboxed.

Opposite terms
boxed, reference, pointer.

Immune set
The set of objects which are not condemned.

Opposite term
condemned set.

Immutable
In some programming languages, objects of some types are immutable, that is, they cannot be modified. For example, in Standard ML, only arrays and refs are mutable; all other objects are immutable.

This property can be very useful for garbage collection. For instance, no immutable object may contain a reference to an object younger than itself, and no immutable object will appear in a remembered set. Garbage collectors for these languages often take advantage of this property.

In lazy languages, the evaluation of an expression may require an object of a different size, and adjustment of references may take place. This means that, although objects might be immutable at the language level, they are not immutable at the implementation level, and may contain references to younger objects.
Opposite term

mutable.

See also:

generational garbage collection.

immutable object

See

generational garbage collection.

in-band header

Also known as

frame, header.

Some memory managers allocate a fixed amount more than is necessary for each block and use it to store information such as the size of the block or a tag. This extra memory is known as an in-band header or a frame. This is a form of internal fragmentation, although sometimes, alignment requirements result in free space for the header.

Storing control information in-band often results in bad locality, particularly for deallocation.

Opposite term

out-of-band header.

See also:

stack frame, activation frame.

In the MPS

In-band headers are supported by some pool classes and the size of the header is specified by passing the MPS_KEY_FMT_HEADER_SIZE keyword argument to mps_fmt_create_k().

A pointer to the first word after the in-band header is called a client pointer.

in parameter

A function parameter that supplies data from the caller to the function. (The usual case in C.)

Opposite term

out parameter.

in/out parameter

A function parameter that is both an in parameter an out parameter.

In the MPS

In/out parameters are given names ending with _io. See Interface conventions.

incremental garbage collection

Some tracing garbage collection algorithms can pause in the middle of a collection cycle while the mutator continues, without ending up with inconsistent data. Such collectors can operate incrementally and are suitable for use in an interactive system.
Primitive garbage collectors (1), once they start a collection cycle, must either finish the task, or abandon all their work so far. This is often an appropriate restriction, but is unacceptable when the system must guarantee response times; for example, in systems with a user interface and in real-time hardware control systems. Such systems might use incremental garbage collection so that the time-critical processing and the garbage collection can proceed effectively in parallel, without wasted effort.

**Similar term**

parallel garbage collection.

**Opposite term**

stop-and-copy collection.

**See also:**

tri-color marking, barrier (1).

**Related publications**


**In the MPS**

The MPS uses incremental collection, except for collections started by calling `mps_arena_collect()`.

**incremental update**

Incremental-update algorithms for tracing, incremental garbage collection note changes made by the mutator to the graph of objects and update the collector (2) state to make it correctly trace the new graph.

In order for the collector to miss a reachable object, the following two conditions need to hold at some point during tracing:

1. The mutator stores a reference to a white object into a black object.
2. All paths from any gray objects to that white object are destroyed.

Incremental-update algorithms ensure the first condition cannot occur, by painting either the black or the white object gray (see Pirinen (1998) for details).

They are so called because they incrementally update the collector’s view of the graph to track changes made by the mutator.

**Historical note**

This distinction between incremental update and snapshot at the beginning was first introduced for write-barrier algorithms, but it applies to any type of tracing algorithm.

**Opposite term**

snapshot at the beginning.

**See also:**

tri-color marking, strong tri-color invariant, barrier (1).

**Related publications**

indefinite extent  An object has indefinite extent if its lifetime is independent of the block or function-call structure of the program.

The lifetime of such an object can sometimes be determined by the programmer, and specified by freeing the object explicitly. This becomes harder to do correctly as the program becomes more complex, especially if objects are passed across module boundaries, or if higher-order functions are used. In some languages it is impossible to determine the extent at compile-time. In these situations, a garbage collector can be used to recycle objects whose lifetime has come to an end.

Opposite term
dynamic extent.

indexed fit  A class of allocation mechanisms that use an indexing data structure, such as a tree or hash table, to identify suitable free blocks, according to the allocation policy. For instance, a tree ordered by block size may be used to implement the best fit policy.

See also:
allocation mechanism, allocation policy, sequential fit, bitmapped fit.

Related publication

indirect method  Indirect methods of automatic memory management are those in which the information necessary to determine whether an object can be reclaimed is not stored in or associated with that object, but is derived from other objects.

Indirect methods detect garbage by tracing reachable objects.

Indirect methods cannot always reclaim memory (2) as soon as it becomes dead, because it may be necessary to inspect many other objects to determine this. However, not having to store and update information on each object may reduce the overhead for the collector (1). In distributed garbage collection, this can reduce the amount of communication between processors.

Similar term
tracing garbage collection.

Opposite term
direct method.

Related publication
Jones et al. (2012).

infant mortality

See
generational hypothesis.

inline allocation1  Allocation of objects by inline code, that is, without calling an allocation function. This is vital for performance in languages that allocate many small objects.

In the MPS
This is achieved by the Allocation point protocol.

**inline allocation** (2) Allocation of child objects inside their parent, as opposed to allocating child objects on the heap and storing pointers to them in the parent.

**inter-generational pointer** An inter-generational pointer is a reference that is stored in an object in one generation and references an object in another generation.

If the referent’s generation is condemned and the referrer’s generation is not, then the reference is important in two ways. First, the reference keeps the referent alive, so the referrer must be scanned during the collection cycle. Second, the reference must always refer to the referent, so if the referent is moved, then the referrer must be updated.

During a collection, the only objects in non-condemned areas that must be scanned are the ones that contain inter-generational pointers. Generational garbage collectors make use of write barriers and data structures like entry tables (2), exit tables, and remembered sets to track those objects at run-time.

Inter-generational pointers can cause floating garbage: even if both referrer and referent die, the inter-generational pointer will stop the referent from being reclaimed until the referrer’s generation is condemned.

**interior pointer**

Also known as

derived pointer.

An interior pointer is a pointer to memory (2) occupied by an object which does not point to the start location. Also called a derived pointer when it’s derived from a base pointer.

A pointer to an object will usually take as its value the address of the start of that object.

It is common to have interior pointers into string buffers or to embedded structures. A suballocator may place a header at the start of each object and pass on an interior pointer.

**Relevance to memory management**

In a system where interior pointers are used, the garbage collector must be able to mark an object as reachable without being told the start of the object. In a system where interior pointers are not used, the collector should either ignore them (in particular, if it is scanning conservatively) and not retain garbage because of them, or possibly report them as bugs.

**Opposite term**

base pointer.

**internal fragmentation** Internal fragmentation is where the memory manager allocates more for each allocation than is actually requested. There are three reasons for this: padding; buddy system; in-band headers.

See also:

external fragmentation.

**invalid page fault** An exception when using virtual memory resulting from an access to a virtual memory location for which no translation is defined.

This is usually an error, often, anachronistically, known as a segmentation violation.

**Similar term**

bus error.
In a virtual memory system, conventional page tables have an entry for every page in the virtual address space. An inverted page table has only as many entries as there are pages in physical memory, and uses a hash lookup to translate virtual addresses to physical addresses in nearly constant time.

The entire virtual address space of each process is described in an auxiliary structure, typically a B*-tree, that can efficiently store contiguous, sparse, or large address space descriptions. This auxiliary structure may itself be paged to avoid permanently consuming physical memory resources.

Inverted page tables are ideal for schemes that store information about objects in the high-order bits of their address. Such schemes may perform poorly with conventional page tables as the sparse address space may cause the page table structures to become so large as to compete with the program working set for physical memory.

Historical note

The Lisp Machine was an early workstation that used an inverted page table with hardware lookup. The UltraSPARC, PowerPC, and IA-64 architectures all include inverted page tables. Some implementations of these architectures have hardware-assisted lookup.

is-forwarded method

In the MPS

A format method that is called by a moving pool to determine if a formatted object is a forwarding object, and if so, to return the address where the object was moved to. See mps_fmt_isfwd_t.

6.2.10 Memory Management Glossary: K

A kilobyte is 1024 bytes.

keyword argument

An argument to a function call that’s identified by an associated keyword rather than by its position in the argument list.

In the MPS

Keyword arguments are passed to functions in the MPS interface as arrays of structures of type mps_arg_s. See Keyword arguments.

kilobyte

Also known as

kB.

A kilobyte is 1024 bytes.
See `byte (1)` for general information on this and related quantities.

The standard abbreviation is “kB”, but “KB” is often used by people unfamiliar with the metric system.

### 6.2.11 Memory Management Glossary: L

**large object area** An *allocation mechanism* designed to optimize the management of large *objects* by separating them from small ones.

Large objects, typically objects one or more orders of magnitude larger than the *virtual memory page* of a platform, can be costly to *allocate*, *initialize*, and *recycle*. By segregating those objects into a separate area, they can be managed using specific mechanisms that would be inefficient for smaller objects but which can reduce the cost of manipulating large ones.

Some example mechanisms:

1. In a *copying collector* large objects can be managed separately using a *mark-and-sweep collector* to avoid copying costs. See *Ungar (1988)*.

2. By aligning large objects on page boundaries, they can be *compacted* or copied by adjusting their *mapping* in *virtual memory*. See *Withington (1991)*.

3. Large objects may be split into a header and a body, where the header is fixed size and the bulk of the object is in the body. See *Ungar (1988)*.

4. By using a page-based *read barrier*, large objects can be initialized incrementally. For example, each page of the large object is initialized to zero when it is first read, rather than all at once at creation time.

5. In a copying collector, large objects can be copied incrementally using a similar technique (the new copy is initialized from the old copy). See *Baker (1978)*.

6. Large objects are often *leaf objects*, so do not need to be *scanned*, or are known to have a fixed *format* with only a few *references* so they can be scanned more efficiently by a specialized scanner.

7. Large objects often have longer than average *lifetimes*, so are not allocated in a *nursery space* of a *generational garbage collector*.

**large page**

See *huge page*.

**leaf object**

*Also known as*

*atomic object*.

A leaf object is an *object* that does not *reference* any other objects.

In a typed language, the compiler can often determine at compile time that certain types can be represented as leaf objects. Usually these types are either a *scalar data type* or a *vector data type* of scalars with bounded magnitude.

**Relevance to memory management**

If leaf objects can be identified, a *garbage collector* can make certain optimizations: leaf objects do not have to be *scanned* for references nor are *barriers (1)* needed to detect and maintain references in the object.
In the MPS

The **AMCZ (Automatic Mostly-Copying Zero-rank)** and **LO (Leaf Object)** pool classes are designed for the storage of leaf objects.

**leak**

See

*memory leak.*

**life**

See

*lifetime.*

**lifetime**

Also known as

*extent, life.*

The lifetime or extent of an *object* is the time for which the object is *live.*

See also:

*dynamic extent, indefinite extent.*

**LIFO-ordered first fit**

The **allocation policy** that always uses the most-recently *freed* (1) suitable *free block*. Commonly implemented by pushing freed blocks on the front of a *free block chain*, and then using *first fit* allocation on this chain. *Free* (1) can be very quick, depending on the *coalescing* policy.

This policy may suffer from severe *fragmentation* in the presence of short-lived large objects of a single size. As smaller objects are allocated, the free block chain fills up with fragments a little smaller than the large object size.

See also:

*FIFO-ordered first fit, address-ordered first fit.*

**Related publication**

*Wilson et al. (1995).*

**limited-field reference count**

Also known as

*sticky reference count.*

A *reference counting* technique whereby the field used to store the number of *references* to an *object* has a limited size. In particular, the field is not large enough to represent the maximum possible number of references to an object.

Using the observation that most objects are not referenced a great number of times, some systems that use reference counts only store the count accurately up to a certain maximum value. If an object has more references than the maximum then the count “sticks” at the maximum and is never decremented. Such objects are expected...
to be rare, but their memory (1) can never be reclaimed using reference counting. A separate (infrequently run) tracing garbage collector is often employed to reclaim this storage.

A degenerate form of limited-field reference counting is one-bit reference counting where an object is considered to be referenced either exactly once or many times.

**linear addressing** In linear addressing, addresses form a single, continuous address space. This term is used mostly in opposition to segmented addressing.

**Opposite term**

segmented addressing.

**live**

Also known as alive, active.

Memory (2) or an object is live if the program will read from it in future. The term is often used more broadly to mean reachable.

It is not possible, in general, for garbage collectors to determine exactly which objects are still live. Instead, they use some approximation to detect objects that are provably dead, such as those that are not reachable.

**Similar term**

reachable.

**Opposite term**

dead.

**See also:**

undead.

**load** To transfer data from memory (2) to a processor’s registers.

Load can also be used in the more general sense of moving data from a part of the memory hierarchy that is slow to access to one that is fast to access (For example, “it takes about 3 ms for the virtual memory system to load a page from disk on this system”). When used in this sense, the qualified term cache (2) load is common.

LOAD (or an abbreviation) is also commonly used in many processor architectures as the mnemonic name for the machine code instructions that are used primarily to make data accessible to the CPU (by loading the data into registers usually). In RISC architectures it is common for the load instructions to be the only means of making data accessible to the CPU; in CISC architectures it is common for a wide variety of instructions to implicitly or explicitly load data from memory.

**Opposite term**

store (1).

**locality of reference** Locality of reference is the extent to which successive accesses of nearby memory (1) locations are nearby in time; for example, a program that reads all the elements of a contiguous array in turn or that repeatedly uses the same memory variable has good locality of reference.

Good locality of reference interacts well with virtual memory and memory caches, as it reduces the working set and improves the hit rate.
There are a number of specialized senses of locality of reference in certain fields such as distributed systems; these are not covered in depth here.

**Relevance to memory management**

A *mutator* may exhibit predictable properties such as accessing in turn *objects* which were *allocated* in turn, or accessing in turn objects which have *references* to each other. An intelligent *allocator* or *copying garbage collector* can use this observation to improve locality of reference.

**Related publications**

*Grunwald et al. (1993), Wilson et al. (1992).*

---

**location**

See *

memory location.*

---

**location dependency**

**In the MPS**

A *location dependency* records the fact that the *client program* depends on the bit patterns of some *references* (and not merely on the identity of the *block* to which the reference refers), and provides a function *(mps_ld_isstale())* to find out whether a reference might have been changed because a block has been *moved*. See *Location dependency*.

---

**lock free**  A multi-threaded program is *lock free* if all schedules for the threads make progress: in particular, no schedule leads to deadlock. This is most easily implemented by avoiding taking locks.

---

**logical address**

See *

virtual address.*

---

**longword**

See *

doubleword.*

---

**6.2.12 Memory Management Glossary: M**

---

**machine word**

See *

word.*

---

**main memory**

Also known as
The main memory (or primary storage) of a computer is memory that is wired directly to the processor, consisting of RAM and possibly ROM.

These terms are used in contrast to mass storage devices and cache memory (although we may note that when a program accesses main memory, it is often actually interacting with a cache).

Main memory is the middle level of the memory hierarchy: it is slower and cheaper than caches, but faster and more expensive than backing store.

It is common to refer only to the main memory of a computer; for example, “This server has 128 GB of memory” and “OS X 10.8 requires at least 2 GB of memory”.

**Historical note**
Main memory used to be called core, and is now likewise often called RAM.

**Similar terms**
RAM, core, physical memory.

**malloc**
A function in the standard C library that performs dynamic allocation of memory.

Many people use “malloc” as a verb to mean “allocate dynamically”.

**Similar term**
allocate.

**Opposite term**
free.

**manual memory management**
In some systems or languages, it is up to the application program to manage all the bookkeeping details of allocating memory from the heap and freeing it when no longer required; this is known as manual memory management.

Manual memory management may be appropriate for small programs, but it does not scale well in general, nor does it encourage modular or object-oriented programming.

To quote Joyner (1996):

In C++ the programmer must manually manage storage due to the lack of garbage collection. This is the most difficult bookkeeping task C++ programmers face, that leads to two opposite problems: firstly, an object can be deallocated prematurely, while valid references still exist (dangling pointers); secondly, dead objects might not be deallocated, leading to memory filling up with dead objects (memory leaks). Attempts to correct either problem can lead to overcompensation and the opposite problem occurring. A correct system is a fine balance.

**Historical note**
Manual memory management was common in early languages, but garbage collection has been around since the late 1950s, in languages like Lisp. Most modern languages use automatic memory management, and some older languages have conservative garbage collection extensions.

**Opposite term**
In the MPS
Manual memory management can be used with pools such as MVFF (Manual Variable First Fit) via the functions mps_alloc() and mps_free().

mapped

Also known as committed.

A range of virtual addresses is said to be mapped (committed on Windows) if there is physical memory associated with the range.

Note that, in some circumstances, the virtual memory system could actually overcommit mapped memory.

Opposite term unmapped.

See also:
mapping, memory mapping, mmap.

In the MPS
The term committed is used. The function mps_arena_committed() returns the total committed memory for an arena.

mapping A mapping is a correspondence between a range of virtual addresses and some memory (or a memory-mapped object). The physical location of the memory will be managed by the virtual memory system.

Each page in a mapping could be paged out or paged in, and the locations it occupies in main memory and/or swap space might change over time.

The virtual address space can contain of a complex set of mappings. Typically, parts of the address space are mapped (have a mapping assigned), others are reserved but unmapped, and most of it is entirely unmapped.

Fig. 6.7: Virtual memory with different kinds of mappings.

See also:
backing store.

mark-compact Mark-compact collection is a kind of tracing garbage collection that operates by marking reachable objects, then compacting the marked objects (which must include all the live objects).

The mark phase follows reference chains to mark all reachable objects; the compaction phase typically performs a number of sequential passes over memory to move objects and update references. As a result of compaction, all the marked objects are moved into a single contiguous block of memory (or a small number of such blocks); the memory left unused after compaction is recycled.

Mark-compact collection can be regarded as a variation of mark-sweep collection, with extra effort spent to eliminate the resulting fragmentation. Compaction also allows the use of more efficient allocation mechanisms, by making large free blocks available.
mark-sweep, mark-and-sweep  Mark-sweep collection is a kind of **tracing garbage collection** that operates by **marking** reachable objects, then sweeping over memory and **recycling** objects that are unmarked (which must be unreachable), putting them on a free list.

The mark phase follows reference chains to mark all reachable objects; the sweep phase performs a sequential (address-order) pass over memory to recycle all unmarked objects. A mark-sweep collector doesn’t move objects.

Historical note
This was the first garbage collection algorithm, devised by John McCarthy for Lisp.

See also:
mark-compact.

Related publication
McCarthy (1960).

marking  Marking is the first phase (“the mark phase”) of the **mark-sweep** algorithm or **mark-compact** algorithm. It follows all references from a set of roots to mark all the reachable objects.

Marking follows reference chains and makes some sort of mark for each object it reaches.

Marking is often achieved by setting a bit in the object, though any conservative representation of a predicate on the memory location of the object can be used. In particular, storing the mark bit within the object can lead to poor locality of reference and to poor cache performance, because the marking phases ends up setting the dirty bit on all pages in the working set. An alternative is to store the mark bits separately: see bitmap marking.

See also:
sweep, compact.

MB

See
megabyte.

megabyte

Also known as
MB.

A megabyte is 1024 kilobytes, or 1048576 byte.

See byte for general information on this and related quantities.

memoization

See
caching (3).
memory

Also known as
storage, store.

memory or storage (or store) is where data and instructions are stored. For example, caches, main memory, floppy and hard disks are all storage devices.

These terms are also used for the capacity of a system to store data, and may be applied to the sum total of all the storage devices attached to a computer.

Historical note
“Store” is old-fashioned, but survives in expressions such as “backing store”.

memory
Memory refers to memory that can be accessed by the processor directly (using memory addressing instructions).

This could be real memory or virtual memory.

memory
A memory location; for example, “My digital watch has 256 memories.”

memory bandwidth Memory bandwidth (by analogy with the term bandwidth from communication theory) is a measure of how quickly information (expressed in terms of bits) can be transferred between two places in a computer system.

Often the term is applied to a measure of how quickly the processor can obtain information from the main memory (for example, “My new bus design has a bandwidth of over 400 Megabytes per second”).

memory cache
See cache.

memory hierarchy
See storage hierarchy.

memory leak
Also known as
leak, space leak, space-leak.

A memory leak is where allocated memory is not freed although it is never used again.

In manual memory management, this usually occurs because objects become unreachable without being freed.

In tracing garbage collection, this happens when objects are reachable but not live.
In *reference counting*, this happens when objects are *referenced* but not *live*. (Such objects may or may not be *reachable*.)

Repeated memory leaks cause the memory usage of a process to grow without bound.

**memory location**

*Also known as*

*location.*

Each separately-addressable unit of *memory* in which data can be stored is called a *memory location*. Usually, these hold a *byte*, but the term can refer to *words*.

**memory management**

*Also known as*

*storage management.*

Memory management is the art and the process of coordinating and controlling the use of *memory* in a computer system.

Memory management can be divided into three areas:

1. Memory management hardware (*MMUs*, *RAM*, etc.);
2. Operating system memory management (*virtual memory*, *protection*);
3. Application memory management (*allocation*, *deallocation*, *garbage collection*).

Memory management hardware consists of the electronic devices and associated circuitry that store the state of a computer. These devices include RAM, MMUs (memory management units), cache, disks, and processor registers. The design of memory hardware is critical to the performance of modern computer systems. In fact, *memory bandwidth* is perhaps the main limiting factor on system performance.

Operating system memory management is concerned with using the memory management hardware to manage the resources of the *storage hierarchy* and allocating them to the various activities running on a computer. The most significant part of this on many systems is *virtual memory*, which creates the illusion that every process has more memory than is actually available. OS memory management is also concerned with *memory protection* and security, which help to maintain the integrity of the operating system against accidental damage or deliberate attack. It also protects user programs from errors in other programs.

Application memory management involves obtaining *memory* from the operating system, and managing its use by an application program. Application programs have dynamically changing storage requirements. The application *memory manager* must cope with this while minimizing the total CPU overhead, interactive pause times, and the total memory used.

While the operating system may create the illusion of nearly infinite memory, it is a complex task to manage application memory so that the application can run most efficiently. Ideally, these problems should be solved by tried and tested tools, tuned to a specific application.

The Memory Management Reference is mostly concerned with application memory management.

See also:

*automatic memory management*, *manual memory management*.

**Memory Management Unit**

See

*MMU.*
memory manager  The memory manager is that part of the system that manages memory, servicing allocation requests, and recycling memory, either manually or automatically.

The memory manager can have a significant effect on the efficiency of the program; it is not unusual for a program to spend 20% of its time managing memory.

Similar terms

allocator, collector (1).

See also:

memory management.

memory mapping

Also known as

file mapping.

Memory mapping is the technique of making a part of the address space appear to contain an “object”, such as a file or device, so that ordinary memory accesses act on that object.

The object is said to be mapped to that range of addresses. (The term “object” does not mean a program object. It comes from Unix terminology on the mmap man page.)

Fig. 6.8: An address space with a range mapped to part of an object.

Memory mapping uses the same mechanism as virtual memory to “trap” accesses to parts of the address space, so that data from the file or device can be paged in (and other parts paged out) before the access is completed.

Historical note

File mapping is available on most modern Unix and Windows systems. However, it has a much longer history. In Multics, it was the primary way of accessing files.

See also:

mapped.

memory protection

See

protection.

message

In the MPS

A data structure which the MPS uses to communicate with the client program. See Messages.

message queue

In the MPS
A queue of messages posted by an arena. It can be queried by calling \texttt{mps_message_poll()}, \texttt{mps_message_queue_type()}, or \texttt{mps_message_get()}. See \textit{Messages}.

\section*{message type}

\textbf{In the MPS}

A value of type \texttt{mps_message_type_t} describing the type of a message. There are three message types: \texttt{mps_message_type_finalization()}, \texttt{mps_message_type_gc()}, and \texttt{mps_message_type_gc_start()}. See \textit{Messages}.

\section*{misaligned}

See \textit{unaligned}.

\section*{miss}

A miss is a lookup failure in any form of cache (3), most commonly at some level of a storage hierarchy, such as a cache (1) or virtual memory system.

The cost of a miss in a virtual memory system is considerable: it may be five orders of magnitude more costly than a hit. In some systems, such as multi-process operating systems, other work may be done while a miss is serviced.

\textbf{Opposite term}

hit.

See also: miss rate.

\section*{miss rate}

At any level of a storage hierarchy, the miss rate is the proportion of accesses which miss.

