Mapping Files into Memory

As an alternative to standard file I/O, the kernel provides an interface that allows an application to map a file into memory, meaning that there is a one-to-one correspondence between a memory address and a word in the file. The programmer can then access the file directly through memory, identically to any other chunk of memory-resident data—it is even possible to allow writes to the memory region to transparently map back to the file on disk.

POSIX.1 standardizes—and Linux implements—the mmap( ) system call for mapping objects into memory. This section will discuss mmap( ) as it pertains to mapping files into memory to perform I/O; in [Chapter 8](https://www.safaribooksonline.com/library/view/linux-system-programming/0596009585/ch08.html), we will visit other applications of mmap( ).

mmap( )

A call to mmap( ) asks the kernel to map len bytes of the object represented by the file descriptor fd, starting at offset bytes into the file, into memory. Ifaddr is included, it indicates a preference to use that starting address in memory. The access permissions are dictated by prot, and additional behavior can be given by flags:

**#include <sys/mman.h>**

**void \* mmap (void \*addr,**

**size\_t len,**

**int prot,**

**int flags,**

**int fd,**

**off\_t offset);**

The addr parameter offers a suggestion to the kernel of where best to map the file. It is only a hint; most users pass 0. The call returns the actual address in memory where the mapping begins.

The prot parameter describes the desired memory protection of the mapping. It may be one or more of the following flags:

PROT\_READ

The pages may be read.

PROT\_WRITE

The pages may be written.

PROT\_EXEC

The pages may be executed.

The desired memory protection must not conflict with the open mode of the file. For example, if the program opens the file read-only, prot must not specify PROT\_WRITE.

The flags argument describes the type of mapping, and some elements of its behavior. It is a bitwise OR of the following values:

MAP\_FIXED

Instructs mmap( ) to treat addr as a requirement, not a hint. If the kernel is unable to place the mapping at the given address, the call fails. If the address and length parameters overlap an existing mapping, the overlapped pages are discarded and replaced by the new mapping. As this option requires intimate knowledge of the process address space, it is nonportable, and its use is discouraged.

MAP\_PRIVATE

States that the mapping is not shared. The file is mapped copy-on-write, and any changes made in memory by this process are not reflected in the actual file, or in the mappings of other processes.

MAP\_SHARED

Shares the mapping with all other processes that map this same file. Writing into the mapping is equivalent to writing to the file. Reads from the mapping will reflect the writes of other processes.

Either MAP\_SHARED or MAP\_PRIVATE must be specified, but not both. Other, more advanced flags are discussed in [Chapter 8](https://www.safaribooksonline.com/library/view/linux-system-programming/0596009585/ch08.html).

When you map a file descriptor, the file's reference count is incremented. Therefore, you can close the file descriptor after mapping the file, and your process will still have access to it. The corresponding decrement of the file's reference count will occur when you unmap the file, or when the process terminates.

As an example, the following snippet maps the file backed by fd, beginning with its first byte, and extending for len bytes, into a read-only mapping:

void \*p;

p = mmap (0, len, PROT\_READ, MAP\_SHARED, fd, 0);

if (p == MAP\_FAILED)

perror ("mmap")

munmap( )

Linux provides the munmap( ) system call for removing a mapping created with mmap( ):

**#include <sys/mman.h>**

**int munmap (void \*addr, size\_t len);**

As an example, the following snippet unmaps any memory regions with pages contained in the interval [addr,addr+len]:

if (munmap (addr, len) == −1)

perror ("munmap");

Advantages of mmap( )

Manipulating files via mmap( ) has a handful of advantages over the standard read( ) and write( ) system calls. Among them are:

* Reading from and writing to a memory-mapped file avoids the extraneous copy that occurs when using the read( ) or write( ) system calls, where the data must be copied to and from a user-space buffer.
* Aside from any potential page faults, reading from and writing to a memory-mapped file does not incur any system call or context switch overhead. It is as simple as accessing memory.
* When multiple processes map the same object into memory, the data is shared among all the processes. Read-only and shared writable mappings are shared in their entirety; private writable mappings have their not-yet-COW (copy-on-write) pages shared.
* Seeking around the mapping involves trivial pointer manipulations. There is no need for the lseek( ) system call.

For these reasons, mmap( ) is a smart choice for many applications.

Disadvantages of mmap( )

There are a few points to keep in mind when using mmap( ):

* Memory mappings are always an integer number of pages in size. Thus, the difference between the size of the backing file and an integer number of pages is "wasted" as slack space. For small files, a significant percentage of the mapping may be wasted. For example, with 4 KB pages, a 7 byte mapping wastes 4,089 bytes.
* The memory mappings must fit into the process' address space. With a 32-bit address space, a very large number of various-sized mappings can result in fragmentation of the address space, making it hard to find large free contiguous regions. This problem, of course, is much less apparent with a 64-bit address space.
* There is overhead in creating and maintaining the memory mappings and associated data structures inside the kernel. This overhead is generally obviated by the elimination of the double copy mentioned in the previous section, particularly for larger and frequently accessed files.

