mmalloc

The GNU memory-mapped malloc package

Fred Fish
Cygnus Support
Mike Haertel
Free Software Foundation

Cygnus Support
fnf@cygnus.com
MMALLOC, the GNU memory-mapped malloc package, Revision: 1.4
TEXinfo 1999-04-25.15

Copyright © 1992 Free Software Foundation, Inc.

Permission is granted to make and distribute verbatim copies of this manual provided the copyright notice and this permission notice are preserved on all copies.

Permission is granted to copy and distribute modified versions of this manual under the conditions for verbatim copying, provided also that the entire resulting derived work is distributed under the terms of a permission notice identical to this one.

Permission is granted to copy and distribute translations of this manual into another language, under the above conditions for modified versions.

1 Overall Description

This is a heavily modified version of GNU `malloc`. It uses `mmap` as the basic mechanism for obtaining memory from the system, rather than `sbrk`. This gives it several advantages over the more traditional malloc:

- Several different heaps can be used, each of them growing or shinking under control of `mmap`, with the `mmalloc` functions using a specific heap on a call by call basis.
- By using `mmap`, it is easy to create heaps which are intended to be persistent and exist as a filesystem object after the creating process has gone away.
- Because multiple heaps can be managed, data used for a specific purpose can be allocated into its own heap, making it easier to allow applications to "dump" and "restore" initialized malloc-managed memory regions. For example, the "unexec" hack popularized by GNU Emacs could potentially go away.

2 Implementation

The `mmalloc` functions contain no internal static state. All `mmalloc` internal data is allocated in the mapped in region, along with the user data that it manages. This allows it to manage multiple such regions and to "pick up where it left off" when such regions are later dynamically mapped back in.

In some sense, malloc has been "purified" to contain no internal state information and generalized to use multiple memory regions rather than a single region managed by `sbrk`. However the new routines now need an extra parameter which informs `mmalloc` which memory region it is dealing with (along with other information). This parameter is called the *malloc descriptor*.

The functions initially provided by `mmalloc` are:

```
void *mmalloc_attach (int fd, void *baseaddr);
void *mmalloc_detach (void *md);
int mmalloc_errno (void *md);
int mmalloc_setkey (void *md, int keynum, void *key);
void *mmalloc_getkey (void *md, int keynum);

void *mmalloc (void *md, size_t size);
void *mrealloc (void *md, void *ptr, size_t size);
void *mvalloc (void *md, size_t size);
void mfree (void *md, void *ptr);
```

2.1 Backwards Compatibility

To allow a single malloc package to be used in a given application, provision is made for the traditional `malloc`, `realloc`, and `free` functions to be implemented as special cases of the `mmalloc` functions. In particular, if any of the functions that expect malloc descriptors are called with a `NULL` pointer rather than a valid malloc descriptor, then they default to using an `sbrk` managed region. The `mmalloc` package provides compatible `malloc`, `realloc`, and `free` functions using this mechanism internally. Applications can avoid this extra interface layer by simply including the following defines:

```
#define malloc(size) mmalloc ((void *)0, (size))
#define realloc(ptr,size) mrealloc ((void *)0, (ptr), (size));
#define free(ptr) mfree ((void *)0, (ptr))
```

or replace the existing `malloc`, `realloc`, and `free` calls with the above patterns if using `#define` causes problems.

2.2 Function Descriptions

These are the details on the functions that make up the `mmalloc` package.

`void *mmalloc_attach (int` *fd*`, void *`*baseaddr*`);`

Initialize access to a `mmalloc` managed region.

If *fd* is a valid file descriptor for an open file, then data for the `mmalloc` managed region is mapped to that file. Otherwise '`/dev/zero`' is used and the data will not exist in any filesystem object.

If the open file corresponding to *fd* is from a previous use of `mmalloc` and passes some basic sanity checks to ensure that it is compatible with the current `mmalloc` package, then its data is mapped in and is immediately accessible at the same addresses in the current process as the process that created the file.

If *baseaddr* is not `NULL`, the mapping is established starting at the specified address in the process address space. If *baseaddr* is `NULL`, the `mmalloc` package chooses a suitable address at which to start the mapped region, which will be the value of the previous mapping if opening an existing file which was previously built by `mmalloc`, or for new files will be a value chosen by `mmap`.

Specifying *baseaddr* provides more control over where the regions start and how big they can be before bumping into existing mapped regions or future mapped regions.

On success, returns a malloc descriptor which is used in subsequent calls to other `mmalloc` package functions. It is explicitly '`void *`' ('`char *`' for systems that don't fully support `void`) so that users of the package don't have to worry about the actual implementation details.

On failure returns `NULL`.

`void *mmalloc_detach (void *`*md*`);`

Terminate access to a `mmalloc` managed region identified by the descriptor *md*, by closing the base file and unmapping all memory pages associated with the region.

Returns `NULL` on success.

Returns the malloc descriptor on failure, which can subsequently be used for further action (such as obtaining more information about the nature of the failure).

`void *mmalloc (void *`*md*`, size_t` *size*`);`

Given an `mmalloc` descriptor *md*, allocate additional memory of *size* bytes in the associated mapped region.

`*mrealloc (void *`*md*`, void *`*ptr*`, size_t` *size*`);`

Given an `mmalloc` descriptor *md* and a pointer to memory previously allocated by `mmalloc` in *ptr*, reallocate the memory to be *size* bytes long, possibly moving the existing contents of memory if necessary.

`void *mvalloc (void *`*md*`, size_t` *size*`);`

Like `mmalloc` but the resulting memory is aligned on a page boundary.

`void mfree (void *`*md*`, void *`*ptr*`);`

Given an `mmalloc` descriptor *md* and a pointer to memory previously allocated by `mmalloc` in *ptr*, free the previously allocated memory.

`int mmalloc_errno (void *`*md*`);`

Given a `mmalloc` descriptor, if the last `mmalloc` operation failed for some reason due to a system call failure, then returns the associated `errno`. Returns 0 otherwise. (This function is not yet implemented).