Because misses are very costly, each level is designed to minimize the miss rate. For instance, in caches (1), miss rates of about 0.01 may be acceptable, whereas in virtual memory systems, acceptable miss rates are much lower (say 0.00005). If a system has a miss rate which is too high, it will spend most of its time servicing the misses, and is said to thrash.

Miss rates may also be given as a number of misses per unit time, or per instruction.

\textbf{Opposite term}

hit rate.

\section*{mmap}

mmap is a system call provided on many Unix systems to create a mapping for a range of virtual addresses.

\section*{MMU}

\textbf{Also known as}

Memory Management Unit.

The MMU (Memory Management Unit) is a hardware device responsible for handling memory (2) accesses requested by the main processor.

This typically involves translation of virtual addresses to physical addresses, cache (1) control, bus arbitration, memory protection, and the generation of various exceptions. Not all processors have an MMU.
mostly-copying garbage collection  A type of semi-conservative tracing garbage collection which permits objects to move if no ambiguous references point to them.

The techniques used are a hybrid of copying garbage collection and mark-sweep.

Mostly-copying garbage collectors share many of the benefits of copying collectors, including compaction. Since they support ambiguous references they are additionally suitable for use with uncooperative compilers, and may be an efficient choice for multi-threaded systems.

Related publications

In the MPS
The AMC (Automatic Mostly-Copying) pool class implements mostly-copying garbage collection.

mostly-exact garbage collection

See
semi-conservative garbage collection.

mostly-precise garbage collection

See
semi-conservative garbage collection.

moving garbage collector, moving memory manager  A memory manager (often a garbage collector) is said to be moving if allocated objects can move during their lifetimes.

Relevance to memory management
In the garbage collecting world this will apply to copying collectors and to mark-compact collectors. It may also refer to replicating collectors.

Similar term
copying garbage collection.

Opposite term
non-moving garbage collector.

mutable  Any object which may be changed by a program is mutable.

Opposite term
immutable.

mutator
Also known as

client program.

In a garbage-collected system, the part that executes the user code, which allocates objects and modifies, or mutates, them.

For purposes of describing incremental garbage collection, the system is divided into the mutator and the collector (2). These can be separate threads of computation, or interleaved within the same thread.

The user code issues allocation requests, but the allocator code is usually considered part of the collector. Indeed, one of the major ways of scheduling the other work of the collector is to perform a little of it at every allocation.

While the mutator mutates, it implicitly frees memory (1) by overwriting references.

Historical note

This term is due to Dijkstra et al. (1976).

Opposite term

collector (2).

In the MPS

The MPS documentation uses the term client program to refer to the mutator.

6.2.13 Memory Management Glossary: N

nailing

See

pinning.

natural alignment  Natural alignment is an alignment constraint such that all objects must be aligned to an address that is a multiple of their size.

Natural alignment is not usually required for objects larger than a word or grain, which usually only need to be word- or grain-aligned.

See also:

alignment, padding.

In the MPS

The MPS platform interface defines the C preprocessor macro MPS_PF_ALIGN to be the natural alignment of the platform.

nepotism  In generational garbage collection nepotism is the tendency for dead objects in old generations to preserve younger dead objects that are referenced by them. In other words, dead parents can cause their children to get promoted.

This happens when an object gets promoted to an old generation and dies there, but does not get reclaimed because the generation it is in does not get considered for garbage collection very often. The old object might
refer to objects in younger generations that are also dead; until the old object is reclaimed the younger objects will be preserved by virtue of the reference from the older, assumed alive, object.

This is a form of floating garbage introduced by partitioning the objects into generations.

**next fit** A variant of the first fit allocation mechanism that uses a roving pointer on a circular free block chain. The pointer is advanced along the chain when searching for a fit. Thus each allocation begins looking where the previous one finished. The rationale is to avoid creating an accumulation of small fragments at the head of the free block chain, which would have to be examined on every allocation.

There are several variants, according to the order of blocks on the free block chain. The most common variant is address-ordered next fit.

This has a tendency to spread related objects out in memory, and also gives quite poor locality for the allocator (as the roving pointer rotates around memory, the free blocks touched are those least-recently used).

See also:

first fit, allocation mechanism.

**Related publication**


**new space, newspace**

See
tospace.

**node** In a graph, a node is a representation of an object at the junction of zero or more edges.

Opposite term

dge.

See also:

graph.

**non-moving garbage collector, non-moving memory manager** A memory manager is said to be non-moving if allocated objects do not move during their lifetimes.

Non-moving memory management techniques include mark-sweep collection, reference counting, and most kinds of manual memory management.

Opposite term

moving garbage collector.

**nursery generation**

See

nursery space.

**nursery space**

Also known as

nursery generation.
In **generational garbage collection**, the *nursery generation* or *space* is the area used for new *allocation*.

The size of the nursery space must be chosen carefully. Often it is related to the size of *physical memory* (1).

**In the MPS**

By default, a garbage-collected *pool* allocates into the first *generation* in its *generation chain*, but this can be altered by setting the `MPS_KEY_GEN` keyword argument when calling `mps_pool_create_k()`.

### 6.2.14 Memory Management Glossary: O

| A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z |

**object**

Also known as

*cell*.

In *memory management*, we use the term *object* or *cell* to mean a contiguous *block* of *memory* (2) forming a single logical structure.

Objects are the units of *allocation, deallocation*, etc. No connection to an object-oriented system is implied.

**In the MPS**

The MPS documentation generally reserves the term *object* for *formatted objects*. For units of allocation in general, it uses the term *block*.

**object format**

**In the MPS**

A data structure provided by the *client program* which describes the format of *objects* allocated in a *pool*. The MPS uses the *format methods* to find *references* in an object, replace an object with *padding*, replace an object with a *forwarding marker*, and other essential *garbage collection* tasks. See *Object formats*.

**object pointer**

In the *C* programming language, a *pointer* to an *object*, as distinct from a *function pointer*. The *C* programming language guarantees that you can cast any object pointer to `void *` and back without losing information.

**Opposite term**

*function pointer*.

**off-white**

Also known as

*ecru*.

In a *treadmill garbage collector*, the *color* off-white is used to describe *objects* which are *free* (3). *Baker (1992c)* used the term *ecru*.

**Opposite terms**
The one-bit reference count is a heuristic mechanism that lets a program test, at low cost, whether an object is dead.

The one-bit reference count is a special case of the limited-field reference count. A single bit in an object, called the MRB (Multiple Reference Bit), is cleared when the object is allocated. Whenever another reference to the object is created, the bit is set. Thus, MRB=0 indicates that there is exactly one reference to the object, and MRB=1 indicates that there may be more than one reference to the object.

The MRB can be stored in the reference rather than in the object; doing so reduces the number of memory accesses due to MRB checking and setting. When a reference is copied, the copy’s MRB is set. If the MRB in the old reference is 0, it also needs to be set. Setting the MRB in the old reference requires that the program knows the location the old reference came from, and that it can prove that location has not since been overwritten with other data.

The one-bit reference count is used by a compiler to augment an object lifetime analysis. When compile-time analysis predicts that a particular object may be dead (typically because the variable that references the object is dead), the compiler can generate code that will check the object’s MRB at run-time. If the MRB is 0, then the object is dead.

Using a one-bit reference count does have a cost: the MRB uses space that could sometimes be put to other use, and the MRB must be set every time the number of references to the object increases. The one-bit reference count is cheaper than other kinds of reference counting, however, since the space cost is only one bit and the reference count is not adjusted when references are destroyed.

Historical note
The one-bit reference count was suggested by Friedman & Wise (1977). Storing the MRB in the reference was suggested by Stoye, Clarke, and Norman (1984).

Related publication
Jones et al. (2012).

Opaque type

In the MPS
In the MPS interface, an opaque type is a pointer to an incomplete structure type. The client programs must not rely on the details of its implementation. For example, the type mps_arena_t is an alias for struct mps_arena_s *, but the implementation of struct mps_arena_s is not public. See Interface conventions.

Opposite terms
derived type, transparent type.
**out parameter**  A function parameter that points to a location for the caller to receive data from the function.

_Opposite term_

*in parameter.*

**In the MPS**

Out parameters are given names ending with _o_. See Interface conventions.

**out-of-band header**  In some memory managers, each allocated block has additional information (such as the size of the block or a tag) stored in a separate block; this is called an out-of-band header.

_Opposite term_

*in-band header.*

**overcommit**  In some circumstances, although a range of virtual addresses has been mapped as far as the user program is concerned, the physical storage might not be allocated until it is accessed. This is called overcommitting.

Overcommitting shares swap space resources more flexibly, especially when crude suballocators are involved, but it can lead to an out-of-resource error during a memory access; few environments deal with this situation gracefully.

Unix systems such as IRIX and AIX can do this on sbrk and mmap calls.

**overwriting error**

_Also known as_

.bounds error.

An overwriting or bounds error occurs when the programmer intends his program to write to a particular block of memory, but a program error causes the program to write outside the bounds of that block.

_See also:_

fencepost.

### 6.2.15 Memory Management Glossary: P

**A B C D E F G H I J K L M N O P Q R S T U V W X Y Z**

**padding**  Padding is redundant memory within the memory allocated to an object. It is usually inserted because of alignment restrictions on the fields of the object or on the object itself.

Padding is a form of internal fragmentation.

**padding method**

_In the MPS_

A format method that is called by a moving pool to create a padding object. See mps_fmt_pad_t.

**padding object**

_In the MPS_
A formatted object that consists of padding. One of three types of formatted objects, the other two being client objects and forwarding objects.

page A virtual memory system usually deals with memory (1) blocks of fixed size as units for paging. These are known as pages.

Pages are often 4 kB or 8 kB in size. This size is determined by the addressing hardware of the machine.

page fault An exception when accessing virtual memory, usually resulting in a page being fetched from disk.

A page fault is an exception occurring during the translation of virtual addresses to physical addresses. “Page fault” usually means an access to a page that has been paged out and hence requires fetching from disk, but it is sometimes also used to mean invalid page fault or protection fault.

See also:
paging, paged in, paged out, read fault, write fault.

page marking Page marking is a form of card-marking where the card is the same size as a page

page protection

See protection.

Many operating systems support protection of memory (2) pages. Individual pages may be protected against a combination of read, write or execute accesses by a process.

page table In a virtual memory system, it is common to map between virtual addresses and physical addresses by means of a data structure called a page table.

The page number of an address is usually found from the most significant bits of the address; the remaining bits yield the offset of the memory location within the page. The page table is normally indexed by page number and contains information on whether the page is currently in main memory, and where it is in main memory or on disk.

Conventional page tables are sized to the virtual address space and store the entire virtual address space description of each process. Because of the need to keep the virtual-to-physical translation time low, a conventional page table is structured as a fixed, multi-level hierarchy, and can be very inefficient at representing a sparse virtual address space, unless the allocated pages are carefully aligned to the page table hierarchy.

See also:
inverted page table.

paged in In a virtual memory system, memory (2) is described as paged in if it is available in physical memory (1).

Similar term
swapped in.

Opposite term
paged out.

See also:
paging.

paged out In a virtual memory system, memory (2) is described as paged out if it is not available in physical memory (1).
Similar term

swapped out.

Opposite term

paged in.

See also:

paging.

paging  In a virtual memory system, paging is the act of transferring pages between physical memory (1) and backing store (usually disk).

When pages need to be paged out, a heuristic is used to select ones that will not be needed soon; “least recently used” is a popular one.

Similar term

swapping.

See also:

paged in, paged out.

palimpsest  A block of memory (2) that has been allocated, freed (1) (or reclaimed), and then allocated again. Such memory may contain data from the previous use if portions of it remain uninitialised.

This commonly occurs on the stack, especially if the compiler allocates large stack frames in anticipation of allocating data structures on the stack.

If the palimpsest is being scanned conservatively, such left-over data may cause unreachable objects to appear reachable and thus become floating garbage. If it is scanned precisely, such left-over data, if treated as pointers, is a bug.

parallel garbage collection

Also known as

concurrent garbage collection.

A parallel or concurrent collector (2) executes simultaneously with the mutator, usually on a multi-processor machine.

Concurrent garbage collection must cope with the mutator changing objects while collection occurs. The problem is similar to that of incremental GC, but harder. The solution typically involves barriers (1).

Similar term

incremental.

See also:

replicating garbage collector.

Related publications

parked state

In the MPS

One of the three states an arena can be in (the others being the clamped state and the unclamped state). In the parked state, no garbage collection is in progress, no object motion occurs and the staleness of location dependencies does not change. Call mps_arena_park() or mps_arena_collect() to put an arena into the parked state.

perfect fit  If an allocation request is satisfied exactly from a free block with no fragmentation, this is said to be a perfect fit.

See also:
free block, allocation mechanism, best fit.

phantom reachable, phantomly reachable  In Java, an object is phantom reachable if it is neither strongly nor softly nor weakly reachable and has been finalized and there is a path from the roots to it that contains at least one phantom reference.

When the Java collector (1) determines that an object is phantom reachable, the reference objects containing the phantom references are enqueued.

The Java specification says that the phantom reference is not cleared when the reference object is enqueued, but actually, there’s no way in the language to tell whether that has been done or not. In some implementations, JNI weak global references are weaker than phantom references, and provide a way to access phantom reachable objects.

See also:
reachability.

Related links
Class java.lang.ref.PhantomReference, Reference Objects and Garbage Collection.

phantom reference  In Java terminology, phantom reference is used to mean a reference encapsulated in a reference object of class PhantomReference.

Phantom references form one of three kinds of weak reference (1) in Java. They are handy for performing clean-ups after an object has died and been finalized.

See also:
phantom reachable.

Related link
Class java.lang.ref.PhantomReference, Reference Objects and Garbage Collection.

physical address

Also known as
absolute address.

Physical addresses are used to index into physical memory (1). On some systems, they are called absolute addresses.

In a virtual memory system the application program handles virtual addresses and these are translated to physical addresses by the MMU.
physical address space  The physical address space is the space of physical addresses.

physical memory\(^{(1)}\)

Also known as
real memory.

Physical memory is memory\((1)\) that is wired to directly to the processor, addressable by physical address. This term is basically synonymous to main memory, but is used in contrast to virtual memory and backing store. While modern computers usually have lots of virtual memory, performance is still closely related to the quantity of physical memory available. If a system has insufficient physical memory, it may thrash.

Similar term
main memory.

physical memory\(^{(2)}\)

Also known as
physical storage.

Physical memory is memory\((1)\) on physical storage devices, such as RAM or disks. This term is often contrasted to virtual address space that might not be mapped to any actual storage.

Similar term
memory\((1)\).

physical storage

See
physical memory\((2)\).

pig in the python

Also known as
pig in the snake.

In a generational collector, when long-lived objects are allocated in nursery space, collection effort will be wasted as those objects survive and are promoted from generation to generation. This is especially noticeable in a copying collector, where long-lived objects will be copied many times. This difficulty is similar to that of a python which swallows its prey whole and is somewhat immobilized as it digests it.
Modern collectors permit objects to be allocated directly into appropriate generations or pools to avoid this problem. Long-lived objects can be allocated directly into long-term generations. Large objects can be allocated directly into pools with special support for large objects (such as copying by remapping, incremental copying, or not copying at all).

See also:

generational garbage collection.

In the MPS
A pool can be configured to allocate into a specific generation in its generation chain by setting the MPS_KEY_GEN keyword argument when calling mps_pool_create_k().

pig in the snake

See
pig in the python.

pinning

Also known as
nailing.

In copying garbage collection, an object may not be movable because it is the target of an ambiguous reference or because it is referenced by foreign code that does not co-operate with the collector. Such an object is said to be pinned.

placement policy

See
allocation policy.

platform

In the MPS
The term platform is used to refer to the combination of operating system, processor architecture, and compiler. See Platforms.

plinth

In the MPS
The plinth is a program module providing the MPS with all the support functions it needs from the execution environment. The plinth removes the need for external libraries, by getting the support from the client program. See Plinth.

pointer  
Pointer data types represent a reference to an object or a location.

Pointers may be specialized by the type of the object referred to.

Typically, pointers are represented by an address, but they can be more complicated when they need to carry more information. For example, when the referent is smaller than a word, an offset within the word might be needed.
Similar terms

reference, address.

See also:

tag.

pool

In the MPS
A pool is responsible for requesting memory from the arena and making it available to the client program via mps_alloc() or via an allocation point. Multiple pools can coexist in one arena. Pools belong to the type mps_pool_t. See Pools and the Pool reference.

pool class

In the MPS
A value of type mps_pool_class_t describing a class of pools that manage memory according to particular policy. See Pool reference.

precise garbage collection

See

exact garbage collection.

precise reference

See

exact reference.

precise root

See

exact root.

premature free

Also known as
use after free.

A premature free or use after free occurs when memory (2) is deallocated, but is later accessed.

Under manual memory management, this usually occurs when one part of a program decides it has finished using a memory block, and is unaware that another part of the program is still using it. This is rare under automatic memory management.

See also:

double free.

premature promotion
See

premature tenuring.

premature tenuring

Also known as

premature promotion.

When a short-lived object allocated in a generational garbage collector is promoted (due to poor timing) into a less-frequently collected generation. This prematurely tenured object may become garbage very soon after promotion, but will not be reclaimed for some time because it is now in a less frequently collected generation.

This problem is essentially due to quantization error: all objects in a generation are treated as if they have the same age, even though they range from as old as the previous promotion cycle to new-born.

Modern collectors offer several remedies for premature tenuring. If the client program knows that it is entering a phase that will create many short-lived objects, it can forestall all promotion until it knows it is done with those objects. Thus no objects will be prematurely promoted: they will all be seen as garbage. Another solution is to create buckets within generations to more accurately classify objects by age and only promote those which have reached a certain minimum.

primary storage

See

main memory.

promotion

Also known as

tenuring.

Promotion or tenuring is the act of moving an object from its current generation to an older one (one that contains objects that are expected to survive longer).

“Tenuring” is used particularly about promotion to the oldest generation.

See also:

generational garbage collection.

protectable root

In the MPS

A root which the MPS may protect with a write barrier. A protectable root is created by specifying the root mode MPS_RM_PROT when calling a registration function such as mps_root_create().

protection

Also known as

memory protection, page protection.

Many operating systems support protection of memory pages. Individual pages may be protected against a combination of read, write or execute accesses by a process.
A process which attempts a protected access will trigger a protection fault. Protection is typically implemented in hardware by the MMU as part of the support for virtual memory.

Pages can be protected for a number of reasons: a generational or incremental garbage collector may want to place barriers (1) on pages; an operating system may want to protect pages for security, or to implement “copy-on-write” or “demand-zero-filled” pages.

See also:
read fault, write fault.

Related publications
Appel et al. (1988), Singhal et al. (1992), Hosking & Moss (1993).

protection exception

See
protection fault.

protection fault

Also known as
barrier hit, protection exception, protection violation.

A protection fault is an exception or trap which occurs when a process attempts to access memory (2) which has been protected.

Relevance to memory management
Some garbage collectors use handlers for protection faults to provide barriers (1).

See also:
segmentation violation, General Protection Fault.

protection violation

See
protection fault.

6.2.16 Memory Management Glossary: Q

quadword
A quadword is a unit of memory consisting of four adjacent words.

Historical note
In Digital’s Alpha architecture, a quadword of 64 bits was actually the natural word size, but the term word was still used for the 16-bit unit, for compatibility with the PDP-11.

See also:
doubleword.
6.2.17 Memory Management Glossary: R

RAM

Also known as
random access memory.

RAM (random access memory) is a type of physical memory(2) that can be read from and written to.

Similar term
main memory.

See also:
ROM, static RAM, dynamic RAM.

random access memory

See
RAM.

ramp allocation

In the MPS
An allocation pattern indicating to the MPS that most of the blocks allocated after the call to mps_ap_alloc_pattern_begin() are likely to be dead by the time of the corresponding call to mps_ap_alloc_pattern_end(). See Ramp allocation.

rank

In the MPS
A value of mps_rank_t indicating whether a reference is ambiguous (mps_rank_ambig()), exact (mps_rank_exact()) or weak (mps_rank_weak()).

rash

In the MPS
A variety in which no MPS functions assert that their data structures are valid. Select it by defining CONFIG_VAR_RASH. Compare cool and hot.

raw

See
unwrapped.

reachable
An object is reachable if it is referred to by a root, or is referred to by a reachable object; that is, if it can be reached from the roots by following references.

Reachability is used as an approximation to liveness in tracing garbage collection.
In Java, the reference objects together with ordinary references and finalization generate a hierarchy of reachability that guides the collector on what to do when an object is about to die. There are six strengths:

1. strongly reachable;
2. softly reachable;
3. weakly reachable;
4. finalizable;
5. phantom reachable;
6. unreachable.

Basically, an object is only as reachable as the weakest link in the strongest path from the roots. Note that the Java specification’s description of the reachabilities is a bit patchy, but that’s what it intends. It is unspecified where Java Native Interface’s weak global references fit into this.

**Similar term**

live.

**Opposite term**

unreachable.

**Related links**

Package java.lang.ref, Reference Objects and Garbage Collection.

**read barrier**  A read barrier is a block on reading from certain memory locations by certain threads or processes.

**Relevance to memory management**

Read barriers are used for incremental or concurrent garbage collection.

**See also:**

write barrier.

**read fault**  An exception which occurs when reading from an address in virtual memory.

This is probably either a page fault, an invalid page fault or a protection fault.

**Similar term**

segmentation violation.

**See also:**

write fault.

**read-only memory**

See

ROM.
**real memory**\(^{(1)}\) A system with no *virtual memory* capability can be said to have *real memory*.

**Historical note**
On older architectures, programs could only directly access data in real memory. Where this was inefficient, they had to store data on disk, and sometimes had alternate portions of program image called *overlays*.

**Opposite term**
*virtual memory*.

**real memory**\(^{(2)}\)

**See**
*physical memory*\(^{(1)}\).

**reclaim** *Reclaiming an object* or the *memory*\(^{(1)}\) occupied by it is making it available for reuse after the object is no longer needed.

This word is usually used only in connection with *automatic memory management*.

**Similar term**
*recycle*.

**recycle** *Recycling memory*\(^{(1)}\) means making it available for reuse after it has been occupied by an *object* that is no longer needed.

In simple cases, this might simply involve adding a *memory*\(^{(2)}\) *block* to the *free list*. Another possibility is *unmapping* memory so that the *backing store* can be allocated to another process.

**Similar term**
*reclaim*.

**reference** In memory management, *a reference* is the general term for a link from one *object* to another. Some programming languages have more specific meanings for the term.

The terms “*pointer*” and “reference” are often interchangeable, but some programming languages differentiate the two in subtle ways.

**Similar terms**
*address*, *pointer*.

**In the MPS**
A reference is represented in the *C* interface by a value of type `mps_addr_t` (an alias for `void *`) which points to a *memory location* within the object (typically the base of the object, but for objects with *headers* this may not be the case). The pointer returned by `mps_alloc()` and `mps_reserve()` is a reference to the object allocated.

The *client program* is free to represent references as it chooses (for example, with *tags*), provided that during *scanning* it is able to decode a reference from its representation into the MPS interface representation and encode a reference from the MPS into its representation.
Reference counting systems perform automatic memory management by keeping a count in each object, usually in a header, of how many references there are to the object. Objects to which there are no references cannot be accessed by the mutator; they are therefore dead and may be reclaimed.

The reference count is incremented for each new reference, and is decremented if a reference is overwritten, or if the referring object is recycled. If a reference count falls to zero, then the object is no longer required and can be recycled.

There are four main problems with simple reference counting:

1. The reference count field usually has to have limited size, and the system therefore breaks down if the number of possible references to an object is unbounded;
2. Reference counting involves an operation on every modification of a pointer, which increases code size, increases demand for memory bandwidth, decreases locality of reference and can be a serious performance penalty (especially in multi-threaded environments where reference count updates require synchronization);
3. Every object needs to be slightly larger in order to store the reference count;
4. If any objects are part of a cyclic data structure then they will always have a non-zero reference count, and hence won’t be reclaimed when they are dead.

Reference counting has the advantage that it can reclaim objects promptly, and for this reason it is often used to reclaim non-cyclic data structures in file systems, databases and operating system kernels. When there is a possibility of cyclic data structures, reference counting is sometimes used together with a tracing garbage collector that runs infrequently. Such combinations are generally less efficient than using a tracing collector by itself, but the promptness of reference counting may be important.

