NAME

mmap, mmap64, munmap − map or unmap files or devices into memory

SYNOPSIS

#include <sys/mman.h>

void *mmap(void * *addr***, size_t** *length***, int** *prot***, int** *flags***,**
int *fd***, off_t** *offset***);**
void *mmap64(void * *addr***, size_t** *length***, int** *prot***, int** *flags***,**
int *fd***, off64_t** *offset***);**
int munmap(void * *addr***, size_t** *length***);**

DESCRIPTION

mmap() creates a new mapping in the virtual address space of the calling process. The starting address for the new mapping is specified in *addr*. The *length* argument specifies the length of the mapping.

If *addr* is NULL, then the kernel chooses the address at which to create the mapping; this is the most portable method of creating a new mapping. If *addr* is not NULL, then the kernel takes it as a hint about where to place the mapping; on Linux, the mapping will be created at a nearby page boundary. The address of the new mapping is returned as the result of the call.

The contents of a file mapping (as opposed to an anonymous mapping; see **MAP_ANONYMOUS** below), are initialized using *length* bytes starting at offset *offset* in the file (or other object) referred to by the file descriptor *fd*. *offset* must be a multiple of the page size as returned by *sysconf(_SC_PAGE_SIZE)*.

The *prot* argument describes the desired memory protection of the mapping (and must not conflict with the open mode of the file). It is either **PROT_NONE** or the bitwise OR of one or more of the following flags:

PROT_EXEC Pages may be executed.

PROT_READ Pages may be read.

PROT_WRITE Pages may be written.

PROT_NONE Pages may not be accessed.

The *flags* argument determines whether updates to the mapping are visible to other processes mapping the same region, and whether updates are carried through to the underlying file. This behavior is determined by including exactly one of the following values in *flags*:

MAP_SHARED Share this mapping. Updates to the mapping are visible to other processes that map this file, and are carried through to the underlying file. The file may not actually be updated until **msync**(2) or **munmap**() is called.

MAP_PRIVATE Create a private copy-on-write mapping. Updates to the mapping are not visible to other processes mapping the same file, and are not carried through to the underlying file. It is unspecified whether changes made to the file after the **mmap**() call are visible in the mapped region.

Both of these flags are described in POSIX.1-2001.

In addition, zero or more of the following values can be ORed in *flags*:

MAP_32BIT (since Linux 2.4.20, 2.6)
Put the mapping into the first 2 Gigabytes of the process address space. This flag is only supported on x86-64, for 64-bit programs. It was added to allow thread stacks to be allocated somewhere in the first 2GB of memory, so as to improve context-switch performance on some early 64-bit processors. Modern x86-64 processors no longer have this performance problem, so use of this flag is not required on those systems. The **MAP_32BIT** flag is ignored when **MAP_FIXED** is set.

MAP_ANON
Synonym for **MAP_ANONYMOUS**. Deprecated.

MAP_ANONYMOUS
The mapping is not backed by any file; its contents are initialized to zero. The *fd* and *offset* arguments are ignored; however, some implementations require *fd* to be −1 if **MAP_ANONYMOUS** (or **MAP_ANON**) is specified, and portable applications should ensure this. The use of **MAP_ANONYMOUS** in conjunction with **MAP_SHARED** is only supported on Linux since kernel 2.4.

MAP_DENYWRITE
This flag is ignored. (Long ago, it signaled that attempts to write to the underlying file should fail with **ETXTBUSY**. But this was a source of denial-of-service attacks.)

MAP_EXECUTABLE
This flag is ignored.

MAP_FILE
Compatibility flag. Ignored.

MAP_FIXED
Don't interpret *addr* as a hint: place the mapping at exactly that address. *addr* must be a multiple of the page size. If the memory region specified by *addr* and *len* overlaps pages of any existing mapping(s), then the overlapped part of the existing mapping(s) will be discarded. If the specified address cannot be used, **mmap**() will fail. Because requiring a fixed address for a mapping is less portable, the use of this option is discouraged.

MAP_GROWSDOWN
Used for stacks. Indicates to the kernel virtual memory system that the mapping should extend downwards in memory.