For these reasons, the benefits of mmap( ) are most greatly realized when the mapped file is large (and thus any wasted space is a small percentage of the total mapping), or when the total size of the mapped file is evenly divisible by the page size (and thus there is no wasted space).

Resizing a Mapping

Linux provides the mremap( ) system call for expanding or shrinking the size of a given mapping. This function is Linux-specific:

**#define \_GNU\_SOURCE**

**#include <unistd.h>**

**#include <sys/mman.h>**

**void \* mremap (void \*addr, size\_t old\_size,**

**size\_t new\_size, unsigned long flags);**

A call to mremap( ) expands or shrinks mapping in the region [addr,addr+old\_size) to the new size new\_size. The kernel can potentially move the mapping at the same time, depending on the availability of space in the process' address space and the value of flags.

TIP

The opening [ in [addr,addr+old\_size) indicates that the region starts with (and includes) the low address, whereas the closing ) indicates that the region stops just before (does not include) the high address. This convention is known as *interval notation*.

The flags parameter can be either 0 or MREMAP\_MAYMOVE, which specifies that the kernel is free to move the mapping, if required, in order to perform the requested resizing. A large resizing is more likely to succeed if the kernel can move the mapping.

Return values and error codes

On success, mremap( ) returns a pointer to the newly resized memory mapping. On failure, it returns MAP\_FAILED, and sets errno to one of the following:

EAGAIN

The memory region is locked, and cannot be resized.

EFAULT

Some pages in the given range are not valid pages in the process' address space, or there was a problem remapping the given pages.

EINVAL

An argument was invalid.

ENOMEM

The given range cannot be expanded without moving (andMREMAP\_MAYMOVE was not given), or there is not enough free space in the process' address space.

Libraries such as *glibc* often use mremap( ) to implement an efficient realloc( ), which is an interface for resizing a block of memory originally obtained via malloc( ). For example:

void \* realloc (void \*addr, size\_t len)

{

size\_t old\_size = look\_up\_mapping\_size (addr);

void \*p;

p = mremap (addr, old\_size, len, MREMAP\_MAYMOVE);

if (p == MAP\_FAILED)

return NULL;

return p;

}

This would only work if all malloc( ) allocations were unique anonymous mappings; nonetheless, it stands as a useful example of the performance gains to be had. The example assumes the programmer has written a look\_up\_mapping\_size( ) function.

The GNU C library does use mmap( ) and family for performing some memory allocations. We will look that topic in depth in [Chapter 8](https://www.safaribooksonline.com/library/view/linux-system-programming/0596009585/ch08.html).

Synchronizing a File with a Mapping

POSIX provides a memory-mapped equivalent of the fsync( ) system call that we discussed in [Chapter 2](https://www.safaribooksonline.com/library/view/linux-system-programming/0596009585/ch02.html):

**#include <sys/mman.h>**

**int msync (void \*addr, size\_t len, int flags);**

A call to msync( ) flushes back to disk any changes made to a file mapped via mmap( ), synchronizing the mapped file with the mapping. Specifically, the file or subset of a file associated with the mapping starting at memory address addr and continuing for len bytes is synchronized to disk. The addr argument must be page-aligned; it is generally the return value from a previous mmap( ) invocation.

Without invocation of msync( ), there is no guarantee that a dirty mapping will be written back to disk until the file is unmapped. This is different from the behavior of write( ), where a buffer is dirtied as part of the writing process, and queued for writeback to disk. When writing into a memory mapping, the process directly modifies the file's pages in the kernel's page cache, without kernel involvement. The kernel may not synchronize the page cache and the disk anytime soon.

The flags parameter controls the behavior of the synchronizing operation. It is a bitwise OR of the following values:

MS\_ASYNC

Specifies that synchronization should occur asynchronously. The update is scheduled, but the msync( ) call returns immediately without waiting for the writes to take place.

MS\_INVALIDATE

Specifies that all other cached copies of the mapping be invalidated. Any future access to any mappings of this file will reflect the newly synchronized on-disk contents.

MS\_SYNC

Specifies that synchronization should occur synchronously. The msync( ) call will not return until all pages are written back to disk.

Either MS\_ASYNC or MS\_SYNC must be specified, but not both.

Usage is simple:

if (msync (addr, len, MS\_ASYNC) == −1)

perror ("msync");

This example asynchronously synchronizes (say that 10 times fast) to disk the file mapped in the region [addr,addr+len).

Return values and error codes

On success, msync( ) returns 0. On failure, the call returns −1.