Pauses due to reference counting are typically fairly short, and it may be appropriate as a form of incremental garbage collection. But removing a single reference may cause the recycling of a large number of objects at once, so it is not suited to real-time systems where minimum pause times must be guaranteed. There are more complex variations of the technique that address this problem.

Reference counting is often used because it can be implemented without any support from the language or compiler. In C++ this can be encapsulated in a class, using a smart pointer. However, it would normally be more efficient to use a tracing garbage collector instead. The performance of reference counting can be improved substantially with compiler support, using refinements such as deferred reference counting, which has been successfully used in Smalltalk and other languages.

Despite the problems, reference counting is often used for distributed garbage collection. This is because refinements such as weighted reference counting require less inter-process communication than tracing.

See also:

limited-field reference count, one-bit reference count.

Reference object In Java, a reference object (java.lang.ref.Reference) encapsulates a reference to some other object, in order to make the garbage collector handle it specially. In particular, a Java program can use this to detect when the referent becomes unreachable.

Basically, the encapsulated reference is a weak reference; it will be cleared by the collector when all other references to the referent have disappeared. However, in order to better control what happens at the end of an object’s lifetime, Java 1.2 provides three classes of reference objects, each with its own peculiarities: SoftReference, WeakReference, and PhantomReference. Each of these classes has its uses in managing memory. The reference objects together with ordinary references and finalization generate a hierarchy of reachability that guides the collector on what to do when an object is about to die.
A reference object can be registered with a queue, and it will be enqueued when the collector determines that the referent is softly, weakly or phantom reachable, as the case may be. A program can use these queues to perform some action when an object is dying. This allows finer control than the older finalization mechanism alone.

**Historical note**

This feature was introduced in Java 1.2 (confusingly, part of the Java 2 Platform).

**See also:**

soft reference, weak reference (2), phantom reference.

**Related link**

Package java.lang.ref, Reference Objects and Garbage Collection.

**Related publication**

Dybvig et al. (1993).

**region inference**  Region inference is a technique for determining when objects become dead (even if they are reachable) by a static analysis of the program.

Region inference infers a region for each object. When a region dies, all the objects in it are known to be dead, whether reachable or not. Regions obey a strict stack discipline; that is, when a region dies, all younger regions also die. In this way, region inference occupies a middle ground between stack allocation and heap allocation.

**Related publication**

Tofte & Talpin (1997).

**register**  A register is a small unit of memory that is attached to a processor and accessible very quickly. Registers typically form the highest level of a computer’s storage hierarchy.

**Relevance to memory management**

In some programs (for example, those compiled by typical C or C++ compilers), a subset of the registers is always accessible by the mutator and so forms a root.

**In the MPS**

The scan method for the root containing the registers is hard to write (it depends on the operating system, the processor architecture, and in some cases the compiler), so the MPS provides (on its supported platforms) the function mps_stack_scan_ambig().

**register set partitioning**  Run-time systems for garbage-collected languages sometimes partition the set of machine registers a priori into two categories: those always traced and updated by the garbage collector and those ignored by it.

The former are always maintained in a format understood by the collector; the latter are never used to hold references to collectable objects. More complicated schemes are also possible.

This partitioning provides a separation of concerns between the compiler and the garbage collector. The compiler can generate code that produces values the garbage collector would not be able to handle (say, because they have no tags), as long as those values are kept in the ignored registers. The garbage collector can trust that the registers it looks at always contain valid data, and can perform exact garbage collection.
Register set partitioning increases the demand for registers (register pressure), but may reduce the amount of boxing needed.

**relocation**  Relocating means moving data from one location to another and updating all references.

Relocation is often performed to avoid external fragmentation.

Program loading sometimes relocates code and static data.

**Similar term**

moving.

See also:

compaction, moving memory manager.

**remembered set**  A remembered set is the technique of keeping a separate list of interesting references between two sets of objects, so you don’t have to find them by scanning.

Many memory management algorithms depend on partitioning the objects and require special handling for references between partitions. Keeping track of such references in a remembered set eliminates the need to scan the originating partition to find them.

A typical use in generational garbage collection is remembering references from an older generation to a younger one.

**Related publications**

Ungar (1984), Jones et al. (2012).

**remote reference**

In the MPS

A reference that logically belongs to a formatted object and so must be fixed when the object is scanned, but which is not stored within the block containing the object. (For example, in an auxiliary table of some sort.)

The MPS does not generally support remote references because those references may be protected and so if scan method attempts to fix them this will hit a barrier (1) and cause a re-entrant call to the MPS.

**replicating garbage collector**  A variant of copying garbage collection, which does not destroy the original object when making a copy.

This is useful in an incremental or concurrent collector (1), as no read barrier is required: the mutator can continue to use old objects. The collector uses a write barrier to replicate the writes to the new copies.

See also:

copying garbage collection, broken heart.

**Related publications**


**reserved**  In a virtual memory system, it is usually possible to hold range of virtual addresses reserved without making it mapped.
Reserving addresses prevents other components of the program using the same addresses, without consuming
*swap space*. This technique is often used in *BIBOP* schemes, where one might want to reserve a large amount
of *address space* but only sparsely map it.

On some systems there are special calls for reserving; on others one can create *mappings* that don’t need *backing
store*. For example, on some Unix systems, `mmap /dev/zero` with no access.

See also:

- *mapping*, *mmap*.

In the MPS

The function `mps_arena_reserved()` returns the total address space reserved by an arena.

---

**resident**  In a *cache* (2) system, that part of the cached storage which currently has a copy in the cache is called
*resident*. Ideally, the *working set* should be resident.

See also:

- *cache* (2), *storage hierarchy*, *resident set*.

**resident set**  In a *virtual memory* system, a process’ resident set is that part of a process’ *address space* which is
currently in *main memory*. If this does not include all of the process’ *working set*, the system may *thrash*.

---

**result code**

In the MPS

A value returned from an MPS function, represented by the type `mps_res_t`. The result code `MPS_RES_OK` indicates success; other values indicate errors. See *Error handing*.

---

**resurrection**  An object is said to have been *resurrected* if it was determined to be *finalizable* by the *garbage collector*
(that is, the only thing keeping it alive was the fact that it required finalization), but then a new *strong reference*
was created to it.

This can happen via a *weak reference* (1) or by the finalization procedure storing a permanent copy of its
reference to the object.

In the MPS

See *Finalization*.

---

**retention**  The failure to *recycle floating garbage*, due to some approximation or optimization in the *garbage collec-
tor*; also the amount of memory thus retained.

Related publication

*Boehm (2001)*.

---

**ROM**

Also known as

*read-only memory*.

ROM (read-only memory) is a type of *physical memory* (2) that can be read from, but not written to. The
contents of ROM are usually set in the factory.

See also:
**Memory Pool System Documentation, Release 1.115.0**

**RAM.**

**Root**  In *tracing garbage collection*, a root holds a *reference* or set of references to *objects* that are *a priori reachable*. The *root set* is used as the starting point in determining all reachable data.

Roots basically comprise the references in the state of the *mutator*. Typical roots are global variables, other *static* data, and the *control stack*.

See also:

*weak root, strong root, ambiguous root, exact root.*

**In the MPS**

See *Roots.*

**Root description**

**In the MPS**

The *arena* uses root descriptions to find *references* within the *client program’s roots*. Root descriptions belong to the type *mps_root_t*.

**Root mode**

**In the MPS**

A value of type *mps_rm_t* describing whether a *root* is *constant, protectable*, or both. The root mode tells the MPS whether it may place a *barrier* (1) on the root.

**Root set**  The *root set* is the collection of *roots* that the *mutator* declares to the *collector* (2).

See also:

*garbage collection.*

**6.2.18 Memory Management Glossary: S**

**A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z**

**sbrk**  *sbrk* is a Unix library function that adjusts the limit of the data segment; this limit is known as the *break*.

*sbrk* returns the previous value of the break, so *sbrk (0)* is a common idiom for getting the current value.

Note that, if you use *brk*, you probably can’t safely use *sbrk* as well, because it may store the last value of the break in a private variable.

**Scalar data type**  A scalar data type is a type that is representable in a single dimension and whose objects have only magnitude as value.

Examples of scalar data types include: integers, floating-point numbers, enumerations, and characters.

**Relevance to memory management**

The objects of a scalar data type are *leaf objects*. Scalar data types with bounded magnitude can be represented compactly using *value objects*.

**Historical note**
Because compact representation solves many memory management issues, many older programming languages only offered bounded scalar data types. For example, the \texttt{int} type in \textit{C} is defined to have a magnitude that can be represented by a \textit{word}.

See also:
vector data type, algebraic data type, value object, leaf object.

\textbf{scan} The examination of an \textit{object} or an area of \textit{memory} (2) to find \textit{references}, typically as part of \textit{tracing}.

Scanning examines memory that has been decided to be non-\textit{garbage}, to find references to objects that have been \textit{condemned}.

\textbf{In the MPS}
See \textit{Scanning}.

\textbf{scan method}

\textbf{In the MPS}
A function that examines a block of memory to find \textit{references} and indicate them to the MPS. A scan method forms part of an \textit{object format}. See \textit{mps_fmt_scan_t}.

\textbf{scan state}

\textbf{In the MPS}
A scan state represents the state of the current \textit{scan}. The MPS passes a scan state to the \textit{scan method} of an \textit{object format} when it needs to \textit{scan} for \textit{references} within a region of memory. Scan states belong to the type \textit{mps_ss_t}.

\textbf{scavenging garbage collection}

See
\textit{copying garbage collection}.

\textbf{SDRAM} Synchronous Dynamic Random Access Memory. A high performance variant of \textit{DRAM}.

SDRAM uses an external clock signal to synchronize its data input and output. It is capable of achieving very high data rates for linear access to memory.

\textbf{segmentation violation} A segmentation violation occurs when an attempt is made to access \textit{memory} (2) whose \textit{address} is well-formed, but to which access cannot be granted. This might be due to either a \textit{protection fault} or an \textit{invalid page fault}.

The term is sometimes used more loosely as a synonym for any memory access error, including a \textit{bus error}.

\textbf{Similar terms} general protection fault, read fault, write fault.

\textbf{segmented addressing} In segmented addressing, \textit{addresses} are in two parts: a segment identifier and an offset into that segment.

Each segment has a base address and a limit. If the offset is greater than the limit, the address is invalid (see \textit{segmentation violation}). Otherwise, the offset is added to the segment’s base address, giving the unsegmented address. Segment identifiers may be implicit; for instance, they may be obtained from a current segment register.
Segmentation may be layered on top of virtual memory, in which case the unsegmented address is a virtual address, or not, in which case it is a physical address.

Note that, in segmented architectures, you can have a two-dimensional address space.

Segments are a feature of some processor architectures and operating systems. This description does not cover all possible variations on segmentation.

**Historical note**

Segment terminology may be used on unsegmented systems for historical reasons. For instance, Unix processes have text segments, even when running on an unsegmented system.

**Opposite term**

linear addressing.

**segregated allocation cache**

In the MPS

A mechanism for adding a segregated free list to a manual pool class. See Segregated allocation caches.

**segregated fit**

One of the segregated free list class of allocation mechanisms. There is an array of free lists, each holding free blocks of a particular range of sizes. The allocator identifies the appropriate free list and allocates from it (often using a sequential fit mechanism such as first fit). If this fails, a larger block is taken from another list and split.

The details of the mechanism depend on the division of sizes between free lists. See exact segregated fit and strict segregated fit.

This implements a good fit allocation policy.

See also:

segregated free list, allocation mechanism, free list, exact segregated fit, strict segregated fit.

**Related publication**


**segregated free list, segregated free-list**

A class of allocation mechanism which divides the free list into several subsets, according to the size of the free blocks. A freed or coalesced block is placed on the appropriate list. An allocation request is serviced from the appropriate list.

This class of mechanism implements a good fit or best fit policy.

Variations within this class include simple segregated storage, segregated fit, and buddy systems.

**Related publication**


**In the MPS**

Segregated allocation caches are a general mechanism for adding a segregated free list to any manually managed pool. See Segregated allocation caches.
semi-conservative garbage collection

Also known as

mostly-precise garbage collection, mostly-exact garbage collection.

A variant of conservative garbage collection which deals with exact references as well as ambiguous references.

For example, references from the root set might be ambiguous, but objects on the heap might be fully described and precisely scanned.

See also:

mostly-copying garbage collection.

Related publication


semi-space

When an area of memory is divided into two parts for the purposes of copying garbage collection, the parts are known as semi-spaces, or sometimes just spaces.

Each semi-space is a contiguous area of memory. Semi-spaces are usually used for two space collection, but can be used for generational collection.

The semi-space where objects reside at the start of the collection is known as the fromspace; the tospace is where objects will reside, and where new objects will be allocated, when the collection is complete.

See also:

two-space collector.

semi-space collector

See

two-space collector.

sequential fit

A class of allocation mechanisms that maintain the free list as a single linear list of free blocks (a free block chain). Sequential fit mechanisms include first fit and next fit.

To quote Wilson et al. (1995):

The list is often doubly-linked and/or circularly linked. Typically, sequential fit algorithms use Knuth’s boundary tag technique, and a doubly-linked list to make coalescing simple and fast. […] In considering sequential fits, it is probably most important to keep strategy and policy issues in mind. The classic linear-list implementations may not scale well to large heaps, in terms of time costs; as the number of free blocks grows the time to search the list may become unacceptable. More efficient and scalable techniques are available, using totally or partially ordered trees, or segregated fits.

See also:

bitmapped fit, indexed fit.

sequential store buffer

Also known as

SSB.

A sequential store buffer is a technique for dividing the cost of a write barrier by remembering which objects are modified and updating remembered sets (and so on) at a later stage.
This turns out to be extremely efficient on pipelined architectures with branch prediction.

**shared memory** Memory locations are *shared* if they are in the range of multiple *address spaces*.

**simple object** In the *PostScript* language, *simple objects* are the *unboxed* objects.

Unlike a *composite object*, a simple object contains all its data in the object itself.

** Similar term**

*unboxed.*

**Opposite term**

*composite object.*

**simple segregated storage** A *segregated free list allocation mechanism* which divides memory into *pages* or other areas and only allocates objects of a single size, or small range of sizes, within each area. This makes allocation fast and avoids *headers*, but may lead to high *external fragmentation*, as unused parts of areas cannot be reused for other object sizes.

**Related publication**


**size**

In the MPS

The term *size* in the documentation always refers to a size that is measured in *bytes*. The term *count* is used for the number of elements in an array.

**size class**

In the MPS

A *segregated allocation cache* maintains a reserve of free *blocks* in a set of *sizes*: each such size is known as a *size class*. When creating a segregated allocation cache by calling `mps_sac_create()`, the *client program* describes the desired set of size classes by passing an array of structures of type `mps_sac_class_s`. See *Segregated allocation caches*.

**skip method**

In the MPS

A *format method* that returns the address of the “next object” in a block of *formatted objects*. See `mps_fmt_skip_t`.

**smart pointer** A smart pointer is an instance of a *C++* class that encapsulates a *pointer* and performs *reference counting*.

By overloading certain operators it is possible for the class to present the illusion of being a pointer, so that `operator*`, `operator->`, etc. can be used as normal. Reference counting allows the objects that are referred to using the smart pointer class to have their memory automatically *reclaimed* when they are no longer *referenced*. It is a common technique used when trying to solve *memory management* problems in C++ applications.
However, reference counting is not always an appropriate memory management technique and smart pointers can be hard to implement properly in C++. A *tracing garbage collector* might be worth considering.

**Related publication**

*Edelson (1992a).*

**snap-out**

*Also known as*

*transport snap-out.*

In a *copying collector*, when there is a *reference* to an *object* that was *condemned*, but has been *transported*, snap-out is the adjustment of that reference to point to the preserved copy.

Typically the first transport leaves a *forwarding pointer* that enables the snap-out.

![Fig. 6.10: Snap-out.](image)

**See also:**

*broken heart.*

**snapshot at the beginning**   
Snapshot-at-the-beginning algorithms for *tracing, incremental GC* note changes made by the *mutator* to the *graph of objects* and update the *collector*-*state* to make it trace relevant *edges* that the mutator deletes.

In order for the collector to miss a *reachable object*, the following two conditions need to hold at some point during tracing:

1. The mutator stores a *reference* to a *white* object into a *black* object.
2. All paths from any *gray* objects to that white object are destroyed.

Snapshot-at-the-beginning algorithms ensure the second condition cannot occur, by causing the collector to process any reference that the mutator overwrites and that might be part of such a path.

They are so called because they keep track of references that existed at the beginning of the *collection cycle*. Note that this does not mean all modifications need to be seen by the collector, only those needed to complete tracing without missing a reachable object (*see Pirinen (1998)* for details), nor does it mean that it won’t trace some references created during the collection.

**Historical note**

This distinction between incremental update and snapshot at the beginning was first introduced for write-barrier algorithms, but it applies to any type of tracing algorithm.

**Opposite term**

*incremental update.*

**See also:**

*tri-color marking, weak tri-color invariant, barrier (1).*

**Related publications**

**soft reference**  In Java terminology, *soft reference* is used to mean a reference encapsulated in a *reference object* of class `SoftReference`.

Soft references form one of three kinds of *weak reference* in Java. They are handy for building *caches* that are automatically flushed when memory is low.

See also:

*softly reachable*.

Related links

Class java.lang.ref.SoftReference, Reference Objects and Garbage Collection.

**softly reachable**  In Java, an object is *softly reachable* if it is not *strongly reachable* and there is a path from the *roots* to it that contains at least one *soft reference* but no *weak* or *phantom references*.

When the Java *collector* determines that an object is softly reachable, it has the option of clearing the soft references involved, which will usually allow the object to be *recycled*. The idea is that they will only be cleared if the process is running short of *memory*. If it is done, all soft references involved are cleared, so that the object is no longer softly reachable, and any affected *reference objects* which are registered with a queue are enqueued.

See also:

reachability, weakly reachable, phantom reachable.

Related link

Class java.lang.ref.SoftReference, Reference Objects and Garbage Collection.

**space leak**

See

memory leak.

**spare commit limit**

In the MPS

The spare commit limit is a limit on the *spare committed memory* that the MPS will obtain from the operating system. It can be retrieved by calling `mps_arena_spare_commit_limit()` and changed by calling `mps_arena_spare_commit_limit_set()`.

**spare committed memory**

In the MPS

Memory which is not in use by any *pool* and not otherwise in use for internal MPS data structures, but which remains committed (mapped to RAM by the operating system). It is used by the *arena* to (attempt to) avoid calling the operating system to repeatedly map and unmap areas of *virtual memory* as the amount of memory in use goes up and down. It is subject to the *spare commit limit*. The total spare committed memory can be retrieved by calling `mps_arena_spare_committed()`.

**spaghetti stack**
Splat

In the MPS
To overwrite a weak reference (1) with a null pointer, when the MPS has determined that there are no remaining strong references to the block referred to. See Weak references.

Split
To divide a free block into two smaller free blocks in the process of satisfying an allocation request.
Deciding when to split a block is an important aspect of an allocation policy.

Opposite term
coalesce.

See also:
coalesce, allocation policy, free block.

SRAM

See
static memory (1).

SSB

See
sequential store buffer.

Stack
A stack is a LIFO (last in, first out) collection: objects may be pushed onto the stack, and popped off it in reverse order of pushing.

When people say “the stack”, they usually mean the control stack supported by the OS and/or the processor.

Relevance to memory management
Stack allocation is an important technique. Control stacks are central to the performance of the system and often require special handling.

Historical note
The terms “stack”, “push”, and “pop” are taken from the spring-loaded dish stack found in cafeterias and salad bars where removing the top plate causes the others to rise up, exposing the next one, and adding a plate causes the spring to compress, leaving only that plate accessible.

So originally, the latest item was the “top”, “down the stack” meant towards earlier items, and “up” towards later ones, but today many use “up” and “down” in the opposite sense.

Similar term
control stack.
stack allocation  Stack allocation means run-time allocation and deallocation of memory in last-in/first-out order.

Typically, stack allocation is performed on top of the main stack, but one can have a separate data stack for this purpose as well, as in Forth, or even multiple ones, as in the PostScript language.

Allocation and deallocation are typically fast, since they can be done simply by adding or subtracting the size of the block from the stack pointer.

Using only stack allocation, without heap allocation, is somewhat restrictive, as only objects whose size is known at compile-time can be returned from a procedure.

Some programming languages (such as some versions of Lisp and C) provide program-controlled stack allocation and deallocation of dynamic extent objects for efficiency, despite its being unsafe.

**Similar term**

automatic storage duration.

**Opposite terms**

heap allocation, static allocation.

**See also:**

region inference, dynamic extent.

stack frame

Also known as

stack record.

A stack frame or record is an activation record that is stored on the stack.

In a register-based architecture, where the current activation record may be partially stored in registers, there may be hardware instructions that facilitate storing registers on the stack when another activation record is made current. Such instructions may prescribe a particular layout for activation records.

**Relevance to memory management**

Hardware support for saving and restoring registers, for stacks and for stack addressing may limit or otherwise prescribe the size and type of data that can be stored in a stack frame. Knowledge of the layout of each stack frame may assist a garbage collector in finding roots.

**Similar term**

activation record.

**See also:**

stack.

stack record

See

stack frame.
**static allocation**  *Static allocation* means *allocation of memory (1)* before the program starts and retention until the end.

The locations of *objects* are basically decided at compile-time, although they might be *relocated* at load-time. This implies the sizes of the objects must be known then.

Using only static allocation is restrictive, as sizes of data structures can’t be dynamically varied, and procedures cannot be recursive. However, it is also fast and eliminates the possibility of running out of memory. For this reason, this scheme is sometimes used in real-time systems.

**Similar term**

*static storage duration.*

**Opposite terms**

*stack allocation, heap allocation.*

**See also:**

*region inference, static memory (2).*

**static memory(1)**

Also known as

*static RAM, SRAM.*

Static *memory (2)* or static RAM (SRAM) is a type of *physical memory (2)* that does not need to be refreshed periodically to avoid losing state.

Static memory is typically faster than *dynamic memory*, or requires essentially no power to preserve its state, but rarely both. These benefits result in static RAM being used for *cache (1)* memory, and also in portable, low-power applications (such as PDAs). It is, however, more expensive than dynamic RAM and requires more transistors, making dynamic RAM the choice for large amounts of memory (the *main memory* of desktop machines, for example).

**Opposite term**

*dynamic memory.*

**static memory(2)**  The *memory (2)* where *statically allocated* objects are stored is sometimes known as *static memory*. In the context of *garbage collection*, the term is used mean memory used to store *static objects*.

**See also:**

*static storage duration.*

**static object**  A static *object* is non-moving. That is, it is not *relocated* by a *memory manager*; its *address* does not change.

**static RAM**

**See**

*static memory (1).*
**static storage duration**  In C and C++, the `static` keyword applied to a file scope variable or function means it is local to the file; the `static` keyword applied to a function or a block scope variable means it is *allocated* and initialized once only.

Objects declared locally in blocks with the `static` keyword are *allocated in static memory* (2), and initialized once (usually by the compiler/linker) instead of each time the block is entered.

Static variables within functions retain their value between function invocations, and therefore must form part of the root set of any collector (1).

Opposite term

*automatic storage duration.*

See also:

*lifetime.*

stepper function

Also known as

*visitor function.*

In the MPS

A function that will be called on each element in a collection. For example, a stepper function of type `mps_formatted_objects_stepper_t` can be passed to `mps_arena_formatted_objects_walk()` and it will be called on all formatted objects in an arena.

sticky reference count

See

*limited-field reference count.*

stop-and-copy collection  *Copying garbage collection* that stops the mutator while the collector runs.

Fig. 6.11: Stop-and-copy in a two-space collector.

Opposite terms

*incremental garbage collection, parallel garbage collection.*

storage

See

*memory (1).*

storage hierarchy

Also known as

*memory hierarchy.*
A typical computer has several different levels of storage. Each level of storage has a different speed, cost, and size. The levels form a storage hierarchy, in which the topmost levels (those nearest the processor) are fastest, most expensive and smallest.

Levels typically include processor registers, possibly some levels of cache (1), main memory, and possibly some levels of backing store.

Each level is commonly used as a cache (2) for the next level. For instance, virtual memory systems use main memory as a cache for backing store.