MAP_HUGETLB (since Linux 2.6.32)
Allocate the mapping using "huge pages." See the kernel source file *Documentation/vm/hugetlbpage.txt* for further information.

MAP_LOCKED (since Linux 2.5.37)
Lock the pages of the mapped region into memory in the manner of **mlock**(2). This flag is ignored in older kernels.

MAP_NONBLOCK (since Linux 2.5.46)
Only meaningful in conjunction with **MAP_POPULATE**. Don't perform read-ahead: only create page tables entries for pages that are already present in RAM. Since Linux 2.6.23, this flag causes **MAP_POPULATE** to do nothing. One day the combination of **MAP_POPULATE** and **MAP_NONBLOCK** may be reimplemented.

MAP_NORESERVE
Do not reserve swap space for this mapping. When swap space is reserved, one has the guarantee that it is possible to modify the mapping. When swap space is not reserved one might get **SIGSEGV** upon a write if no physical memory is available. See also the discussion of the file */proc/sys/vm/overcommit_memory* in **proc**(5). In kernels before 2.6, this flag only had effect for private writable mappings.

MAP_POPULATE (since Linux 2.5.46)
Populate (prefault) page tables for a mapping. For a file mapping, this causes read-ahead on the file. Later accesses to the mapping will not be blocked by page faults. **MAP_POPULATE** is only supported for private mappings since Linux 2.6.23.

Of the above flags, only **MAP_FIXED** is specified in POSIX.1-2001. However, most systems also support **MAP_ANONYMOUS** (or its synonym **MAP_ANON**).

MAP_STACK (since Linux 2.6.27)
Allocate the mapping at an address suitable for a process or thread stack. This flag is currently a no-op, but is used in the glibc threading implementation so that if some architectures require special treatment for stack allocations, support can later be transparently implemented for glibc.

Some systems document the additional flags **MAP_AUTOGROW**, **MAP_AUTORESRV**, **MAP_COPY**, and **MAP_LOCAL**.

Memory mapped by **mmap**() is preserved across **fork**(2), with the same attributes.

A file is mapped in multiples of the page size. For a file that is not a multiple of the page size, the remaining memory is zeroed when mapped, and writes to that region are not written out to the file. The effect of changing the size of the underlying file of a mapping on the pages that correspond to added or removed regions of the file is unspecified.

mmap64()
The **mmap64**() system call operates in exactly the same way as **mmap**(), except that the final argument specifies the offset as a 64-bit off64_t. This enables applications to aceess the large files.

munmap()
The **munmap**() system call deletes the mappings for the specified address range, and causes further references to addresses within the range to generate invalid memory references. The region is also automatically unmapped when the process is terminated. On the other hand, closing the file descriptor does not unmap the region.

The address *addr* must be a multiple of the page size. All pages containing a part of the indicated range are unmapped, and subsequent references to these pages will generate **SIGSEGV**. It is not an error if the indicated range does not contain any mapped pages.

Timestamps changes for file-backed mappings
For file-backed mappings, the *st_atime* field for the mapped file may be updated at any time between the **mmap**() and the corresponding unmapping; the first reference to a mapped page will update the field if it has not been already.

The *st_ctime* and *st_mtime* field for a file mapped with **PROT_WRITE** and **MAP_SHARED** will be updated after a write to the mapped region, and before a subsequent **msync**(2) with the **MS_SYNC** or **MS_ASYNC** flag, if one occurs.

RETURN VALUE
On success, **mmap**() returns a pointer to the mapped area. On error, the value **MAP_FAILED** (that is, *(void *) −1*) is returned, and *errno* is set appropriately. On success, **munmap**() returns 0, on failure −1, and *errno* is set (probably to **EINVAL**).

ERRORS
EACCES
A file descriptor refers to a non-regular file. Or **MAP_PRIVATE** was requested, but *fd* is not open for reading. Or **MAP_SHARED** was requested and **PROT_WRITE** is set, but *fd* is not open in read/write (**O_RDWR**) mode. Or **PROT_WRITE** is set, but the file is append-only.

EAGAIN
The file has been locked, or too much memory has been locked (see **setrlimit**(2)).