**Fig. 6.12: Storage hierarchy with (typical) relative cost, speed, and size.**

**storage level** One level in a storage hierarchy, for instance a cache (1), main memory, backing store, and so on.

See also:

storage hierarchy.

**storage management**

See memory management.

**store** To transfer data from a processor’s registers to memory (2).

Store can also be used in the more general sense of transferring data from a part of the memory hierarchy that is fast to access to one that is slow to access.

STORE (or an abbreviation) is also commonly used in many processor architectures as the mnemonic for the machine code instructions that store data into memory.

**Opposite term**

load.

**store**

See memory (1).

**stretchy vector** A vector that may grow or shrink to accommodate adding or removing elements. Named after the <stretchy-vector> abstract class in Dylan.

**Relevance to memory management**

In the presence of an asynchronous garbage collector, the vector and its size may need to be updated atomically.

**Related link**


**In the MPS**

See The stretchy vector problem.
strict segregated fit  A segregated fit allocation mechanism which has only one block size on each free list. A requested block size is rounded up to the next provided size, and the first block on that list is returned. The sizes must be chosen so that any block of a larger size can be split into a number of smaller sized blocks. Buddy systems are a special case of strict segregated fit allocators.

See also:

buddy system, segregated fit, segregated free list, allocation mechanism.

Related publication


strong reference  In a tracing garbage collector, a strong reference is a reference that keeps the object it refers to alive.

A strong reference is the usual sort of reference: the term is usually used to draw a contrast with weak reference.

Opposite term

weak reference (1).

See also:

strong root.

strong root  A strong root is a root such that all references in it are strong references.

A strong root is the usual sort of root: the term is usually used to draw a contrast with weak root.

Opposite term

weak root.

In the MPS

Strong roots have rank mps_rank_ambig() or mps_rank_exact().

strong tri-color invariant, strong tri-colour invariant, strong tricolor invariant, strong tricolour invariant  The strong tri-color invariant is the property of a reference graph that there is no edge from a black node to a white node.

By preserving this property throughout tri-color marking, a tracing algorithm can ensure that the collector will not miss reachable objects, even if the mutator manipulates the graph during the collection. This invariant can also be used to ensure that a copying garbage collector doesn’t confuse the mutator. Mutator actions might need to change the color of the nodes affected in order to preserve the invariant.

Algorithms using this invariant are incremental update algorithms.

Similar term

tri-color invariant.

See also:

barrier (1), weak tri-color invariant.

Related publications
strongly reachable  In Java, an object is strongly reachable, if there is a path from the roots to it that contains only strong references, that is, contains no reference objects.

See also: reachability, softly reachable, weakly reachable, phantom reachable.

Related link
Reference Objects and Garbage Collection.

suballocator  A suballocator is an allocator functioning on top of another allocator.

Application programmers sometimes write their own suballocators when faced with an inefficient or inadequate memory manager. Suballocators can take advantage of special knowledge of program behavior, but are less efficient in general than fixing the underlying allocator, mainly because memory management is a global issue for an application, and a global strategy can make a big difference. For example, different suballocators can interact catastrophically with each other and with the virtual memory system, causing the application’s memory requirements to grow unnecessarily due to fragmentation.

subgraph  A subgraph S of a graph G is a graph such that all the nodes in S are also in G and all the edges in S are also in G; that is, it is a part of a graph.

superpage

See huge page.

sure reference

See exact reference.

swap space  Backing store used by a swapping system.

See also: swapping, backing store.

Swapped in  A process or page is swapped in if it is available in physical memory (1). This usually applies to the entire program image.

Swapped out  See also: swapping.
swapped out  A process or page is *swapped out* if it is not available in *physical memory* \(^1\). This usually applies to the entire program image.

Similar term

*paged out.*

Opposite term

*swapped in.*

See also:

*swapping.*

swapping  Historically, swapping was the technique of moving entire program images to disk (or drum) and back into *physical memory* \(^1\), an early form of *virtual memory*. Nowadays, it is used as a synonym for *paging*.

Similar term

*paging.*

See also:

*swapped in, swapped out.*

sweeping  Sweeping is the second phase ("the sweep phase") of the *mark-sweep* algorithm. It performs a sequential (address-order) pass over memory to *recycle* unmarked blocks.

Sweeping typically gathers all unmarked blocks into one or more *free lists*.

See also:

*marking.*

synchronous garbage collector  A *collector* \(^2\) is asynchronous with respect to the *mutator* if it runs at predictable times, for example only when a collection function is called.

This means that mutator need not ensure that *formatted objects* are always *scannable*, as long as it makes them scannable before the collector runs.

Opposite term

*asynchronous garbage collector.*

6.2.19 Memory Management Glossary: T

A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z

tabling

See

caching (3).

tag  A tag is a piece of information associated with an *object* or *reference* that allows the representation of the object to be determined.
Tags are often used to represent types in the implementation of a dynamically-typed language. In statically-typed languages, types are usually implicit and not permitted to change at run-time, so tagging is rarely required. One of the simplest forms of tag is a *word* at the beginning of the object that points to a block of information about the object’s *format*.

Fig. 6.13: Example of a tag-word at the start of an object.

Another common form of tagging is to *align* objects and keep information in the least significant bits of the reference.

Fig. 6.14: Example of reference tagging, with objects aligned to addresses that are multiples of four, and the tag stored in the least significant two bits of the reference.

In C, when a structure contains a union, it is common to add a field to the structure to indicate which union member is currently being used. This field is known as a *discriminator*, and is a form of tag. Analogues occur in other languages, sometimes with compiler or run-time support.

**See also:**
tagged architecture, in-band header.

**Related publication**

*Gudeman (1993).*

**In the MPS**

See *Tagged references*.

**tagged architecture** A tagged architecture is a hardware architecture where each memory *word* is divided into a “data” and a *tag* section. The data section is sufficiently large to contain a memory *address* and the tag section is used to describe how the data section is to be interpreted (that is, it encodes the type of the data).

**Relevance to memory management**

Tagged architectures greatly simplify the implementation of a memory manager because each word of memory is self-describing.

**Historical note**

The *Lisp Machine* was an example of a tagged architecture.

**tagged reference** A *reference* containing a *tag* in part of its address, for example by *aligning* objects and keeping the tag in the least significant bits of the address.

**In the MPS**

See *Tagged references*.

**TB**

See *terabyte*. 

---

324  Chapter 6. Appendices
TB\(^{(2)}\)  
See  
translation lookaside buffer.

telemetry filter  
In the MPS  
A bitmap indicating which events the MPS should include in the telemetry stream. It can be read or changed by calling mps_telemetry_control().

telemetry label  
In the MPS  
An identifier representing a string, returned from mps_telemetry_intern(), that can be associated with certain addresses, and so appear in the telemetry stream attached to events concerning those addresses. See Telemetry.

telemetry stream  
In the MPS  
A sequence of events reported by the MPS to assist with debugging and profiling. The events that appear in the stream can be configured by setting the telemetry filter. See Telemetry.

tenuring  
See  
promotion.

terabyte  
Also known as  
TB.  
A terabyte is 1024 gigabytes, or 1099511627776 bytes\(^{(1)}\).  
See byte\(^{(1)}\) for general information on this and related quantities.

termination  
See  
finalization.

thrash A cache\(^{(2)}\) is said to thrash when its miss rate is too high, and it spends most of its time servicing misses. Thrashing is bad for performance, particularly virtual memory thrashing, because the relative cost of a miss is so high: it may slow a machine down by a factor of a hundred or more.

Thrashing is typically caused by a process or system having a working set which is larger than its cache\(^{(1)}\) or main memory. It may also be caused by a failure of cache policy. A system with an inflexible cache policy may thrash even when the working set is quite small.
For instance, a virtual memory system which has four megabytes of *physical memory* but which has a working set of ten megabytes will *thrash* badly.

**Related publications**

*Denning (1968), Denning (1970), Denning & Schwartz (1972).*

**thread** A thread of execution is a sequence of instructions that take place sequentially. In a multi-threaded program, multiple threads of execution operate in parallel, and are generally asynchronous with respect to each other.

**Relevance to memory management**

Access to shared resources such as memory management interface must be thread-safe. Each thread has its own *control stack* which may contain *references* to blocks on the heap.

**In the MPS**

Threads are represented by values of type `mps_thr_t`, created by calling `mps_thread_reg()`. In order for the MPS to find references on the control of the thread, the thread must be also be registered as a root by calling `mps_root_create_reg()`. See *Threads*.

**threatened set**

See

*condemned set.*

**TLB**

See

*translation lookaside buffer.*

**to space, tospace**

Also known as

*new space, newspace.*

In *copying garbage collection*, the space to which *live* object are copied.

**Opposite term**

*fromspace.*

**trace** In *tracing garbage collection*, tracing is the process of following the *graph* from all *roots* to all *reachable* data.

**Similar term**

*scan.*

**tracing garbage collection** Tracing garbage collection is *garbage collection* based on *reachability*.

Tracing garbage collection relies on the fact that if an *object* is not *reachable*, there is no way the *mutator* could ever access it, and therefore it cannot be *live*. In each *collection cycle*, some or all of the objects are *condemned* and the *graph* is *traced* to find which of the condemned objects are reachable. Those that were not reachable may be *reclaimed*. 
**translation buffer, translation lookaside buffer**

Also known as

address translation cache, ATC, TB.

The **translation lookaside buffer** or **address translation cache** is small piece of associative memory within a processor which caches part of the translation from virtual addresses to physical addresses.

In a virtual memory system there is a translation from virtual addresses to physical addresses. This translation can often be very large and complex and the data structures that implement the translation (often a page table) can be too large to store efficiently on the processor. Instead, a few elements of the translation are stored in the TLB; the processor can access the TLB extremely quickly. If a required translation for a particular virtual address is not present in the TLB then a **TLB miss** is taken and the address is resolved using the more general mechanism.

**transparent alias, transparent type**

In the MPS

In the MPS interface, a **transparent type** is an alias defined using typedef, and this is documented so that the client program can rely on that fact. For example, mps_addr_t is a transparent alias for void *. See Interface conventions.

**Opposite terms**

derived type, opaque type.

**transport** In a copying collector, transporting is preventing an object in the condemned set from being collected by copying it and adjusting the reference by which it was discovered to point to the new copy.

See also:

scavenging, snap-out.

**transport snap-out**

See

snap-out.

**treadmill** Henry Baker devised an incremental non-moving garbage collector that uses a circular doubly-linked list, called the treadmill, to implement tri-color marking.

Every object is on the list. The list has four sections corresponding to colors. The black, gray and white sections are used for tri-color marking, and an additional off-white section is used for free (3) objects. The color of an object is changed by unlinking it from the list and relinking it to a different part of the list.

Fig. 6.15: A treadmill. (Based on Jones (2012).)

**Related publication**


**tri-color invariant, tri-colour invariant, tricolor invariant, tricolour invariant** The term “tri-color invariant” is used to refer to any of a number of properties of a reference graph that are preserved throughout a tri-color marking algorithm to ensure the correctness.
There are two important ones: the strong tri-color invariant and the weak tri-color invariant. When people say “the tri-color invariant” they probably mean the strong one.

Related publications


Tri-color marking is a tracing garbage collection algorithm that assigns a color (black, white, or gray) to each node in the graph. It is basic to incremental garbage collection.

Initially all nodes are colored white. The distinguished root set is colored gray. The collector proceeds to discover the reachable nodes by finding an edge from a gray node to a white node and coloring the white node gray. Hence each tracing step involves choosing a gray node and graying its white children.

When all the edges from a gray node lead only to other gray (or black) nodes, the node is colored black. When no gray nodes remain, the reachable part of the graph has been discovered and any nodes that are still white may be recycled.

The mutator is free to access any part of the graph and allocate new nodes while the collector is determining the reachable nodes, provided the tri-color invariant is maintained, by changing the colors of the nodes affected, if necessary.

Historical note

“Tri-color marking” is the term used to describe an algorithm developed in 1975 by E. W. Dijkstra and others, as an exercise in proving cooperating programs correct. They chose as their problem a parallel garbage collector, with the intent of illustrating cooperating sequential processes with a large shared data space but minimal exclusion and synchronization constraints.

Although the algorithm developed in the paper is not necessarily the most efficient algorithm for a collector, it has been generally accepted to be correct: an important feature that not all garbage collectors can claim. A number of other garbage collection algorithms have been shown to be isomorphic to the tri-color marking algorithm and thus are also believed to be correct.

See also:

barrier.

Related publication

Dijkstra et al. (1976).

two-space collector, two space collector

Also known as

semi-space collector.

A two-space collector is a simple form of a copying garbage collector. The available memory is divided into two halves, called semi-spaces. Objects are allocated in one semi-space until it is full. The reachable objects are then copied into the other semi-space (usually using a Cheney scan) and the old semi-space is reclaimed. Allocation continues in the new semi-space until it is full, at which point the process is repeated in reverse.

The main disadvantage of a two-space collector is that it only makes use of half of the available memory. This can be tolerable in a virtual memory system if the garbage collector is written carefully to preserve locality of reference. Other forms of copying garbage collector, such as generational garbage collectors, have much lower overheads.
Fig. 6.16: Two-space collector.

See also:
flip.

type-accurate garbage collection
See
exact garbage collection.

type punning  Interpreting a value of one type as if it were a value of another (for example, via a type cast in C), especially if such interpretation is not defined by the language standard. For example, interpreting a value of type T** (pointer to pointer to T) as U** is undefined.

In the MPS
See Interface conventions.

6.2.20 Memory Management Glossary: U

unaligned

Also known as
misaligned.

An address is unaligned or misaligned if it does not comply with some alignment constraint on it.

For example, typically double precision floating point numbers occupy 8 byte (1) and have an alignment of 4 bytes; that is, their address must be a multiple of four. If a program tries to access such a number using an address that is not a multiple of four, a bus error may result, depending on the processor architecture and instruction used.

Opposite term
aligned.

See also:
alignment, bus error.

unboxed  Unboxed objects are represented by an encoding of the data itself, and not by a pointer to that data.

Representations are typically chosen so that unboxed values are the same size as the pointer part of a boxed object. Sometimes the value is tagged to distinguish it from a boxed object. The entire object is duplicated when the object is passed around, so updates to it, if allowed, only affect one copy.

Similar term
immediate data.

Opposite term
boxed.

Related publication

unclamped state

In the MPS
One of the three states an arena can be in (the others being the clamped state and the parked state). In the unclamped state, object motion and other background activity may occur. Call mps_arena_release() to put an arena into the unclamped state.

undead  An undead object is an object that cannot be proven to be dead by the garbage collector, but whose liveness is dubious.

For example, an ambiguous reference to an object on a page may mark the entire page as reachable. No further data is collected about that page. The other objects on the page will survive, even though their reachability has not been determined. They are undead.

unmapped

Also known as
free.

A range of virtual addresses is said to be unmapped (free on Windows) if there is no physical memory associated with the range.

An unmapped range may or may not be reserved.

Opposite term
mapped.

unreachable  An object is unreachable if there is no reference chain to it from any root.

An object will become unreachable when the mutator overwrites its last (direct or indirect) reference to the object.

Similar term
dead.

Opposite terms
reachable, live.

See also:
reachable, garbage collection.

unsure reference

See
ambiguous reference.
unwrapped

Also known as
raw.

A value is unwrapped or raw if it is not encoded with type information.

In a dynamically-typed language, the compiler may sometimes be able to pick a more compact or efficient representation for a value if it can prove that the type can be determined at compile-time. This is a particularly useful optimization for numeric values such as integers or floats.

Opposite term
wrapped.

See also:
boxed, tag, value object.

Related publication

use after free

See
premature free.

6.2.21 Memory Management Glossary: V

value object

Also known as
immutable object.

A value object or immutable object is an object whose identity depends solely upon its value or magnitude.

In a typed language, the compiler can often determine at compile time that certain types can be represented as value objects. Usually these types are a scalar data type with bounded magnitude.

Relevance to memory management

If value objects can be identified, the compiler and the memory manager can make certain optimizations: Value objects can be represented as immediate data to minimize storage overhead, they can be replicated to improve locality, and a vector data type of value objects can be represented as a leaf object.

Historical note

Some programming languages expose representational details such as the use of value objects. In Lisp, for example, numbers are often represented as value objects but not always as immediate data. The EQ predicate of Lisp tests if two objects have the same representation, whereas the EQL predicate tests if two objects represent
the same type and value (are computationally identical). Because the choice of representation is an optimization, exposing it at the language level can cause programs to behave differently under different compilers or optimization settings. Modern languages, such as Dylan hide this representational distinction, permitting the compiler greater freedom in optimization.

**Similar term**

*immediate data.*

**See also:**

*immutable.*

**Related publication**

*Baker (1993a).*

---

**variety**

**In the MPS**

A behaviour of the MPS that must be selected at compilation time. There are three varieties: *cool, hot* and *rash.* See *Building the Memory Pool System.*

**vector data type** A vector data type is an aggregate type of more than one dimension whose objects have a value for each dimension, where each dimension is of the same type.

Examples of vector data types include: strings, arrays, and lists.

**Relevance to memory management**

Vector data types are seldom represented using *value objects,* but may be represented using *leaf objects* if they are an aggregate of a type that can be represented by *value objects.* Scanning information for vectors can be compactly encoded in terms of the aggregated type and the vector dimension.

**See also:**

*scalar data type, algebraic data type, value object, leaf object.*

---

**virtual address**

**Also known as**

*logical address.*

In a *virtual memory* system, the *addresses* that application programs deal with are known as *virtual addresses.*

The virtual addresses used by the application program are translated by the virtual memory system (often using *translation lookaside buffers* and *page tables*) to *physical addresses.* It is the physical address that is used to retrieve the contents from the memory (3).

**Opposite term**

*physical address.*

---

**virtual address space** The virtual *address space* is the space of *virtual addresses.*
On virtual memory systems, user processes see the virtual address space, and commonly have a separate virtual address space each, so that they map the same addresses to different data. These systems often have shared memory as well.

Opposite term

physical address space.

virtual memory

Also known as

VM.

In a virtual memory (VM) system, the program code deals with virtual addresses. Upon use, the virtual address is translated by the MMU to obtain a physical address that is used to access physical memory (1).

Some operating systems can simulate having more memory (2) than is available as main memory, by storing part of the data in backing store, typically on disk. If the page referenced by the virtual address is not currently in main memory, a page fault occurs, triggering an operating system handler that swaps in the page. Some other page might be swapped out to make room.

Each process typically has its own separate virtual address space with its own mappings and protections.

Fig. 6.17: Example of the relationship between the virtual address spaces of two processes, physical memory, and backing store.

Virtual memory technology can be used in many useful memory management techniques, such as barriers (1), copy-on-write, and memory mapping.

“Virtual” means never knowing where your next byte is coming from. — fortune(6)

Opposite term

real memory (1).

See also:

paging, paged in, paged out, swapping, swap space, mapped, reserved, unmapped, shared memory.

virtual memory arena

In the MPS

An arena class which gets its memory (2) from the operating system’s virtual memory interface. See Virtual memory arenas.

visitor function

See

stepper function.

VM (1)

See

virtual memory.
VM In the *PostScript* language, VM is the *memory* where the values of the *composite objects* reside.

VM is short for “virtual memory”, but this has nothing to do with the usual sense of the phrase (see *virtual memory*).

### 6.2.22 Memory Management Glossary: W

**A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z**

#### weak-key hash table
A hash table which has *weak references* to its keys. If the key dies, the value for that key is automatically deleted from the table too. It can be used to store extra information about objects without keeping them alive.

**Similar terms**

doubly weak hash table, weak-value hash table.

**In the MPS**
See AWL (*Automatic Weak Linked*).

#### weak-value hash table
A hash table which has *weak references* to its value. If the value dies, any keys that refer to that value are automatically deleted from the table too. It can be used to index a set of objects without keeping them alive.

**Similar terms**

doubly weak hash table, weak-key hash table.

**In the MPS**
See AWL (*Automatic Weak Linked*).

#### weak hash table
A *weak-key* or *weak-value hash table* (usually the former).

#### weak reference
In *tracing garbage collection*, a weak reference is a *reference* that does not keep the *object* it refers to *alive*.

A weak reference does not keep the referent alive, but it will continue to refer to the object as long as it remains otherwise alive. When only weak references to the object remain, the weak references can be deleted (“splatted” or “cleared”) and the object *reclaimed*.

*Java* offers three kinds of weak references, called *soft references*, weak references (2), and *phantom references*, in order of increasing weakness.

**Opposite term**

*strong reference*.

**See also:**

weak root.

#### weak reference
In *Java* terminology, *weak reference* is used to mean a *reference* encapsulated in a *reference object* of class *WeakReference*.
Weak references form one of three kinds of weak reference (1) in Java. They are handy for associating extra data with objects when you cannot store it in the objects themselves.

See also:
weakly reachable.

Related links
Class java.lang.ref.WeakReference, Reference Objects and Garbage Collection.

weak root  A weak root is a root, such that all references in it are weak references (1); that is, they do not affect the liveness of the objects referred to.

Opposite term
strong root.

In the MPS
A weak root has rank mps_rank_weak ().

weak tri-color invariant, weak tri-colour invariant, weak tricolor invariant, weak tricolour invariant  The weak tri-color invariant is the property of a reference graph that all white nodes pointed to by a black node are also reachable from some gray node through a chain of white nodes.

By preserving this property throughout tri-color marking, a tracing algorithm can ensure that the collector (2) will not miss reachable objects, even if the mutator manipulates the graph during the collection. Mutator actions might need to change the color of the nodes affected in order to preserve the invariant.

Algorithms using this invariant are snapshot-at-the-beginning algorithms.

See also:
barrier (1), strong tri-color invariant.

Related publications

weakly reachable  In Java, an object is weakly reachable if it is neither strongly nor softly reachable and there is a path from the roots to it that contains at least one weak reference (2) but no phantom references.

When the Java collector (1) determines that an object is weakly reachable, it clears all the weak references involved, and declares the object finalizable. (Operationally, finalization works as if it was implemented by a class of “final references” that stand between weak and phantom references.) Also, the reference objects containing the weak references are enqueued, if they were registered with a queue.

See also:
reachability, phantom reachable.

Related link
Class java.lang.ref.WeakReference, Reference Objects and Garbage Collection.

weighted buddies  A buddy system allocation mechanism using two series of size classes: binary buddies (2, 4, 8, . . . ) and three-times-power-of-two (3, 6, 12, . . . ). A block that is in the latter series may be split in two different ways. Thus a block of size 12 may be split into two blocks of size 6 or one block of size 4 and one block of size 8. The same applies for coalescing. This gives this system more flexibility than a regular buddy system.
weighted reference counting A technique for reference counting which is in common use for distributed garbage collection because of the low level of inter-process communication it requires.

Inter-process references to objects are counted, but instead of simply counting the number of references, each reference is given a weight, which is usually a power of 2 for easy division. The object records the sum of all the weights of all of its references. Whenever a reference is copied, its weight is divided equally between the new and original copies. Since this operation preserves the weighted reference sum, there is no need for communication with the object at this time. When a reference is deleted, the weighted reference sum is decremented by the weight of the reference. This is communicated to the object by sending it a message. When the object detects that the weighted reference sum has dropped to zero, it may be reclaimed. The algorithm is tolerant of communication protocols which don’t guarantee order of arrival of deletion messages.

white In a tri-color marking scheme, white objects are objects that were condemned at the beginning of the collection cycle and have not been shown to be reachable. When tracing is complete, white objects will be subject to reclamation.

Opposite terms gray, black.

word

Also known as machine word.

Almost all processor architectures have a characteristic data size that is handled most efficiently. This is known as the word size, and data of that size are known as words. The word size is usually a power of two multiple of bytes.

Often the platform’s word size is used to characterize the architecture by quoting the number of bits in it. For example, a 32-bit platform has a word size of four bytes and a 64-bit platform has eight-byte words (assuming 8-bit bytes). Typically, pointers are the size of a word, and traditionally this determined the word size. Nowadays, word size is usually driven by the need for more accuracy and range in mathematical calculations.

Historical note

In the past, the convenience of dealing with powers of two was not as significant, and word sizes such as 36- or 72-bits were not unknown.

See also:

alignment, grain.

working set The working set of a program or system is that memory or set of addresses which it will use in the near future.

This term is generally used when discussing miss rates at some storage level; the time scale of “near future” depends upon the cost of a miss. The working set should fit in the storage level; otherwise the system may thrash.
See also:
resident set, cache (2), storage hierarchy.

Related publication
Denning & Schwartz (1972).

**worst fit**  The allocation policy that always allocates from the largest free block. Commonly implemented using a size-ordered free block chain (largest first).

In practice, this tends to work quite badly because it eliminates all large blocks, so large requests cannot be met.

See also:
allocation policy, first fit, best fit.

Related publication

**wrapped**  A value is wrapped if it is encoded with type information.

Opposite term
unwrapped.

See also:
wrapper, boxed, tag.

Related publication

**wrapper**  A wrapper is that part of a wrapped representation that is copied when the value is passed by value.

The wrapper does not include parts of the representation that are accessed indirectly, and are not copied when the value is passed.

For instance, a Lisp implementation might use the top two bits of a value representation as a tag to distinguish between integers and cons (1) cells, setting these bits to 01 for a pointer to a cons cell and 11 for an integer. Then the wrapped value of the number 4 would have binary representation 11000...00100, and the wrapper for this number is the whole of this wrapped value. The pointer to a cons cell stored at location 4 would have binary representation 01000...00100. The wrapped value of the cons cell is the combination of this pointer and the cons cell in memory itself. The wrapper of the cons cell is just the pointer; when the cons cell is passed as a function argument, just the pointer is passed.

See also:
wrapped, boxed.

Related publication

**write barrier**  A write barrier (1) is a block on writing to certain memory (2) locations by certain threads or processes.

Relevance to memory management
Write barriers are used for incremental or concurrent garbage collection. They are also used to maintain remembered sets for generational collectors (1).

See also:
read barrier.

write fault An exception which occurs when writing to an address in virtual memory. This is probably either a page fault, an invalid page fault or a protection fault.

Similar term
segmentation violation.

See also:
read fault.

6.2.23 Memory Management Glossary: Z

A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z

ZCT

See
go count table.

zero count table

Also known as
ZCT.

A zero count table is used in deferred reference counting to record objects whose reference counts have dropped to zero but which have not been processed to see if they can be reclaimed.

6.2.24 All
6.2. Memory Management Glossary
6.3 Memory Pool System Kit Open Source License

This is the license under which the Memory Pool System Kit is available as open source.

It is not the only way the MPS is licensed.

If the licensing terms aren’t suitable for you (for example, you’re developing a closed-source commercial product or a compiler run-time system) you can easily license the MPS under different terms from Ravenbrook. Please write to us at mailto:mps-questions@ravenbrook.com for more information.

The open source license for the MPS is the Sleepycat License also known as the “Berkeley Database License”. This license is GPL compatible and OSI approved. The MPS is “multi licensed” in a manner similar to MySQL.

6.3.1 License

Copyright © 2001–2013 Ravenbrook Limited. All rights reserved. This is the open source license for the Memory Pool System, but it is not the only one. Contact Ravenbrook at mailto:mps-questions@ravenbrook.com if you would like a different license.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. Redistributions in any form must be accompanied by information on how to obtain complete source code for this software and any accompanying software that uses this software. The source code must either be included in the distribution or be available for no more than the cost of distribution plus a nominal fee, and must be freely redistributable under reasonable conditions. For an executable file, complete source code means the source code
for all modules it contains. It does not include source code for modules or files that typically accompany the major components of the operating system on which the executable file runs.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS “AS IS” AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT, ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDERS AND CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

6.3.2 Exceptions

Open Dylan

Software that makes use of the Memory Pool System only via the Dylan programming language using the Open Dylan implementation and any accompanying software is exempt from clause 3 of the license, provided that the Dylan program is not providing memory management services. For the avoidance of doubt, this exemption does not apply to software directly using a copy of the Memory Pool System received as part of the Open Dylan source code.

6.4 Contact us

- If you have questions about the project, or suggestions for improvement, please write to mps-questions@ravenbrook.com.
- We can often be found on the #memorypoolsystem channel on irc.freenode.net.
- You can also join the mps-discussion mailing list if you are interested in project progress, or if you’re adapting or extending the Memory Pool System. The goals of the list are:
  1. to provide feedback to the project on requirements, design, implementation, etc.;
  2. to allow people to exchange information and experience with using and adapting the project;
  3. to keep people informed about project progress.

To join, follow these instructions, or send a message with the word “subscribe” in the body to mps-discussion-request@ravenbrook.com.

The mailing list is archived and the archives are published on Ravenbrook’s web site.

6.5 Contributing to the MPS

We are very happy to receive contributions to the Memory Pool System so that we can improve it for everyone.

6.5.1 Review

The MPS is highly engineered and rigorously controlled in order to prevent defects. This approach has lead to an extremely small number of bugs in production since its first commercial use in 1997. There are a fairly large number
of rules, both low- and high-level that your code must follow in order to be accepted. These rules are the result of continuous process improvement to prevent defects. Unfortunately, we do not have many of them published at present. We apologise if you find it frustrating that we do not accept your changes as they are.

The style guide in guide.impl.c.format contains basic rules for style.

6.5.2 Licensing

In order for us to accept your contribution, you must agree to the MPS contribution agreement, so that we can continue to commercially license the MPS and thereby fund future development.

I grant Ravenbrook Ltd an irrevocable, perpetual, worldwide, non-exclusive licence to do anything with [your contribution] that I would have the right to do. This includes (but is not limited to):

1. reproducing it and doing any other act that is restricted by copyright;
2. the right to sublicence to others the code and any derivative work.

A member of Ravenbrook staff may ask you to expressly (in writing) agree. You just need to reply with “I agree.” We apologise for the inconvenience.

6.5.3 Thank you

Finally, thank you for making the MPS more useful to everyone.

6.6 Release notes

6.6.1 Release 1.115.0

New features

1. When creating an AMC (Automatic Mostly-Copying) pool, mps_pool_create_k() accepts the new keyword argument MPS_KEY_EXTEND_BY, specifying the minimum size of the memory segments that the pool requests from the arena.

Interface changes

1. The type of pool classes is now mps_pool_class_t. The old name mps_class_t is still available via a typedef, but is deprecated.

2. The functions mps_mv_free_size(), mps_mv_size(), mps_mvff_free_size(), mps_mvff_size(), mps_mvt_free_size() and mps_mvt_size() are now deprecated in favour of the generic functions mps_pool_free_size() and mps_pool_total_size().

Other changes

1. mps_arena_committed() now returns a meaningful value (the amount of memory marked as in use in the page tables) for client arenas. See job001887.

2. AMC (Automatic Mostly-Copying) pools now assert that exact references into the pool are aligned to the pool’s alignment. See job002175.
3. Internal calculation of the address space available to the MPS no longer takes time proportional to the number of times the arena has been extended, speeding up allocation when memory is tight. See job003814.

4. Setting MPS_KEY_SPARE for a MVFF (Manual Variable First Fit) pool now works. See job003870.

5. When the arena is out of memory and cannot be extended without hitting the commit limit, the MPS now returns MPS_RES_COMMIT_LIMIT rather than substituting MPS_RES_RESOURCE. See job003899.

6.6.2 Release 1.114.0

New features

1. Ambiguous interior pointers now keep objects in AMC (Automatic Mostly-Copying) and AMCZ (Automatic Mostly-Copying Zero-rank) pools alive.

   This means that if the compiler optimizes away a pointer to the base of an object, leaving an interior pointer as the only reference keeping the object alive, this does not cause the object to be incorrectly collected. Or, if you are writing your own compiler, you can now perform such an optimization safely.

   If you require the old behaviour (in which ambiguous interior pointers were ignored) then you can set the MPS_KEY_INTERIOR keyword argument to FALSE when calling mps_pool_create_k().

2. The logic for deciding which generations should be collected has changed. Now, a chain may be scheduled for collection if the new size of any of its generations exceeds its capacity, and when a chain is collected, all generations are collected up to, and including, the highest generation whose new size exceeds its capacity. This ensures that all generations are collected reliably on chains where there is no allocation into the nursery generation. See Scheduling of collections.

   (Previously, only the nursery generation in each chain was considered, and a chain was collected up to, but not including, the lowest generation whose new size was within its capacity.)

   As a result of this change, we recommend that you retune your generation sizes. (This is not necessary, but may improve performance.)

3. New pool introspection functions mps_pool_free_size() and mps_pool_total_size().

Interface changes

1. The granularity with which the arena manages memory can now be specified using the MPS_KEY_ARENA_GRAIN_SIZE keyword argument to mps_arena_create_k(). See mps_arena_class_cl() and mps_arena_class_vm().

2. There is now a default value (currently 256 megabytes) for the MPS_KEY_ARENA_SIZE keyword argument to mps_arena_create_k() when creating a virtual memory arena. See mps_arena_class_vm().

3. The keyword argument MPS_KEY_AMS_SUPPORT_AMBIGUOUS now defaults to TRUE in order to better support the general case: the value FALSE is appropriate only when you know that all references are exact. See AMS (Automatic Mark and Sweep).

4. There is now a default value for the MPS_KEY_AWL_FIND_DEPENDENT keyword argument to mps_pool_create_k() when creating an AWL (Automatic Weak Linked) pool. The default value is a function that always returns NULL (meaning that there is no dependent object).

5. It is now possible to configure the alignment of objects allocated in a MV (Manual Variable) pool, by passing the MPS_KEY_ALIGN keyword argument to mps_pool_create_k().

6. The MVFF (Manual Variable First Fit) pool class takes a new keyword argument MPS_KEY_SPARE. This specifies the maximum proportion of memory that the pool will keep spare for future allocations.
7. The alignment requirements for **MVFF (Manual Variable First Fit)** and **MVT (Manual Variable Temporal)** pools have been relaxed on the platforms w3i3mv and w3i6mv. On all platforms it is now possible to specify alignments down to `sizeof(void *)` as the alignment for pools of these classes.

8. The sizes of the templates in a `mps_pool_debug_option_s` structure no longer have to be related to the alignment of the pools that they are used with. This makes it easier to reuse these structures.

### Other changes

1. The **AMS (Automatic Mark and Sweep)** pool class no longer triggers the assertion `!AMS_IS_INVALID_COLOUR(seg, i)` under rare circumstances (namely, detaching an allocation point from a grey segment when `MPS_KEY_AMS_SUPPORT_AMBIGUOUS` is `FALSE`). See job001549.

2. `mps_arena_roots_walk()` no longer triggers an assertion failure when run twice in succession. See job003496.

3. The alignment of **AWL (Automatic Weak Linked)** pools is now configurable via the object format, as documented, and is no longer always `MPS_PF_ALIGN`. See job003745.

4. The debugging version of the **MVFF (Manual Variable First Fit)** pool class, `mps_class_mvff_debug()`, no longer triggers an assertion failure if you allocate a large object. See job003751.

5. `mpseventtxt` now successfully processes a telemetry log containing multiple labels associated with the same address. See job003756.

6. **AMS (Automatic Mark and Sweep)**, **AWL (Automatic Weak Linked)** and **LO (Leaf Object)** pools get reliably collected, even in the case where the pool is the only pool on its generation chain and is allocating into some generation other than the nursery. See job003771.

7. Allocation into **AWL (Automatic Weak Linked)** pools again reliably provokes garbage collections of the generation that the pool belongs to. (In version 1.113, the generation would only be collected if a pool of some other class allocated into it.) See job003772.

8. All unreachable objects in **LO (Leaf Object)** pools are finalized. (Previously, objects on a segment attached to an allocation point were not finalized until the allocation point was full.) See job003773.

9. The **MVT (Manual Variable Temporal)** and **MVFF (Manual Variable First Fit)** pool classes are now around 25% faster (in our benchmarks) than they were in version 1.113.

10. The default assertion handler in the default `plinth` now flushes the telemetry stream before aborting. See `mps_lib_assert_fail()`.

11. Garbage collection performance is substantially improved in the situation where the arena has been extended many times. Critical operations now take time logarithmic in the number of times the arena has been extended (rather than linear, as in version 1.113 and earlier). See job003554.

### 6.6.3 Release 1.113.0

#### New features

1. In previous releases there was an implicit connection between blocks allocated by **AWL (Automatic Weak Linked)** and **LO (Leaf Object)** pools, and blocks allocated by other automatically managed pool classes.

   In particular, blocks allocated by AWL and LO pools were garbage collected together with blocks allocated by **AMS (Automatic Mark and Sweep)** pools, and blocks allocated by **AMC (Automatic Mostly-Copying)** pools in generation 1 of their chains.
This is no longer the case: to arrange for blocks to be collected together you need to ensure that they are allocated in the same generation chain, using the MPS_KEY_CHAIN and MPS_KEY_GEN keyword arguments to `mps_pool_create_k()`.

So if you have code like this:

```
res = mps_pool_create(&my_amc, arena, mps_class_amc(), my_chain);
res = mps_pool_create(&my_awl, arena, mps_class_awl());
```

and you want to retain the connection between these pools, then you must ensure that they use the same generation chain:

```
MPS_ARGS_BEGIN(args) {
  MPS_ARGS_ADD(args, MPS_KEY_CHAIN, my_chain);
  res = mps_pool_create_k(&my_amc, arena, mps_class_amc(), args);
} MPS_ARGS_END(args);

MPS_ARGS_BEGIN(args) {
  MPS_ARGS_ADD(args, MPS_KEY_CHAIN, my_chain);
  MPS_ARGS_ADD(args, MPS_KEY_GEN, 1);
  res = mps_pool_create_k(&my_awl, arena, mps_class_awl(), args);
} MPS_ARGS_END(args);
```

### Interface changes

1. When creating a list of keyword arguments, there is no longer any need to call `MPS_ARGS_DONE()`. See Keyword arguments.
2. When creating an automatically managed pool using `mps_pool_create_k()`, it is no longer necessary to pass in a generation chain. The arena has a default generation chain and this is used by all automatically managed pools where no generation chain was specified.
3. It is now possible to specify a generation chain for AWL (Automatic Weak Linked) and LO (Leaf Object) pool classes, by using the optional MPS_KEY_CHAIN keyword argument to `mps_pool_create_k()`.
4. It is now possible to specify which generation the AMS (Automatic Mark and Sweep), AWL (Automatic Weak Linked), and LO (Leaf Object) pool classes allocate new objects into, using the optional MPS_KEY_GEN keyword argument to `mps_pool_create_k()`.

### Other changes

1. The MPS now retains some unused memory instead of returning it to the operating system. This reduces unnecessary overhead due to system calls, thrashing the operating system’s page table, and zeroing memory when re-allocated. See `job003700`.

### 6.6.4 Release 1.112.0

#### New features

1. New supported platform lli6ll (Linux, x86-64, Clang/LLVM).
2. On Windows, you can now request that the MPS allocate address space from the top down, allowing a 32-bit executable linked with `/LARGEADDRESSAWARE` to use the top half of the address space. Use the keyword argument `MPS_KEY_VMW3_TOP_DOWN` when creating an arena of class `mps_arena_class_vm()`.
3. On OS X, multi-threaded programs are now supported. See `Threads`.
4. On OS X, you can now debug the MPS using `lldb`.

**Interface changes**

1. In the *hot* (production) variety, the default assertion handler now prints messages to standard error but does *not* terminate the program. Even though assertions indicate serious problems in the program, an end-user does not always want an application to terminate when there is a chance to shut down safely and save work, or even to limp along indefinitely. See [Assertion handling](#).

2. The behaviour when an assertion is triggered is now configurable in the default *plinth* by installing an assertion handler. See `mps_lib_assert_fail_install()`.

3. Functions that take a variable number of arguments (`mps_arena_create()`, `mps_pool_create()`, `mps_ap_create()`) and their `va_list` alternatives (`mps_arena_create_v()` etc.) are now deprecated in favour of functions that use a *keyword argument* interface (`mps_arena_create_k()`, `mps_pool_create_k()`, `mps_ap_create_k()`).

   Similarly, the object format variant structures (`mps_fmt_A_s` etc.) and the functions that take them as arguments (`mps_fmt_create_A()` etc.) are now deprecated in favour of `mps_fmt_create_k()`.

   The new interfaces provide better reporting of errors, default values for arguments, and forward compatibility. See [Keyword arguments](#).

   The old interfaces continue to be supported for now, but new features will become available through the keyword interface only.

4. *MFS (Manual Fixed Small)* pools no longer refuse to manage blocks that are smaller than the platform alignment. They now round up smaller sizes internally if necessary.

5. *MVT (Manual Variable Temporal)* pools now allow the client to specify the alignment of blocks. Use the keyword argument `MPS_KEY_ALIGN` when creating a pool of class `mps_class_mvt()`.

6. On OS X, signals are no longer used for handling memory protection exceptions. This means that programs are free to handle `SIGBUS`, but must not install a thread-local Mach exception handler for `EXC_BAD_ACCESS` exceptions. See [Signal and exception handling issues](#).

7. On OS X, when debugging with `gdb`, you no longer need to turn on `dont-handle-bad-access` or to request special handling of `SIGBUS`.

**Other changes**

1. On Windows, an execute exception no longer triggers an assertion. See job003301.

2. Rehashing of large address-based hash tables no longer provokes a nursery collection that immediately renders the hash table stale again. See job003435.

3. An *MVT (Manual Variable Temporal)* pool no longer triggers an assertion failure when it runs out of space on its reserved block queue. See job003486.

4. The `-i` and `-o` options no longer cause `mpseventsql` to crash. See job003507.

5. On Windows, telemetry files now have correct clock values. Previously the top 32 bits were incorrectly output as zero. See job003519.

6. On 64-bit Windows, it’s no longer possible to get a stack overflow exception while the MPS is holding the arena lock. See job003640.
6.6.5 Release 1.111.0

New features

1. Reporting features have been removed from the mpseventcnv utility. Instead, the telemetry system comes with two new utility programs to assist with reporting and analysis: mpseventtxt converts an event stream into human-readable form, and mpseventsql loads an event stream into a SQLite database for further analysis. See Telemetry.

2. The new pool class MFS (Manual Fixed Small) provides manually managed allocation of fixed-size objects.

3. The new pool class MVT (Manual Variable Temporal) provides manually managed allocation of variable-size objects using a temporal fit allocation policy (that is, objects that are allocated together are expected to be freed together).

Interface changes

1. It is no longer necessary for client programs to use mps_tramp() to ensure that exceptions due to barrier hits are caught. This function is now deprecated.

2. You can set the environment variable MPS_TELEMETRY_CONTROL to all to make the telemetry system output all events. See Telemetry.

3. New functions mps_telemetry_get(), mps_telemetry_set() and mps_telemetry_reset() provide a more convenient interface to telemetry control than mps_telemetry_control(), which is now deprecated. See Telemetry.

4. The pool classes MV (Manual Variable) and SNC (Stack No Checking) are now deprecated.

5. Allocation frames are now deprecated. See Allocation frames.

6. Additionally, the functions mps_arena_expose(), mps_arenaUnsafe_expose_remember_protection(), mps_arenaUnsafe_restore_protection(), mps_arena_roots_walk() and mps_fix() are now deprecated.

Other changes

1. mps_arena_step() no longer unclamps the arena as a side effect. If the arena is clamped or parked before calling mps_arena_step(), it is clamped afterwards. See job003320.

2. The ambiguous stack scanner, mps_stack_scan_ambig(), no longer asserts on Linux when there are multiple threads. See job003412.

3. It is no longer possible for the “ramp” allocation pattern, mps_alloc_pattern_ramp(), to get stuck. Now mps_ap_alloc_pattern_end() reliably clears this pattern. See job003454.

4. The build system now correctly detects the FreeBSD operating system running on the x86-64 architecture, for FreeBSD version 9.1 or later. See job003473.

6.6.6 Release 1.110.0

New features

1. New supported platforms:
   - fri6gc (FreeBSD, x86-64, GCC)
   - lii6gc (Linux, x86-64, GCC)
• w3i6mv (Windows, x86-64, Microsoft Visual C)
• xci3ll (OS X, IA-32, Clang/LLVM)
• xci6gc (OS X, x86-64, GCC)
• xci6ll (OS X, x86-64, Clang/LLVM)

2. Support removed for platforms:
• iam4cc (Irix 6, MIPS R4000, MIPSpro C)
• lii3eg (Linux, IA-32, EGCS)
• lippgc (Linux, PowerPC, GCC)
• o1alcc (OSF/1, Alpha, Digital C)
• o1algc (OSF/1, Alpha, GCC)
• s7ppmw (System 7, PowerPC, MetroWerks C)
• sos8gc (Solaris, SPARC 8, GCC)
• sos9sc (Solaris, SPARC 9, SunPro C)
• sus8gc (SunOS, SPARC 8, GCC)
• xcpggc (OS X, PowerPC, GCC)

3. On Unix platforms, the MPS can now be built and installed by running `./configure && make install`. See Building the Memory Pool System.

4. The MPS can be compiled in a single step via the new source file `mps.c`. This also allows you to compile the MPS in the same compilation unit as your object format, allowing the compiler to perform global optimizations between the two. See Building the Memory Pool System.

5. The set of build varieties has been reduced to three: the cool variety for development and debugging, the hot variety for production, and the rash variety for people who like to live dangerously. See Varieties.

6. The environment variable `MPS_TELEMETRY_CONTROL` can now be set to a space-separated list of event kinds. See Telemetry.

7. Telemetry output is now emitted to the file named by the environment variable `MPS_TELEMETRY_FILENAME`, if it is set. See Telemetry.

Interface changes

1. Deprecated constants `MPS_MESSAGE_TYPE_FINALIZATION`, `MPS_MESSAGE_TYPE_GC` and `MPS_MESSAGE_TYPE_GC_START` have been removed. Use `mps_message_type_finalization()`, `mps_message_type_gc()` and `mps_message_type_gc_start()` instead.

2. Deprecated constants `MPS_RANK_AMBIG`, `MPS_RANK_EXACT` and `MPS_RANK_WEAK` have been removed. Use `mps_rank_ambig()`, `mps_rank_exact()` and `mps_rank_weak()` instead.

3. Deprecated functions with names starting `mps_space_` have been removed. Use the functions with names starting `mps_arena_` instead.

• genindex
6.7 Introduction to memory management

6.7.1 Overview

Memory management is a complex field of computer science and there are many techniques being developed to make it more efficient. This guide is designed to introduce you to some of the basic memory management issues that programmers face.

This guide attempts to explain any terms it uses as it introduces them. In addition, there is a Memory Management Glossary of memory management terms that gives fuller information; some terms are linked to the relevant entries.

Memory management is usually divided into three areas, although the distinctions are a little fuzzy:

- **Hardware memory management**
- **Operating system memory management**
- **Application memory management**

These are described in more detail below. In most computer systems, all three are present to some extent, forming layers between the user’s program and the actual memory hardware. The Memory Management Reference is mostly concerned with application memory management.

**Hardware memory management**

Memory management at the hardware level is concerned with the electronic devices that actually store data. This includes things like RAM and memory caches.

**Operating system memory management**

In the operating system, memory must be allocated to user programs, and reused by other programs when it is no longer required. The operating system can pretend that the computer has more memory than it actually does, and also that each program has the machine’s memory to itself; both of these are features of virtual memory systems.

**Application memory management**

Application memory management involves supplying the memory needed for a program’s objects and data structures from the limited resources available, and recycling that memory for reuse when it is no longer required. Because application programs cannot in general predict in advance how much memory they are going to require, they need additional code to handle their changing memory requirements.

Application memory management combines two related tasks:

**Allocation**

When the program requests a block of memory, the memory manager must allocate that block out of the larger blocks it has received from the operating system. The part of the memory manager that does this is known as the allocator. There are many ways to perform allocation, a few of which are discussed in Allocation techniques.

**Recycling**

When memory blocks have been allocated, but the data they contain is no longer required by the program, then the blocks can be recycled for reuse. There are two approaches to recycling memory: either the programmer must decide when memory can be reused (known as manual memory management); or the memory manager must be able to work it out (known as automatic memory management). These are both described in more detail below.
An application memory manager must usually work to several constraints, such as:

**CPU overhead**

The additional time taken by the memory manager while the program is running.

**Pause times**

The time it takes for the memory manager to complete an operation and return control to the program.

This affects the program’s ability to respond promptly to interactive events, and also to any asynchronous event such as a network connection.

**Memory overhead**

How much space is wasted for administration, rounding (known as *internal fragmentation*), and poor layout (known as *external fragmentation*).

Some of the common problems encountered in application memory management are considered in the next section.

**Memory management problems**

The basic problem in managing memory is knowing when to keep the data it contains, and when to throw it away so that the memory can be reused. This sounds easy, but is, in fact, such a hard problem that it is an entire field of study in its own right. In an ideal world, most programmers wouldn’t have to worry about memory management issues. Unfortunately, there are many ways in which poor memory management practice can affect the robustness and speed of programs, both in manual and in automatic memory management.

Typical problems include:

**Premature frees and dangling pointers**

Many programs give up memory, but attempt to access it later and crash or behave randomly. This condition is known as a *premature free*, and the surviving reference to the memory is known as a *dangling pointer*. This is usually confined to *manual memory management*.

**Memory leak**

Some programs continually allocate memory without ever giving it up and eventually run out of memory. This condition is known as a *memory leak*.

**External fragmentation**

A poor allocator can do its job of giving out and receiving blocks of memory so badly that it can no longer give out big enough blocks despite having enough spare memory. This is because the free memory can become split into many small blocks, separated by blocks still in use. This condition is known as *external fragmentation*.

**Poor locality of reference**

Another problem with the layout of allocated blocks comes from the way that modern hardware and operating system memory managers handle memory: successive memory accesses are faster if they are to nearby memory locations. If the memory manager places far apart the blocks a program will use together, then this will cause performance problems. This condition is known as poor *locality of reference*.

**Inflexible design**

Memory managers can also cause severe performance problems if they have been designed with one use in mind, but are used in a different way. These problems occur because any memory management solution tends to make assumptions about the way in which the program is going to use memory, such as typical block sizes, reference patterns, or lifetimes of objects. If these assumptions are wrong, then the memory manager may spend a lot more time doing bookkeeping work to keep up with what’s happening.
Interface complexity

If objects are passed between modules, then the interface design must consider the management of their memory.

A well-designed memory manager can make it easier to write debugging tools, because much of the code can be shared. Such tools could display objects, navigate links, validate objects, or detect abnormal accumulations of certain object types or block sizes.

Manual memory management

Manual memory management is where the programmer has direct control over when memory may be recycled. Usually this is either by explicit calls to heap management functions (for example, `malloc` and `free` in C), or by language constructs that affect the control stack (such as local variables). The key feature of a manual memory manager is that it provides a way for the program to say, “Have this memory back; I’ve finished with it”; the memory manager does not recycle any memory without such an instruction.

The advantages of manual memory management are:

- it can be easier for the programmer to understand exactly what is going on;
- some manual memory managers perform better when there is a shortage of memory.

The disadvantages of manual memory management are:

- the programmer must write a lot of code to do repetitive bookkeeping of memory;
- memory management must form a significant part of any module interface;
- manual memory management typically requires more memory overhead per object;
- memory management bugs are common.

It is very common for programmers, faced with an inefficient or inadequate manual memory manager, to write code to duplicate the behavior of a memory manager, either by allocating large blocks and splitting them for use, or by recycling blocks internally. Such code is known as a suballocator. Suballocators can take advantage of special knowledge of program behavior, but are less efficient in general than fixing the underlying allocator. Unless written by a memory management expert, suballocators may be inefficient or unreliable.

The following languages use mainly manual memory management in most implementations, although many have conservative garbage collection extensions: Algol; C; C++; COBOL; Fortran; Pascal.

Automatic memory management

Automatic memory management is a service, either as a part of the language or as an extension, that automatically recycles memory that a program would not otherwise use again. Automatic memory managers (often known as garbage collectors, or simply collectors) usually do their job by recycling blocks that are unreachable from the program variables (that is, blocks that cannot be reached by following pointers).

The advantages of automatic memory management are:

- the programmer is freed to work on the actual problem;
- module interfaces are cleaner;
- there are fewer memory management bugs;
- memory management is often more efficient.

The disadvantages of automatic memory management are:

- memory may be retained because it is reachable, but won’t be used again;
• automatic memory managers (currently) have limited availability.

There are many ways of performing automatic recycling of memory, a few of which are discussed in Recycling techniques.

Most modern languages use mainly automatic memory management: BASIC, Dylan, Erlang, Haskell, Java, JavaScript, Lisp, ML, Modula-3, Perl, PostScript, Prolog, Python, Scheme, Smalltalk, etc.

More information

For more detailed information on the topics covered briefly above, please have a look at the Memory Management Glossary. Books and research papers are available on many specific techniques, and can be found via our Bibliography; particularly recommended are: Wilson (1994), which is survey of garbage collection techniques; Wilson et al. (1995), which is a survey of allocation techniques; and Jones et al. (2012), which is a handbook covering all aspects of garbage collection.

6.7.2 Allocation techniques

Memory allocation is the process of assigning blocks of memory on request. Typically the allocator receives memory from the operating system in a small number of large blocks that it must divide up to satisfy the requests for smaller blocks. It must also make any returned blocks available for reuse. There are many common ways to perform this, with different strengths and weaknesses. A few are described briefly below.

• First fit
• Buddy system
• Suballocators

These techniques can often be used in combination.

First fit

In the first fit algorithm, the allocator keeps a list of free blocks (known as the free list) and, on receiving a request for memory, scans along the list for the first block that is large enough to satisfy the request. If the chosen block is significantly larger than that requested, then it is usually split, and the remainder added to the list as another free block.

The first fit algorithm performs reasonably well, as it ensures that allocations are quick. When recycling free blocks, there is a choice as to where to add the blocks to the free list—effectively in what order the free list is kept:

Memory location (address)

This is not fast for allocation or recycling, but supports efficient merging of adjacent free blocks (known as coalescence). According to Wilson et al. (1995), this ordering reduces fragmentation. It can also improve locality of reference.

Increasing size

This is equivalent to the best fit algorithm, in that the free block with the “tightest fit” is always chosen. The fit is usually sufficiently tight that the remainder of the block is unusably small.

Decreasing size

This is equivalent to the worst fit algorithm. The first block on the free list will always be large enough, if a large enough block is available. This approach encourages external fragmentation, but allocation is very fast.

Increasing time since last use
This is very fast at adding new free blocks, because they are added to the beginning of the list. It encourages good *locality of reference* (where blocks used together are not spread throughout memory), but can lead to bad external fragmentation.

A variation of first fit, known as *next fit*, continues each search for a suitable block where the previous one left off, by using a roving pointer into the free block chain. This is not usually combined with increasing or decreasing size ordering because it would eliminate their advantages.

**Buddy system**

In a *buddy system*, the allocator will only allocate blocks of certain sizes, and has many free lists, one for each permitted size. The permitted sizes are usually either powers of two, or form a Fibonacci sequence (see below for example), such that any block except the smallest can be divided into two smaller blocks of permitted sizes.

When the allocator receives a request for memory, it rounds the requested size up to a permitted size, and returns the first block from that size’s free list. If the free list for that size is empty, the allocator splits a block from a larger size and returns one of the pieces, adding the other to the appropriate free list.

When blocks are recycled, there may be some attempt to merge adjacent blocks into ones of a larger permitted size (*coalescence*). To make this easier, the free lists may be stored in order of address. The main advantage of the buddy system is that coalescence is cheap because the “buddy” of any free block can be calculated from its address.

![Fig. 6.18: A binary buddy heap before allocation](image1)

![Fig. 6.19: A binary buddy heap after allocating a 8 kB block.](image2)

![Fig. 6.20: A binary buddy heap after allocating a 10 kB block; note the 6 kB wasted because of rounding up.](image3)

For example, an allocator in a binary buddy system might have sizes of 16, 32, 64, . . . , 64 kB. It might start off with a single block of 64 kB. If the application requests a block of 8 kB, the allocator would check its 8 kB free list and find no free blocks of that size. It would then split the 64 kB block into two block of 32 kB, split one of them into two blocks of 16 kB, and split one of them into two blocks of 8 kB. The allocator would then return one of the 8 kB blocks to the application and keep the remaining three blocks of 8 kB, 16 kB, and 32 kB on the appropriate free lists. If the application then requested a block of 10 kB, the allocator would round this request up to 16 kB, and return the 16 kB block from its free list, wasting 6 kB in the process.

A Fibonacci buddy system might use block sizes 16, 32, 48, 80, 128, 208, . . . bytes, such that each size is the sum of the two preceding sizes. When splitting a block from one free list, the two parts get added to the two preceding free lists.

A buddy system can work very well or very badly, depending on how the chosen sizes interact with typical requests for memory and what the pattern of returned blocks is. The rounding typically leads to a significant amount of wasted memory, which is called *internal fragmentation*. This can be reduced by making the permitted block sizes closer together.

**Suballocators**

There are many examples of application programs that include additional memory management code called a *suballocator*. A suballocator obtains large blocks of memory from the system memory manager and allocates the memory to the application in smaller pieces. Suballocators are usually written for one of the following reasons:

- To avoid general inefficiency in the system memory manager;
• To take advantage of special knowledge of the application’s memory requirements that cannot be expressed to the system memory manager;
• To provide memory management services that the system memory manager does not supply.

In general, suballocators are less efficient than having a single memory manager that is well-written and has a flexible interface. It is also harder to avoid memory management bugs if the memory manager is composed of several layers, and if each application has its own variation of suballocator.

Many applications have one or two sizes of block that form the vast majority of their allocations. One of the most common uses of a suballocator is to supply the application with objects of one size. This greatly reduces the problem of external fragmentation. Such a suballocator can have a very simple allocation policy.

There are dangers involved in making use of special knowledge of the application’s memory requirements. If those requirements change, then the performance of the suballocator is likely to be much worse than that of a general allocator. It is often better to have a memory manager that can respond dynamically to changing requirements.

6.7.3 Recycling techniques

There are many ways for automatic memory managers to determine what memory is no longer required. In the main, garbage collection relies on determining which blocks are not pointed to by any program variables. Some of the techniques for doing this are described briefly below, but there are many potential pitfalls, and many possible refinements. These techniques can often be used in combination.

**Tracing collectors**

Automatic memory managers that follow pointers to determine which blocks of memory are reachable from program variables (known as the *root set*) are known as *tracing* collectors. The classic example is the mark-sweep collector.

**Mark-sweep collection**

In a *mark-sweep* collection, the collector first examines the program variables; any blocks of memory pointed to are added to a list of blocks to be examined. For each block on that list, it sets a flag (the mark) on the block to show that it is still required, and also that it has been processed. It also adds to the list any blocks pointed to by that block that have not yet been marked. In this way, all blocks that can be reached by the program are marked.

In the second phase, the collector sweeps all allocated memory, searching for blocks that have not been marked. If it finds any, it returns them to the allocator for reuse.

![Fig. 6.21: Five memory blocks, three of which are reachable from program variables.](image)

In the diagram above, block 1 is directly accessible from a program variable, and blocks 2 and 3 are indirectly accessible. Blocks 4 and 5 cannot be reached by the program. The first step would mark block 1, and remember blocks 2 and 3 for later processing. The second step would mark block 2. The third step would mark block 3, but wouldn’t remember block 2 as it is already marked. The sweep phase would ignore blocks 1, 2, and 3 because they are marked, but would recycle blocks 4 and 5.

The two drawbacks of simple mark-sweep collection are:

• it must scan the entire memory in use before any memory can be freed;
• it must run to completion or, if interrupted, start again from scratch.

If a system requires real-time or interactive response, then simple mark-sweep collection may be unsuitable as it stands, but many more sophisticated garbage collection algorithms are derived from this technique.
Copying collection

After many memory blocks have been allocated and recycled, there are two problems that typically occur:

- the memory in use is widely scattered in memory, causing poor performance in the memory caches or virtual memory systems of most modern computers (known as poor locality of reference);
- it becomes difficult to allocate large blocks because free memory is divided into small pieces, separated by blocks in use (known as external fragmentation).

One technique that can solve both these problems is copying garbage collection. A copying garbage collector may move allocated blocks around in memory and adjust any references to them to point to the new location. This is a very powerful technique and can be combined with many other types of garbage collection, such as mark-sweep collection.

The disadvantages of copying collection are:

- it is difficult to combine with incremental garbage collection (see below) because all references must be adjusted to remain consistent;
- it is difficult to combine with conservative garbage collection (see below) because references cannot be confidently adjusted;
- extra storage is required while both new and old copies of an object exist;
- copying data takes extra time (proportional to the amount of live data).

Incremental collection

Older garbage collection algorithms relied on being able to start collection and continue working until the collection was complete, without interruption. This makes many interactive systems pause during collection, and makes the presence of garbage collection obtrusive.

Fortunately, there are modern techniques (known as incremental garbage collection) to allow garbage collection to be performed in a series of small steps while the program is never stopped for long. In this context, the program that uses and modifies the blocks is sometimes known as the mutator. While the collector is trying to determine which blocks of memory are reachable by the mutator, the mutator is busily allocating new blocks, modifying old blocks, and changing the set of blocks it is actually looking at.

Incremental collection is usually achieved with either the cooperation of the memory hardware or the mutator; this ensures that, whenever memory in crucial locations is accessed, a small amount of necessary bookkeeping is performed to keep the collector’s data structures correct.

Conservative garbage collection

Although garbage collection was first invented in 1958, many languages have been designed and implemented without the possibility of garbage collection in mind. It is usually difficult to add normal garbage collection to such a system, but there is a technique, known as conservative garbage collection, that can be used.

The usual problem with such a language is that it doesn’t provide the collector with information about the data types, and the collector cannot therefore determine what is a pointer and what isn’t. A conservative collector assumes that anything might be a pointer. It regards any data value that looks like a pointer to or into a block of allocated memory as preventing the recycling of that block.

Note that, because the collector does not know for certain which memory locations contain pointers, it cannot readily be combined with copying garbage collection. Copying collection needs to know where pointers are in order to update them when blocks are moved.
You might think that conservative garbage collection could easily perform quite poorly, leaving a lot of garbage uncollected. In practice, it does quite well, and there are refinements that improve matters further.

**Reference counts**

A reference count is a count of how many references (that is, pointers) there are to a particular memory block from other blocks. It is used as the basis for some automatic recycling techniques that do not rely on tracing.

**Simple reference counting**

In a simple reference counting system, a reference count is kept for each object. This count is incremented for each new reference, and is decremented if a reference is overwritten, or if the referring object is recycled. If a reference count falls to zero, then the object is no longer required and can be recycled.

Reference counting is frequently chosen as an automatic memory management strategy because it seems simple to implement using manual memory management primitives. However, it is hard to implement efficiently because of the cost of updating the counts. It is also hard to implement reliably, because the standard technique cannot reclaim objects connected in a loop. In many cases, it is an inappropriate solution, and it would be preferable to use tracing garbage collection instead.

Reference counting is most useful in situations where it can be guaranteed that there will be no loops and where modifications to the reference structure are comparatively infrequent. These circumstances can occur in some types of database structure and some file systems. Reference counting may also be useful if it is important that objects are recycled promptly, such as in systems with tight memory constraints.

**Deferred reference counting**

The performance of reference counting can be improved if not all references are taken into account. In one important technique, known as deferred reference counting, only references from other objects are counted, and references from program variables are ignored. Since most of the references to the object are likely to be from local variables, this can substantially reduce the overhead of keeping the counts up to date. An object cannot be reclaimed as soon as its count has dropped to zero, because there might still be a reference to it from a program variable. Instead, the program variables (including the control stack) are periodically scanned, and any objects which are not referenced from there and which have zero count are reclaimed.

Deferred reference counting cannot normally be used unless it is directly supported by the compiler. It’s more common for modern compilers to support tracing garbage collectors instead, because they can reclaim loops. Deferred reference counting may still be useful for its promptness—but that is limited by the frequency of scanning the program variables.

**One-bit reference counting**

Another variation on reference counting, known as the one-bit reference count, uses a single bit flag to indicate whether each object has either “one” or “many” references. If a reference to an object with “one” reference is removed, then the object can be recycled. If an object has “many” references, then removing references does not change this, and that object will never be recycled. It is possible to store the flag as part of the pointer to the object, so no additional space is required in each object to store the count. One-bit reference counting is effective in practice because most actual objects have a reference count of one.
Weighted reference counting

Reference counting is often used for tracking inter-process references for distributed garbage collection. This fails to collect objects in separate processes if they have looped references, but tracing collectors are usually too inefficient as inter-process tracing entails much communication between processes. Within a process, tracing collectors are often used for local recycling of memory.

Many distributed collectors use a technique called weighted reference counting, which reduces the level of communication even further. Each time a reference is copied, the weight of the reference is shared between the new and the old copies. Since this operation doesn’t change the total weight of all references, it doesn’t require any communication with the object. Communication is only required when references are deleted.

6.7.4 Memory management in various languages

**ALGOL**  ALGOL, designed in 1958 for scientific computing, was the first block-structured language. It spawned a whole family of languages, and inspired many more, including Scheme, Simula and Pascal.

The block structure of ALGOL 60 induced a stack allocation discipline. It had limited dynamic arrays, but no general heap allocation. The substantially redesigned ALGOL 68 had both heap and stack allocation. It also had something like the modern pointer type, and required garbage collection for the heap. The new language was complex and difficult to implement, and it was never as successful as its predecessor.

**Related publication**

Branquart & Lewi (1972).

**BASIC**  BASIC is a simple and easily-learned programming language created by T. E. Kurtz and J. G. Kemeny in 1963–4. The motivation was to make computers easily accessible to undergraduate students in all disciplines.

Most BASICs had quite powerful string handling operations that required a simple garbage collector. In many implementations, the garbage collector could be forced to run by running the mysterious expression `FRE (" ")`.

BASIC is now old-fashioned, but survives as a scripting language, in particular in Visual BASIC, which is an application development environment with a BASIC-like scripting language. These descendants invariably have automatic memory management as well.

**C**  C is a systems programming language sometimes described as “a portable assembler” because it was intended to be sufficiently low-level to allow performance comparable to assembler or machine code, but sufficiently high-level to allow programs to be reused on other platforms with little or no modification.

Memory management is typically manual (the standard library functions for memory management in C, `malloc` and `free`, have become almost synonymous with manual memory management), although with the Memory Pool System, or the Boehm–Demers–Weiser collector, it is now possible to use garbage collection.

The language is notorious for fostering memory management bugs, including:

1. Accessing arrays with indexes that are out of bounds;
2. Using stack-allocated structures beyond their lifetimes (see use after free);
3. Using heap-allocated structures after freeing them (see use after free);
4. Neglecting to free heap-allocated objects when they are no longer required (see memory leak);
5. Failing to allocate memory for a pointer before using it;
6. Allocating insufficient memory for the intended contents;
7. Loading from allocated memory before storing into it;
8. Dereferencing non-pointers as if they were pointers.
See also:

*automatic storage duration, static storage duration.*

Related publications


Related links


COBOL  COBOL was designed by the CODASYL committee in 1959–60 to be a business programming language, and has been extended many times since. A 1997 Gartner Group report estimated that 80% of computer software (by count of source lines) was written in COBOL.

Prior to 2002, COBOL had no *heap allocation*, and did well in its application domain without it. COBOL 2002 has *pointers* and heap allocation through `ALLOCATE` and `FREE`, mainly in order to be able to use C-style interfaces. It also supports a high level of abstraction through object-oriented programming and *garbage collection* (including *finalization*).

Related link

COBOL standardization.

Common Lisp  Common Lisp is the major dialect of the *Lisp* family. In addition to the usual Lisp features, it has an advanced object system, data types from hash tables to complex numbers, and a rich standard library.

Common Lisp is a *garbage-collected* language, and modern implementations, such as LispWorks and Allegro CL, include advanced features, such as *finalization* and *weakness*.

Related link

Common Lisp HyperSpec.

C#  C# is a strongly typed object-oriented language created at Microsoft in 1999–2000. It is designed to run on the Common Language Runtime, the virtual machine from the .NET Framework. It also runs on the open source Mono runtime.

Memory is *automatically managed*: memory is allocated when an object is created, and reclaimed at some point after the object becomes *unreachable*.

The language supports *finalization* (classes may have *destructor functions*, which are run just before the object is reclaimed by the memory manager), and *weak references* (1) (via the *WeakReference class*).

The *garbage collector* in the .NET Framework is configurable to run in soft real time, or in batch mode.

The Mono runtime comes with two collectors: the Boehm–Demers–Weiser *conservative collector*, and a *generational copying collector*.

Related links

Automatic memory management in C#, WeakReference Class, Memory Management and Garbage Collection in the .NET Framework, Mono project.

C++  C++ is a (weakly) object-oriented language, extending the systems programming language *C* with a multiple-inheritance class mechanism and simple method dispatch.
The standard library functions for memory management in C++ are `new` and `delete`. The higher abstraction level of C++ makes the bookkeeping required for manual memory management even harder. Although the standard library provides only manual memory management, with the Memory Pool System, or the Boehm–Demers–Weiser collector, it is now possible to use garbage collection. Smart pointers are another popular solution.

The language is notorious for fostering memory management bugs, including:

1. Using stack-allocated structures beyond their lifetimes (see `use after free`);
2. Using heap-allocated structures after freeing them (see `use after free`);
3. Neglecting to free heap-allocated objects when they are no longer required (see `memory leak`);
4. Excessive copying by copy constructors (1);
5. Unexpected sharing due to insufficient copying by copy constructors;
6. Allocating insufficient memory for the intended contents;
7. Accessing arrays with indexes that are out of bounds.

**Historical note**

C++ was designed by Bjarne Stroustrup, as a minimal object-oriented extension to C. It has since grown to include some other modern programming language ideas. The first implementations were preprocessors that produced C code, but modern implementations are dedicated C++ compilers.

Ellis and Stroustrup write in *The Annotated C++ Reference Manual*:

C programmers think memory management is too important to be left to the computer. Lisp programmers think memory management is too important to be left to the user.

**See also:**

`constructor (2), destructor (2).`

**Related publications**


**Related link**


**Dylan**

Dylan is a modern programming language invented by Apple around 1993 and developed by Harlequin and other partners. The language is a distillation of the best ideas in dynamic and object-oriented programming. Its ancestors include *Lisp, Smalltalk,* and *C++*. Dylan is aimed at building modular component software and delivering safe, compact applications. It also facilitates the rapid development and incremental refinement of prototype programs.

Dylan provides *automatic memory management*. The generic allocation function is called `make`. Most implementations provide `finalization` and `weak` hash tables, although interfaces for these features have not yet been standardized. An object may be registered for finalization via the function `finalize-when-unreachable`, in which case there will be a call to the `finalize` function once the garbage collector has determined that the object is `unreachable`. Weak hash tables may have either weak keys or values, depending on a parameter supplied at allocation time. A hash table entry will be deleted once the garbage collector has determined that there are no `strong references` to the key or value of the entry, for weak key or value tables, respectively.

**Related link**
Emacs Lisp  Emacs Lisp or elisp is a dialect of Lisp used in the Emacs family of text editors, of which the most widely-used is GNU Emacs.

Like most Lisps, Emacs Lisp requires garbage collection. GNU Emacs has a simple mark-sweep collector. It has been speculated that the non-incremental nature of the Emacs collector, combined with the fact that, prior to version 19.31 (May 1996), it printed a message whenever it collected, gave garbage collection a bad name in programming circles.

Erik Naggum reported at the time:

I have run some tests at the U of Oslo with about 100 users who generally agreed that Emacs had become faster in the latest Emacs pretest. All I had done was to remove the “Garbage collecting” message which people perceive as slowing Emacs down and tell them that it had been sped up. It is, somehow, permissible for a program to take a lot of time doing any other task than administrative duties like garbage collection.

Emacs was originally written in Teco, not in Lisp, but it still had a garbage collector, though this was heuristic and conservative in nature. Teco-based Emacs was capable of running for weeks at a time in a 256 kB address space.

Related links

Fortran  Fortran, created in 1957, was one of the first languages qualifying as a high-level language. It is popular among scientists and has substantial support in the form of numerical libraries.

Early versions of Fortran required the size of arrays to be known at compilation time, and the earliest Fortran compilers accordingly used only static allocation (however, the 1966 standard gave compiler writers freedom to use other allocation mechanisms).

The Fortran 90 standard added recursion and automatic arrays with stack allocation semantics (though many compilers in fact allocate them on the heap). It also added dynamic allocation using ALLOCATE with manual deallocation using DEALLOCATE. Fortran 95 made it explicit that allocated arrays have dynamic extent and are automatically deallocated when they go out of scope.

Related link
Fortran standardization.

Java  A modern object-oriented language with a rich collection of useful features. The Java language started as an attempt by the Java group at Sun Microsystems to overcome software engineering problems introduced by C++. Key reasons for the language’s success were the security model and the portable execution environment, the Java Virtual Machine (JVM), which created a lot of interest for it as a platform for distributed computing on open networks.

Java is garbage-collected, as this facilitates object-oriented programming and is essential for security (which use after free would break). It had finalization from version 1.0 and three kinds of weakness from version 1.2 (confusingly, part of the Java 2 Platform).

Early JVMs had simple collectors that didn’t scale well for large programs, but the current crop is catching up to the state of the art.

See also:
**JavaScript**  JavaScript is a scripting language used by web browsers. The loose type system resembles other scripting languages, although the syntax follows C. There’s a prototype-based object system. Note that JavaScript is not related to Java in any way except name. There’s a standard by ECMA, known as ECMAScript.

Despite the C++-like syntax (with `new` and `delete` operators), JavaScript is *garbage-collected*.

---

**Lisp**

Lisp is a family of computer languages combining functional and procedural features with automatic memory management.

Lisp was invented by John McCarthy around 1958 for the manipulation of symbolic expressions. As part of the original implementation of Lisp, he invented *garbage collection*. He noted:

> This process, because it is entirely automatic, is more convenient for the programmer than a system in which he has to keep track of lists and erase unwanted lists.

Modern Lisp implementations, such as LispWorks and Allegro CL, have advanced *garbage collectors*.

Lisp is now used for all kinds of symbolic programming and other advanced software development. Major dialects today are Emacs Lisp, Common Lisp and Scheme. Most modern dialects and related languages, such as Dylan, are object-oriented.

See also:

*cons (1)*

---

**Related publications**


---

**Related link**

Common Lisp HyperSpec.

---

**Lisp Machine**  Of particular interest in the history of memory management are the *Lisp Machines*, early workstation computers built around a custom processor designed to improve the execution speed of Lisp by implementing primitive Lisp operations in microcode. The Lisp Machine *garbage collector* is a generalization of the algorithm described in *Baker (1978)* and used a technique similar to that described in *Ungar (1984)*, but utilizing hardware to improve performance.

A description of the garbage collector of one particular model is in *Moon (1984)*. The features important for its performance were:

1. Hardware support for data typing using *tags*;
2. Reference-based *read barriers* for *incremental* collecting;
3. *Write barriers* for *remembered sets* and *generational* collecting;
4. A tight integration with the *virtual memory* system.

The remembered sets were based on a *BIBOP* division of the virtual *address space*. The Lisp Machine *page table*, unlike virtually all modern virtual memory systems, was a flat, hash-based table (sometimes called an *inverted page table*), and thus insensitive to sparsely-populated virtual address spaces associated with BIBOP schemes.

---

6.7. *Introduction to memory management*  361
These custom processors eventually lost out to rapidly advancing stock hardware. Many of the techniques pioneered on Lisp Machines are used in today’s implementations, at a cost of a few more cycles.

**Related links**

---

**Lua**
Lua is a dynamically typed language created by Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes in 1993. The language supports object-oriented and functional styles of programming, and is designed to be easily embedded in a larger programming system as an extension or scripting language.

Lua uses automatic memory management and comes with a non-moving incremental garbage collector supporting soft real time applications. This uses a software barrier in order to be highly portable.

The language supports weak references in the form of weak (hash) tables, which have the unusual feature that their keys and values can be dynamically switched from being strong references to weak references, and vice versa (by assigning to the ___mode field of the table’s metatable). It also supports finalization (by assigning the ___gc field of the object’s metatable).

**Related links**
Lua, Garbage Collection.

---

**ML**
ML is a family of strongly-typed functional languages, of which the principal members are Standard ML and Caml.

Like other functional languages, ML provides automatic memory management. Modern ML implementations usually have advanced garbage collectors. The combination of clean functional semantics and strong typing allows advanced techniques, such as region inference.

The Standard ML of New Jersey (SML/NJ) system, which implements a slight variant of Standard ML, has been important to memory management research for three reasons. Firstly, the source code is publicly available and widely ported, allowing experimentation with both the collector and mutator. Secondly, the compiler generates code that does not use a control stack, but allocates function activation records on the heap instead. This means that the allocation rate is very high (up to one byte per instruction), and also that the collector has a very small root set. Thirdly, it uses a simple copying collector that is easy to modify.

**See also:**
immutable.

**Related publications**

---

**Modula-3**
An object-oriented descendant of Pascal.

Modula-3 is mostly garbage-collected, although it is possible to use manual memory management in certain modules.

**Related links**
modula3.org, Modula-3 language definition.
Pascal  An imperative language characterized by block structure and a relatively strong (for its time) static type system. Pascal was designed by Niklaus Wirth around 1970.

Pascal was popular as a teaching language due to its small size, but it lacked many features needed for applications programming. Now it’s been largely supplanted by its more feature-rich descendants Modula-2, Modula-3, and Oberon, mainly surviving in the popular Delphi development tool.

Pascal uses manual memory management (with the operators NEW and DISPOSE). The descendants mentioned all offer automatic memory management.

Related links
Borland Delphi Home Page, Pascal standardization.

Perl  Perl is a complex but powerful language that is an eclectic mixture of scripting languages and programming languages.

Perl programmers can work with strings, arrays, and associative arrays without having to worry about manual memory management. Perl is well-suited to complex text file manipulation, such as report generation, file format conversion, and web server CGI scripts. It is also useful for rapid prototyping, but large Perl scripts are often unmaintainable.

Perl’s memory management is well-hidden, but is based on reference counts and garbage collection. It also has mortal variables, whose lifetimes are limited to the current context. It is possible to free the memory assigned to variables (including arrays) explicitly, by undef-ing the only reference to them.

Related link
The Perl Programming Language.

PostScript  The PostScript language is an interpretive language with powerful graphics features, widely used as a page description language for printers and typesetters.

The Level 1 PostScript language has a simple stack-like memory management model, using save and restore operators to recycle memory. The Level 2 PostScript language adds garbage collection to this model.

See also:
VM, composite object, simple object.

Related link
Harlequin RIP.

Prolog  A logic programming language invented by Alain Colmerauer around 1970, Prolog is popular in the AI and symbolic computation community. It is special because it deals directly with relationships and inference rather than functions or commands.

Storage is usually managed using a garbage collector, but the complex control flow places special requirements on the collector.

Related links
Prolog Standardization, Prolog Memory Management - Garbage Collection.

Python  Python is a “duck-typed” object-oriented language created in the early 1990s by Guido van Rossum.

There are several implementations running on a variety of virtual machines: the original “CPython” implementation runs on its own virtual machine; IronPython runs on the Common Language Runtime; Jython on the Java Virtual Machine.
CPython manages memory using a mixture of **reference counting** and **non-moving mark-and-sweep garbage collection**. Reference counting ensures prompt deletion of objects when their reference count falls to zero, while the garbage collector reclaims **cyclic data structures**.

The language supports **finalization** (classes may have a `__del__` method, which is run just before the object is destroyed), and **weak references** (via the `weakref` module).

**Related links**

Python, Garbage Collector interface, `__del__` method, weakref module.

---

**Scheme**  A small functional language blending influences from **Lisp** and **Algol**.

Key features of Scheme include symbol and list operations, **heap allocation** and **garbage collection**, lexical scoping with first-class function objects (implying **closures**), reliable tail-call elimination (allowing iterative procedures to be described tail-recursively), the ability to dynamically obtain the current **continuation** as a first-class object, and a language description that includes a formal semantics.

Scheme has been gaining popularity as an extension language; Project GNU’s extension package of choice, **Guile**, is a Scheme interpreter. **Garbage collection** is an important part of the ease of use that is expected from an extension language.

**Related links**

Scheme Standards documents, Scheme Requests for Implementation.

---

**Simula**  Simula was designed as a language for simulation, but it expanded into a full general-purpose programming language and the first object-oriented language.

Simula I, designed in 1962–64 by Kristen Nygaard and Ole-Johan Dahl, was based on **ALGOL** 60, but the stack allocation discipline was replaced by a two-dimensional **free list**.

It was Simula 67 that pioneered classes and inheritance to express behavior. This domain-oriented design was supported by **garbage collection**.

**Related publication**


---

**Smalltalk**  Smalltalk is an object-oriented language with single inheritance and message-passing.

**Automatic memory management** is an essential part of the Smalltalk philosophy. Many important techniques were first developed or implemented for Smalltalk.

**Related publications**


**Related link**

Smalltalk standardization.

---

### 6.8 Home

Welcome to the Memory Management Reference! This is a resource for programmers and computer scientists interested in **memory management** and **garbage collection**.
Memory Management Glossary
A glossary of more than 500 memory management terms, from absolute address to zero count table.

Introduction to memory management
Articles giving a beginner’s overview of memory management.

Bibliography
Books and research papers related to memory management.

Frequently Asked Questions
Frequently asked questions about memory management.

The Memory Management Reference is maintained by Ravenbrook Limited. We also maintain the Memory Pool System (an open-source, thread-safe, incremental garbage collector), and we are happy to provide advanced memory management solutions to language and application developers through our consulting service.

6.9 Frequently Asked Questions

This is a list of questions that represent the problems people often have with memory management. Some answers appear below, with links to helpful supporting material, such as the Memory Management Glossary, the Bibliography, and external sites. For a full explanation of any terms used, see the glossary.

6.9.1 C-specific questions

Can I use garbage collection in C?

Yes. Various conservative garbage collectors for C exist as add-on libraries.

Related links

Why do I need to test the return value from malloc? Surely it always succeeds?

For small programs, and during light testing, it is true that malloc usually succeeds. Unfortunately, there are all sorts of unpredictable reasons why malloc might fail one day; for example:

• someone uses your program for a far larger data set than you anticipated;
• your program is running on a machine with less memory than you expected;
• the machine your program is running on is heavily loaded.

In this case, malloc will return NULL, and your program will attempt to store data by resolving the null pointer. This might cause your program to exit immediately with a helpful message, but it is more likely to provoke mysterious problems later on.
If you want your code to be robust, and to stand the test of time, you must check all error or status codes that may be returned by functions you call, especially those in other libraries, such as the C run-time library.

If you really don’t want to check the return value from `malloc`, and you don’t want your program to behave mysteriously when out of memory, wrap `malloc` in something like this:

```c
#include <stdio.h>
#include <stdlib.h>

void *my_malloc(size_t size)
{
    void *p = malloc(size);

    if (p == NULL) {
        fputs("Out of memory.
", stderr);
        exit(EXIT_FAILURE);
    }

    return p;
}
```

Undefined behavior is worth eliminating even in small programs.

What’s the point of having a garbage collector? Why not use malloc and free?

*Manual memory management*, such as `malloc` and `free` (2), forces the programmer to keep track of which memory is still required, and who is responsible for freeing it. This works for small programs without internal interfaces, but becomes a rich source of bugs in larger programs, and is a serious problem for interface abstraction.

*Automatic memory management* frees the programmer from these concerns, making it easier for him to code in the language of his problem, rather than the tedious details of the implementation.

See also:

*garbage collection*

What’s wrong with ANSI malloc in the C library?

The `malloc` function provides a very basic *manual memory management* service. However, it does not provide the following things, which may be desirable in your memory manager:

- high performance for specified block sizes;
- *tagged references*;
- simultaneous frees;
- *locality of reference* hints;
- *formatted objects*;
- garbage collection;
- deallocation of partial blocks;
- multi-threading without synchronization;
- inlined allocation code;
- *finalization*.

Many of these can be added on top of `malloc`, but not with full performance.
6.9.2 C++-specific questions

Can I use garbage collection in C++?

Yes. The C++ specification has always permitted garbage collection. Bjarne Stroustrup (C++’s designer) has proposed that this be made explicit in the standard. There exist various conservative and semi-conservative garbage collectors for C++.

See also:

C++, conservative garbage collection, semi-conservative garbage collection.

Related link

Why is delete so slow?

Often delete must perform a more complex task than simply freeing the memory associated with an object; this is known as finalization. Finalization typically involves releasing any resources indirectly associated with the object, such as files that must be closed or ancillary objects that must be finalized themselves. This may involve traversing memory that has been unused for some time and hence is paged out.

With manual memory management (such as new and delete), it is perfectly possible for the deallocation operation to vary in complexity. Some systems do quite a lot of processing on freed blocks to coalesce adjacent blocks, sort free blocks by size (in a buddy system, say), or sort the free list by address. In the last case, deallocating blocks in address order (or sometimes reverse address order) can result in poor performance.

What happens if you use class libraries that leak memory?

In C++, it may be that class libraries expect you to call delete on objects they create, to invoke the destructor (2). Check the interface documentation.

Failing this, if there is a genuine memory leak in a class library for which you don’t have the source, then the only thing you can try is to add a garbage collector.

Related link

Can’t I get all the benefits of garbage collection using C++ constructors and destructors?

Carefully designed C++ constructors (2) and destructors (2) can go a long way towards easing the pain of manual memory management. Objects can know how to deallocate all their associated resources, including dependent objects (by recursive destruction). This means that clients of a class library do not need to worry about how to free resources allocated on their behalf.

Unfortunately, they still need to worry about when to free such resources. Unless all objects are allocated for precisely one purpose, and referred to from just one place (or from within one compound data structure that will be destroyed atomically), then a piece of code that has finished with an object cannot determine that it is safe to call the destructor; it cannot be certain (especially when working with other people’s code) that there is not another piece of code that will try to use the object subsequently.
This is where garbage collection has the advantage, because it can determine when a given object is no longer of interest to anyone (or at least when there are no more references to it). This neatly avoids the problems of having multiple copies of the same data or complex conditional destruction. The program can construct objects and store references to them anywhere it finds convenient; the garbage collector will deal with all the problems of data sharing.

### 6.9.3 Common objections to garbage collection

**What languages use garbage collection?**

*Java, C#, Python, Lisp, ML, …* the list goes on. It surprises many to learn that many implementations of *BASIC* use garbage collection to manage character strings efficiently.

*C++* is sometimes characterized as the last holdout against garbage collection, but this is not accurate. See *Can I use garbage collection in C++?*

The notion of *automatic memory management* has stood the test of time and is becoming a standard part of modern programming environments. Some will say “the right tool for the right job”, rejecting automatic memory management in some cases; few today are bold enough to suggest that there is never a place for garbage collection among tools of the modern programmer—either as part of a language or as an add-on component.

**What’s the advantage of garbage collection?**

*Garbage collection* frees you from having to keep track of which part of your program is responsible for the deallocation of which memory. This freedom from tedious and error-prone bookkeeping allows you to concentrate on the problem you are trying to solve, without introducing additional problems of implementation.

This is particularly important in large-scale or highly modular programs, especially libraries, because the problems of manual memory management often dominate interface complexity. Additionally, garbage collection can reduce the amount of memory used because the interface problems of manual memory management are often solved by creating extra copies of data.

In terms of performance, garbage collection is often faster than manual memory management. It can also improve performance indirectly, by increasing *locality of reference* and hence reducing the size of the *working set*, and decreasing *paging*.

**Related publication**


---

**Programs with GC are huge and bloated; GC isn’t suitable for small programs or systems**

While it is true that the major advantages of *garbage collection* are only seen in complex systems, there is no reason for garbage collection to introduce any significant overhead at any scale. The data structures associated with garbage collection compare favorably in size with those required for *manual memory management*.

Some older systems gave garbage collection a bad name in terms of space or time overhead, but many modern techniques exist that make such overheads a thing of the past. Additionally, some garbage collectors are designed to work best in certain problem domains, such as large programs; these may perform poorly outside their target environment.

**Related publication**

I can’t use GC because I can’t afford to have my program pause

While early garbage collectors had to complete without interruption and hence would pause observably, many techniques are now available to ensure that modern collectors can be unobtrusive.

See also:
incremental garbage collection, parallel garbage collection.

Isn’t it much cheaper to use reference counts rather than garbage collection?

No, updating reference counts is quite expensive, and they have a couple of problems:

- They can’t cope with cyclic data structures; that is, sets of objects that are referred to only by objects in that set, but that don’t have a zero reference count.
- Reference counting gets more expensive if you have to allow for the count overflowing.

There are many systems that use reference counts, and avoid the problems described above by using a conventional garbage collector to complement it. This is usually done for real-time benefits. Unfortunately, experience shows that this is generally less efficient than implementing a proper real-time garbage collector, except in the case where most reference counts are one.

Related publication

Isn’t GC unreliable? I’ve heard that GCs often kill the program

Garbage collectors usually have to manipulate vulnerable data structures and must often use poorly-documented, low-level interfaces. Additionally, any garbage collection problems may not be detected until some time later. These factors combine to make most garbage collection bugs severe in effect, hard to reproduce, and difficult to work around.

On the other hand, commercial garbage collection code will generally be heavily tested and widely used, which implies it must be reliable. It will be hard to match that reliability in a manual memory manager written for one program, especially given that manual memory management doesn’t scale as well as the automatic variety.

In addition, bugs in the compiler or run-time (or application if the language is as low-level as C) can corrupt the heap in ways that only the garbage collector will detect later. The collector is blamed because it found the corruption. This is a classic case of shooting the messenger.

I’ve heard that GC uses twice as much memory

This may be true of primitive collectors (like the two-space collector), but this is not generally true of garbage collection. The data structures used for garbage collection need be no larger than those for manual memory management.

Doesn’t garbage collection make programs slow?

No. Benjamin Zorn (1992) found that:

the CPU overhead of conservative garbage collection is comparable to that of explicit storage management techniques. […] Conservative garbage collection performs faster than some explicit algorithms and slower than others, the relative performance being largely dependent on the program.

Note also that the version of the conservative collector used in this paper is now rather old and the collector has been much improved since then.
Manual memory management gives me control—it doesn’t pause

It is possible for manual memory management to pause for considerable periods, either on allocation or deallocation. It certainly gives no guarantees about performance, in general.

With automatic memory management, such as garbage collection, modern techniques can give guarantees about interactive pause times, and so on.

See also:
incremental garbage collection, parallel garbage collection.

6.9.4 Miscellaneous

Why does my disk rattle so much?

When you are using a virtual memory system, the computer may have to fetch pages of memory from disk before they can be accessed. If the total working set of your active programs exceeds the physical memory available, paging will happen continually, your disk will rattle, and performance will degrade significantly. The only solutions are to install more physical memory, run fewer programs at the same time, or tune the memory requirements of your programs.

The problem is aggravated because virtual memory systems approximate the theoretical working set with the set of pages on which the working set lies. If the actual working set is spread out onto a large number of pages, then the working page-set is large.

When objects that refer to each other are distant in memory, this is known as poor locality of reference. This happens either because the program’s designer did not worry about this, or the memory manager used in the program doesn’t permit the designer to do anything about it.

Note that copying garbage collection can dynamically organize your data according to the program’s reference patterns and thus mitigate this problem.

See also:
thrash

Related publication
Denning (1968).

Where can I find out more about garbage collection?

Many modern languages have garbage collection built in, and the language documentation should give details. For some other languages, garbage collection can be added, for example via the Memory Pool System, or the Boehm–Demers–Weiser collector.

See also:
garbage collection

Related publications

Related link
Memory Pool System, Boehm–Demers–Weiser collector, GC-LIST FAQ.
Where can I get a garbage collector?

The Memory Pool System and the Boehm–Demers–Weiser collector are suitable for C or C++. The best way to get a garbage collector, however, is to program in a language that provides garbage collection natively.

See also:

garbage collection

Related link


Why does my program use so much memory?

If you are using manual memory management (for example, malloc and free (2) in C), it is likely that your program is failing to free memory blocks after it stops using them. When your code allocates memory on the heap, there is an implied responsibility to free that memory. If a function uses heap memory for returning data, you must decide who takes on that responsibility. Pay special attention to the interfaces between functions and modules. Remember to check what happens to allocated memory in the event of an error or an exception.

If you are using automatic memory management (almost certainly garbage collection), it is probable that your code is remembering some blocks that it will never use in future. This is known as the difference between liveness and reachability. Consider clearing variables that refer to large blocks or networks of blocks, when the data structure is no longer required.

I use a library, and my program grows every time I call it. Why?

If you are using manual memory management, it is likely that the library is allocating data structures on the heap every time it is used, but that they are not being freed. Check the interface documentation for the library; it may expect you to take some action when you have finished with returned data. It may be necessary to close down the library and re-initialize it to recover allocated memory.

Unfortunately, it is all too possible that the library has a memory management bug. In this case, unless you have the source code, there is little you can do except report the problem to the supplier. It may be possible to add a garbage collector to your language, and this might solve your problems.

With a garbage collector, sometimes objects are retained because there is a reference to them from some global data structure. Although the library might not make any further use of the objects, the collector must retain the objects because they are still reachable.

If you know that a particular reference will never be used in future, it can be worthwhile to overwrite it. This means that the collector will not retain the referred object because of that reference. Other references to the same object will keep it alive, so your program doesn’t need to determine whether the object itself will ever be accessed in future. This should be done judiciously, using the garbage collector’s tools to find what objects are being retained and why.

If your garbage collector is generational, it is possible that you are suffering from premature tenuring, which can often be solved by tuning the collector or using a separate memory area for the library.

Should I write my own memory allocator to make my program fast?

If you are sure that your program is spending a large proportion of its time in memory management, and you know what you’re doing, then it is certainly possible to improve performance by writing a suballocator. On the other hand,
advances in memory management technology make it hard to keep up with software written by experts. In general, improvements to memory management don’t make as much difference to performance as improvements to the program algorithms.

*Benjamin Zorn (1992)* found that:

> In four of the programs investigated, the programmer felt compelled to avoid using the general-purpose storage allocator by writing type-specific allocation routines for the most common object types in the program. [...] The general conclusion [...] is that programmer optimizations in these programs were mostly unnecessary. [...] simply using a different algorithm appears to improve the performance even more.

and concluded:

> programmers, instead of spending time writing domain-specific storage allocators, should consider using other publicly-available implementations of storage management algorithms if the one they are using performs poorly.

**Why can’t I just use local data on the stack or in global variables?**

Global, or static, data is fixed size; it cannot grow in response to the size or complexity of the data set received by a program. Stack-allocated data doesn’t exist once you leave the function (or program block) in which it was declared.

If your program’s memory requirements are entirely predictable and fixed at compile-time, or you can structure your program to rely on stack data only while it exists, then you can entirely avoid using heap allocation. Note that, with some compilers, use of large global memory blocks can bloat the object file size.

It may often seem simpler to allocate a global block that seems “probably large enough” for any plausible data set, but this simplification will almost certainly cause trouble sooner or later.

See also: *stack allocation, heap allocation, static allocation.*

**Why should I worry about virtual memory? Can’t I just use as much memory as I want?**

While *virtual memory* can greatly increase your capacity to store data, there are three problems typically experienced with it:

- It does not provide an unlimited amount of memory. In particular, all memory that you actually allocate (as opposed to reserve) has to be stored somewhere. Usually you must have disk space available for all pages containing allocated memory. In a few systems, you can subtract the available physical memory from the disk space required. If the memory contains images of program or data files, then *file mapping*, or assigning existing files to regions of the virtual address space, can help considerably.

- In most computers, there is a large difference in speed between main memory and disk; running a program with a *working set* that does not fit in physical memory almost always results in unacceptable performance.

- An additional problem with using unnecessary quantities of memory is that poor *locality of reference* can result in heavy paging.

See also: *thrash.*
6.10 Copyright

6.10.1 Use subject to copyright restrictions

The copyright in The Memory Management Reference is owned by Ravenbrook Limited.

Permission to copy part or all of The Memory Management Reference for personal or classroom use is granted without fee, provided that copies are not made or distributed for profit or commercial advantage; that the copyright notice, the title of the publication, and its date appear; and that notice is given that copying is by permission of Ravenbrook Limited. To copy otherwise, to republish, to post on servers, or to redistribute to lists requires prior specific permission.

6.10.2 Warranty disclaimer

The Memory Management Reference is provided “as is” without warranty of any kind, express or implied, including, but not limited to, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement.

6.11 Acknowledgements

The Memory Management Reference is maintained by Ravenbrook Limited.

Most of it was originally written by memory management experts in the Adaptive Memory Management Group at Harlequin Limited:

• Nick Barnes
• Richard Brooksby
• David Jones
• David Lovemore
• Tony Mann
• Gavin Matthews
• Pekka P. Pirinen
• Richard Tucker
• P. T. Withington

Gavin Matthews was the original editor of the Memory Management Reference.

The Adaptive Memory Management Group no longer exists, and Harlequin has become a part of Global Graphics. However, most of the group’s work has been acquired by Ravenbrook Limited, whose directors are Richard Brooksby, the group’s chief architect and manager, and Nick Barnes, a senior group member.

Particular thanks go to Richard Jones for his enormously useful book The Garbage Collection Handbook, and for his comments on the Reference.

Many people have made valuable suggestions for the Reference and contributed glossary entries. We are grateful for the help of:

• Judy Anderson
• Giuseppe Attardi
• Daniel Barrett
• Leah Bateman
• Stephen Bevan
• Hans Boehm
• Hans Feldt
• Lars Hansen
• Anthony Hosking
• Paul Jackson
• Mark S. Johnstone
• Richard Kistruck
• Stavros Macrakis
• Gareth McCaughan
• David A. Moon
• Eliot Moss
• John S. Pieper
• Kent Pitman
• Andrew Shires
• Walter Spector
• Martin Simmons
• David Stoutamire
• Tom Thomson
• Mark Tillotson
• JonL White
• David S. Wise
• Benjamin Zorn
Symbols

-d
  mpseventsql command line option, 125
-f
  mpseventsql command line option, 125
-f <filename>
  mpseventcnv command line option, 124
-h
  mpseventcnv command line option, 124
  mpseventtxt command line option, 124
-i <filename>
  mpseventsql command line option, 125
-l <filename>
  mpseventtxt command line option, 124
-o <filename>
  mpseventtxt command line option, 124
-p
  mpseventsql command line option, 125
-r
  mpseventsql command line option, 125
-t
  mpseventsql command line option, 125
-v
  mpseventsql command line option, 125

A
abort, 24
absolute address, 229
activation frame, 229
activation record, 229
activation stack, 229
active, 230
address, 230
address space, 230
address space layout randomization, 24, 230
address translation cache, 230
address-ordered first fit, 231
aging space, 231
algebraic data type, 231
ALGOL, 357
alignment, 9, 231
  object, 9
alive, 232
allocate, 232
allocation, 66
  frame, 118
  manual, 66
  pattern, 117
  ramp pattern, 117
  segregated allocation cache, 115
  segregated-fit, 112
  stack-like, 118
  tutorial, 19
allocation frame, 233
allocation mechanism, 233
allocation pattern, 233
allocation point, 67, 233
allocation point protocol, 68, 233
  bugs, 71
  cautions, 70
  example, 70
  tutorial, 19
allocation points
  implementation, 72
allocation policy, 233
  temporal fit, 172
allocation strategy, 234
allocator, 234
ambiguous reference, 234
  scanning, 82
ambiguous root, 234
AMC, 156
  interface, 157
  introspection, 157
  properties, 156
AMCZ, 158
  interface, 159
  introspection, 159
  properties, 158
AMS, 159
  interface, 160
properties, 159
arena, 55, 234
clamped state, 61
creating, 8
introspection, 64
multiple, 56
parked state, 61
properties, 58
states, 61
unclamped state, 61
arena class, 235
choosing, 8
client, 56
virtual memory, 57
arguments
keyword, 48
ASLR, 24, 235
assertion, 53, 235
common causes, 53
asynchronous garbage collector, 235
asynchrony
cautions, 20
ATC, 235
atomic object, 235
atomic updates, 21
automatic memory management, 235
automatic storage duration, 236
AWL, 161
cautions, 164
dependent object, 162
interface, 164
properties, 161
protection faults, 163

B
backing store, 236
barrier
handling in debugger, 24
barrier (1), 236
barrier (2), 237
barrier hit, 237
base pointer, 237
BASIC, 357
best fit, 237
BIBOP, 238
big bag of pages, 238
binary buddies, 238
bit array, 238
bit table, 238
bit vector, 239
bitmap, 239
bitmap marking, 239
bitmapped fit, 239
bitmask, 239
bitset, 239
black, 239
black-listing, 240
blacklisting, 240
block, 240
bounds error, 240
boxed, 240
break-table, 240
brk, 240
broken heart, 241
bucket, 241
buddy system, 241
buffer, 241
bug
allocating with wrong size, 27
allocation point protocol, 71
underscanning, 25
building, 2, 341
bus error, 241
byte (1), 242
byte (2), 242
byte (3), 242
byte (4), 242

C
C, 357
C++, 358
C89, 242
C90, 242
C99, 243
C#, 358
cache (1), 243
cache (2), 243
cache memory, 243
cache policy, 243
caching (3), 244
cactus stack, 244
card, 244
card marking, 244
cautions
allocation point protocol, 70
asynchrony, 20
AWL, 164
finalization, 105
object format, 76
root, 89
cell, 245
chain
generation, 96
Cheney collector, 245
Cheney scan, 245
clamped state, 61, 245
client arena, 245
client arena class, 56
client object, 245
client pointer, 245
client program, 246
closure, 246
coalesce, 246
COBOL, 358
code
  platform, 134
collect, 246
collection, 96, 246
    running, 62
    scheduling, 97
collection cycle, 246
collector (1), 247
collector (2), 247
color, 247
colour, 247
commit limit, 247
committed (1), 247
committed (2), 247
Common Lisp, 358
compactifying, 247
compaction, 248
compiling, 2, 341
composite object, 248
comprehensive, 248
concurrent garbage collection, 248
condemned set, 248
CONFIG_PLINTH_NONE (C macro), 129
CONFIG_VAR_COOL (C macro), 55
CONFIG_VAR_HOT (C macro), 55
CONFIG_VAR_RASH (C macro), 55
connected, 249
cons (1), 249
cons (2), 249
conservative garbage collection, 249
consistency
  maintaining, 20
constant root, 249
constructor (1), 249
constructor (2), 250
continuation, 250
control stack, 250
cool, 250
cold variety, 24, 55
copying garbage collection, 250
copyright, 340
core, 251
creation space, 251
critical path, 251
  scanning, 82
crossing map, 251
customizing
telemetry, 127

cyclic data structure, 252

D
dangling pointer, 252
data stack, 252
dead, 252
deallocate, 252
debugger, 24
debugging, 23
  fencepost, 119
  free space splatting, 120
  pool, 119
debugging pool, 252
delayed coalescing, 253
delayed reference counting, 253
dependency
  location, 107
dependent object, 253
  AWL, 162
deprecated interfaces, 142
    in version 1.111, 149
    in version 1.112, 144
    in version 1.113, 143
    in version 1.115, 142
derived pointer, 253
derived type, 253
destroying, 21
destructor (1), 253
destructor (2), 254
DGC, 254
direct method, 254
dirty bit, 254
distributed garbage collection, 254
double buddies, 254
double free, 255
double word, 255
doubly weak hash table, 255
DRAM, 255
Dylan, 359
dynamic allocation, 255
dynamic extent, 255
dynamic memory, 256
dynamic RAM, 256

E
ecru, 256
edge, 256
Emacs Lisp, 360
entry table (1), 256
entry table (2), 257
environment variable
  MPS_TELEMETRY_CONTROL, 121, 123, 133, 347, 348
  MPS_TELEMETRY_DATABASE, 124, 125
H
handle, 269
hash table
  address-based example, 34
  weak example, 37
header, 270
headers
  interface, 45
  object format, 76
heap, 270
heap allocation, 270
hit, 270
hit rate, 270
hosted, 270
hosted environment, 128
hot, 271
hot variety, 55
huge page, 271
I
I/O module, 129
identifiers
  interface, 46
idle time
  using for garbage collection, 63
immediate data, 271
immune set, 271
immutable, 271
immutable object, 272
in parameter, 272
in-band header, 272
in-band headers
  object format, 76
in/out parameter, 272
incremental garbage collection, 272
incremental update, 272
indefinite extent, 274
indexed fit, 274
indirect method, 274
infant mortality, 274
inline allocation (1), 274
inline allocation (2), 275
installing, 2, 341
inter-generational pointer, 275
interface
  functions, 46
  headers, 45
  identifiers, 46
  introduction, 45
  language, 45
  macros, 47
  support policy, 45
  type punning, 47
  types, 46
internal pointer, 275
internal fragmentation, 275
introspection
  AMC, 157
  AMCZ, 159
  arena, 64
  object format, 79
  pool, 66
  root, 95
invalid page fault, 275
inverted page table, 276
inverted page-table, 276
is-forwarded method, 13, 276
J
Java, 360
JavaScript, 361
K
kB, 276
keyword
  arguments, 48
  keyword argument, 276
  kilobyte, 276
L
labels
  telemetry, 126
language
  interface, 45
large object area, 277
large page, 277
leaf object, 277
leak, 278
library module, 130
license, 340
  commercial terms, 1
life, 278
lifetime, 278
load, 279
locality of reference, 279
LO, 165
  interface, 166
  properties, 166
load, 279
location dependency, 107, 280
  adding, 108
  creating, 108
example, 34
interface, 110
testing staleness, 109
thread safety, 110
lock free, 280
logical address, 280
longword, 280
Lua, 362

M
machine word, 280
macros
interface, 47
main memory, 280
malloc, 281
implementing, 43
manual allocation, 66
manual memory management, 281
mapped, 282
mapping, 282
mark-and-sweep, 283
mark-compact, 282
mark-sweep, 283
marking, 283
MB, 283
megabyte, 283
memoization, 283
memory (1), 284
memory (2), 284
memory (3), 284
memory (4), 284
memory bandwidth, 284
memory cache, 284
memory hierarchy, 284
memory leak, 284
memory location, 285
memory management, 285
Memory Management Unit, 285
memory manager, 286
memory mapping, 286
Memory Pool System
overview, 1
performance, 29
supported target platforms, 1
technical introduction, 1
tutorial, 6
memory protection, 286
message, 286
example, 100
finalization, 100, 107
garbage collection, 98
garbage collection start, 98
interface, 102
introduction, 100
queue interface, 103
types, 101
message queue, 286
message type, 287
MFS, 167
interface, 168
properties, 167
misaligned, 287
miss, 287
miss rate, 287
ML, 362
mmap, 287
MMU, 287
mode
root, 90
Modula-3, 362
mostly-copying garbage collection, 288
mostly-exact garbage collection, 288
mostly-precise garbage collection, 288
moving garbage collector, 288
moving memory manager, 288
mps_addr_fmt (C function), 79
mps_addr_pool (C function), 66
mps_addr_t (C type), 48
mps_align_t (C type), 48
mps_alloc (C function), 67
mps_alloc_pattern_ramp (C function), 118
mps_alloc_pattern_ramp_collect_all (C function), 118
mps_alloc_pattern_t (C type), 117
mps_amc_apply (C function), 158
mps_amc_apply_stepper_t (C type), 158
mps_ap_alloc_pattern_begin (C function), 117
mps_ap_alloc_pattern_end (C function), 117
mps_ap_alloc_pattern_reset (C function), 117
mps_ap_create (C function), 147
mps_ap_create_k (C function), 67
mps_ap_create_v (C function), 147
mps_ap_destroy (C function), 67
mps_ap_fill (C function), 74
mps_ap_frame_pop (C function), 119
mps_ap_frame_push (C function), 119
mps_ap_s (C type), 74
mps_ap_t (C type), 67
mps_ap_trip (C function), 74
MPS_ARCH_I3 (C macro), 134
MPS_ARCH_I6 (C macro), 134
mps_arena_clamp (C function), 62
mps_arena_class_cl (C function), 57
mps_arena_class_t (C type), 56
mps_arena_class_vm (C function), 57
mps_arena_collect (C function), 62
mps_arena_commit_limit (C function), 58
mps_arena_commit_limit_set (C function), 59
mps_arena_committed (C function), 59

380 Index
mps_lib_memcpy (C function), 133
mps_lib_memset (C function), 133
mps_message_clock (C function), 102
mps_message_discard (C function), 103
mps_message_finalization_ref (C function), 107
mps_message_gc_condemned_size (C function), 99
mps_message_gc_live_size (C function), 99
mps_message_gc_not_condemned_size (C function), 99
mps_message_gc_start_why (C function), 98
mps_message_get (C function), 103
mps_message_poll (C function), 103
mps_message_queue_type (C function), 104
mps_message_t (C type), 102
mps_message_type (C function), 103
mps_message_type_disable (C function), 101
mps_message_type_enable (C function), 102
mps_message_type_finalization (C function), 107
mps_message_type_gc (C function), 99
mps_message_type_gc_start (C function), 98
mps_message_type_t (C type), 101
mps_mv_free_size (C function), 142
mps_mv_size (C function), 142
mps_mvff_free_size (C function), 143
mps_mvt_free_size (C function), 143
mps_mvt_size (C function), 143
MPS_OS_FR (C macro), 135
MPS_OS_LI (C macro), 135
MPS_OS_W3 (C macro), 135
MPS_OS_XC (C macro), 135
MPS_PF_ALIGN (C macro), 135
MPS_PF_FRI3GC (C macro), 135
MPS_PF_FRI6GC (C macro), 135
MPS_PF_LII3GC (C macro), 135
MPS_PF_LII6GC (C macro), 135
MPS_PF_LII6LL (C macro), 135
MPS_PF_STRING (C macro), 135
MPS_PF_W3I3MV (C macro), 135
MPS_PF_W3I6MV (C macro), 135
MPS_PF_W3I16LL (C macro), 135
MPS_PF_XCI3GC (C macro), 135
MPS_PF_XCI3LL (C macro), 135
MPS_PF_XCI6LL (C macro), 135
mps_pool_check_fenceposts (C function), 121
mps_pool_check_free_space (C function), 121
mps_pool_class_t (C type), 65
mps_pool_class_t (C type), 65
mps_pool_create (C function), 144
mps_pool_create_k (C function), 65
mps_pool_create_v (C function), 147
mps_pool_debug_option_s (C type), 120
mps_pool_destroy (C function), 65
mps_pool_free_size (C function), 66
mps_pool_total_size (C function), 66
mps_rank_ambig (C function), 90
mps_rank_exact (C function), 90
mps_rank_t (C type), 90
mps_rank_weak (C function), 90
mps_reg_scan_t (C type), 93
MPS_RES_COMMIT_LIMIT (C macro), 52
MPS_RES.FAIL (C macro), 52
MPS_RES.IO (C macro), 52
MPS_RES_MEMORY (C macro), 52
MPS_RES_OK (C macro), 52
MPS_RES_PARAM (C macro), 52
MPS_RES RESOURCE (C macro), 52
mps_res_t (C type), 51
MPS_RES_UNIMPL (C macro), 53
mps_reserve (C function), 69
MPS_RESERVE_BLOCK (C function), 69
MPS_RM_CONST (C macro), 91
MPS_RM PROT (C macro), 91
MPS_RM PROT INNER (C macro), 91
mps_rm_t (C type), 91
mps_root_create (C function), 91
mps_root_create_fmt (C function), 92
mps_root_create_reg (C function), 92
mps_root_create_table (C function), 93
mps_root_create_table_masked (C function), 94
mps_root_destroy (C function), 95
mps_root_scan_t (C type), 92
mps_root_t (C type), 91
mps_roots_stepper_t (C type), 96
mps_sac_alloc (C function), 115
MPS_SAC_ALLOC_FAST (C function), 116
MPS_SAC_CLASS_LIMIT (C macro), 113
mps_sac_class_s (C type), 113
mps_sac_create (C function), 114
mps_sac_create (C function), 114
mps_sac_destroy (C function), 115
mps_sac_flush (C function), 115
mps_sac_free (C function), 116
MPS_SAC_FREE_FAST (C function), 116
mps_sac_t (C type), 113
MPS_SACALLOC_FAST (C function), 84
MPS_SCAN BEGIN (C function), 84
MPS_SCAN END (C function), 84
mps_stack_scan_ambig (C function), 93
mps_telemetry_flush (C function), 125
mps_telemetry_get (C function), 126
mps_telemetry_label (C function), 126
382 Index
Index

N
nailing, 289
natural alignment, 289
nepotism, 289
new space, 290
newspace, 290
next fit, 290
node, 290
non-moving garbage collector, 290
non-moving memory manager, 290
nursery generation, 290
nursery space, 290

O
object, 291
object format, 9, 291
cautions, 76
format method, 77
headers, 76
in-band headers, 76
interface, 75
introduction, 74
introspection, 79
scan method, 80
object pointer, 291
off-white, 291
old space, 292
oldspace, 292
one-bit reference count, 292
opaque type, 292
out parameter, 293
out-of-band header, 293
overcommit, 293
overwriting error, 293

P
padding, 293
padding method, 14, 293
padding object, 293
page, 294
page fault, 294
page marking, 294
page protection, 294
page table, 294
paged in, 294
paged out, 294
paging, 295
palimpsest, 295
parallel garbage collection, 295
parked state, 61, 296
Pascal, 363
pattern
  allocation, 117
pause
  limiting, 63
perfect fit, 296
performance, 29
Perl, 363
phantom reachable, 296
phantom reference, 296
phantomly reachable, 296
physical address, 296
physical address space, 297
physical memory (1), 297
physical memory (2), 297
physical storage, 297
pig in the python, 297
pig in the snake, 298
pinning, 298
placement policy, 298
platform, 298
  code, 134
  historical codes, 136
  historical list, 137
  interface, 134
platforms
  supported, 1
plinth, 298
  I/O module, 129
  introduction, 128
  library module, 130
pointer, 298
  creating, 15, 64
  debugging, 119
  introspection, 66
pool, 299
  creating, 15, 64
  debugging, 119
  introspection, 66
pool class, 65, 299
  AMC, 156
  AMCZ, 158
  AMS, 159
  AWL, 161
  choosing, 9, 153
  LO, 165
  MFS, 167
  MV, 168
  MVFF, 169
  MVT, 171
  SNC, 174
  table of properties, 154
  writing, 156
porting
  plinth, 128
PostScript, 363
precise garbage collection, 299
precise reference, 299
precise root, 299
premature free, 299
premature promotion, 299
premature tenuring, 300
primary storage, 300
Prolog, 363
promotion, 300
protectable root, 300
protection, 300
protection exception, 301
protection fault, 301
protection faults
  AWL, 163
protection violation, 301
punning
  type, 47
Python, 363
Q
quadword, 301
R
RAM, 302
  ramp allocation, 117, 302
  random access memory, 302
rank, 302
  root, 90
rash, 302
rash variety, 55
Ravenbrook Limited, 1
raw, 302
reachable, 302
read barrier, 303
read fault, 303
read-only memory, 303
real memory (1), 304
real memory (2), 304
reclaim, 304
recycle, 304
reference, 304
  reference counting, 305
  reference object, 305
  region inference, 306
  register, 306
  register set partitioning, 306
  relocation, 307
  remembered set, 307
  remote reference, 307
  replicating garbage collector, 307
  reserved, 307
  resident, 308
  resident set, 308
  result code, 51, 308
resurrection, 308
retention, 308
ROM, 308
root, 309
  cautions, 89
  creating, 15
  interface, 91
  introduction, 88
  introspection, 95
  mode, 90
  rank, 90
  registering, 88
  thread, 89
root description, 89
root mode, 309
root set, 309
S
sbrk, 309
scalar data type, 309
scan, 310
scan method, 10, 310
scan state, 310
scanning
  ambiguous reference, 82
  critical path, 82
  example, 83
  fixing, 85
  interface, 84
  introduction, 80
  protocol, 81
  tagged reference, 81
  unfixed reference, 82
scavenging garbage collection, 310
Scheme, 364
  address-based hash table, 34
  allocation, 19
  finalization, 31
  forward method, 12
  global symbol table, 41
  interactive chatter, 100
  is-forwarded method, 13
  location dependency, 34
  object alignment, 9
  object format, 9
  padding method, 14
  pool, 15
  ports, 31
  root, 15
  scan method, 10
  scanning, 83
  segregation, 42
  skip method, 11
  telemetry, 122
  thread, 18
  thread root, 90
  toy interpreter, 6
  weak hash table, 37
SDRAM, 310
segmentation violation, 310
segmented addressing, 310
segregated allocation cache, 112, 311
  allocation, 115
  creating, 113
segregated fit, 311
segregated free list, 112, 311
segregated free-list, 311
semi-conservative garbage collection, 312
semi-space, 312
semi-space collector, 312
sequential fit, 312
sequential store buffer, 312
shared memory, 313
signal
  handling, 87
  simple object, 313
  simple segregated storage, 313
Simula, 364
size, 313
size class, 313
skip method, 11, 313
Smalltalk, 364
smart pointer, 313
snap-out, 314
snapshot at the beginning, 314
SNC, 174
  interface, 175
  properties, 174
soft reference, 315
softly reachable, 315
space leak, 315
spaghetti stack, 315
spare commit limit, 315
spare committed memory, 315
splat, 316
split, 316
SRAM, 316
SSB, 316
stack, 316
  allocation, 118
  stack allocation, 317
  stack frame, 317
  stack record, 317
staleness
  testing, 109
static allocation, 318
static memory (1), 318
static memory (2), 318
static object, 318
static RAM, 318
static storage duration, 319
stepper function, 319
sticky reference count, 319
stop-and-copy collection, 319
storage, 319
storage hierarchy, 319
storage level, 320
storage management, 320
store (1), 320
store (2), 320
stretchy vector, 320
stretchy vectors, 21
strict segregated fit, 321
strong reference, 321
strong root, 321
strong tri-color invariant, 321
strong tri-colour invariant, 321
strong tricolor invariant, 321
strong tricolour invariant, 321
strongly reachable, 322
suballocator, 322
subgraph, 322
superpage, 322
support policy
  interface, 45
sure reference, 322
swap space, 322
swapped in, 322
swapped out, 323
swapping, 323
sweeping, 323
synchronous garbage collector, 323
T
  tabling, 323
tag, 323
tagged architecture, 324
tagged reference, 324
    scanning, 81
TB (1), 324
TB (2), 325
tearing down, 21
telemetry
  customizing, 127
decoding event stream, 124
environment variables, 123
event categories, 123
example, 122
I/O module, 129
interface, 125
labels, 126
loading into SQLite, 125
making event stream readable, 124
utilities, 121
telemetry filter, 325
telemetry label, 325
telemetry stream, 325
temporal fit
  MVT, 172
tenuring, 325
terabyte, 325
termination, 325
trash, 325
thread, 86, 326
  exception handling, 87
  interface, 87
  registering, 18
  registration, 87
  root, 89
  signal handling, 87
thread safety, 86
  location dependency, 110
threatened set, 326
tidying up, 21
TLB, 326
to space, 326
tospace, 326
trace, 326
tracing garbage collection, 326
trampline, 151
translation buffer, 327
translation lookaside buffer, 327
transparent alias, 327
transparent type, 327
transport, 327
transport snap-out, 327
treadmill, 327
tri-color invariant, 327
tri-color marking, 328
tri-colour invariant, 327
tri-colour marking, 328
tricolor invariant, 327
tricolor marking, 328
tricolour invariant, 327
tricolour marking, 328
two space collector, 328
two-space collector, 328
type punning, 47, 329
type-accurate garbage collection, 329
types
  interface, 46
U
unaligned, 329
unboxed, 329
unclamped state, 61, 330
undead, 330
underscanning, 25
unfixed reference
  scanning, 82
unmapped, 330
unreachable, 330
unsure reference, 330
unwrapped, 331
use after free, 331
V
value object, 331
variety, 55, 332
cool, 24, 55
hot, 55
rash, 55
vector data type, 332
virtual address, 332
virtual address space, 332
virtual memory, 333
virtual memory arena, 333
virtual memory arena class, 57
visitor function, 333
VM (1), 333
VM (2), 334

W
weak hash table, 334
weak reference
   example, 37
weak reference (1), 334
weak reference (2), 334
weak references, 127
weak root, 335
weak tri-color invariant, 335
weak tri-colour invariant, 335
weak tricolor invariant, 335
weak tricolour invariant, 335
weak-key hash table, 334
weak-value hash table, 334
weakly reachable, 334
weighted buddies, 335
weighted reference counting, 336
white, 336
word, 336
working set, 336
worst fit, 337
wrapped, 337
wrapper, 337
write barrier, 337
write fault, 338

Z
ZCT, 338
zero count table, 338