EBADF
fd is not a valid file descriptor (and **MAP_ANONYMOUS** was not set).

EINVAL
We don't like *addr*, *length*, or *offset* (e.g., they are too large, or not aligned on a page boundary).

EINVAL
(since Linux 2.6.12) *length* was 0.

EINVAL
flags contained neither **MAP_PRIVATE** or **MAP_SHARED**, or contained both of these values.

ENFILE
The system limit on the total number of open files has been reached.

ENODEV
The underlying file system of the specified file does not support memory mapping.

ENOMEM
No memory is available, or the process's maximum number of mappings would have been exceeded.

EPERM
The *prot* argument asks for **PROT_EXEC** but the mapped area belongs to a file on a file system that was mounted no-exec.

ETXTBSY
MAP_DENYWRITE was set but the object specified by *fd* is open for writing.

Use of a mapped region can result in these signals:

SIGSEGV
Attempted write into a region mapped as read-only.

SIGBUS
Attempted access to a portion of the buffer that does not correspond to the file (for example, beyond the end of the file, including the case where another process has truncated the file).

CONFORMING TO

SVr4, 4.4BSD, POSIX.1-2001.

AVAILABILITY

On POSIX systems on which **mmap**(), **msync**(2) and **munmap**() are available, **_POSIX_MAPPED_FILES** is defined in <*unistd.h*> to a value greater than 0. (See also **sysconf**(3).)

NOTES

Since kernel 2.4, this system call has been superseded by **mmap2**(2). Nowadays, the glibc **mmap**() wrapper function invokes **mmap2**(2) with a suitably adjusted value for *offset*.

On some hardware architectures (e.g., i386), **PROT_WRITE** implies **PROT_READ**. It is architecture dependent whether **PROT_READ** implies **PROT_EXEC** or not. Portable programs should always set **PROT_EXEC** if they intend to execute code in the new mapping.

The portable way to create a mapping is to specify *addr* as 0 (NULL), and omit **MAP_FIXED** from *flags*. In this case, the system chooses the address for the mapping; the address is chosen so as not to conflict with any existing mapping, and will not be 0. If the **MAP_FIXED** flag is specified, and *addr* is 0 (NULL), then the mapped address will be 0 (NULL).

BUGS

On Linux there are no guarantees like those suggested above under **MAP_NORESERVE**. By default, any process can be killed at any moment when the system runs out of memory.

In kernels before 2.6.7, the **MAP_POPULATE** flag only has effect if *prot* is specified as **PROT_NONE**.

SUSv3 specifies that **mmap**() should fail if *length* is 0. However, in kernels before 2.6.12, **mmap**() succeeded in this case: no mapping was created and the call returned *addr*. Since kernel 2.6.12, **mmap**() fails with the error **EINVAL** for this case.

EXAMPLE

The following program prints part of the file specified in its first command-line argument to standard output. The range of bytes to be printed is specified via offset and length values in the second and third

command-line arguments. The program creates a memory mapping of the required pages of the file and then uses **write**(2) to output the desired bytes.

```
#include <sys/mman.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#define handle_error(msg) \
   do { perror(msg); exit(EXIT_FAILURE); } while (0)

int
main(int argc, char *argv[])
{
   char *addr;
   int fd;
   struct stat sb;
   off_t offset, pa_offset;
   size_t length;
   ssize_t s;

   if (argc < 3 || argc > 4) {
      fprintf(stderr, "%s file offset [length]\n", argv[0]);
      exit(EXIT_FAILURE);
   }

   fd = open(argv[1], O_RDONLY);
   if (fd == -1)
      handle_error("open");

   if (fstat(fd, &sb) == -1)           /* To obtain file size */
      handle_error("fstat");

   offset = atoi(argv[2]);
   pa_offset = offset & ~(sysconf(_SC_PAGE_SIZE) - 1);
      /* offset for mmap() must be page aligned */

   if (offset >= sb.st_size) {
      fprintf(stderr, "offset is past end of file\n");
      exit(EXIT_FAILURE);
   }

   if (argc == 4) {
      length = atoi(argv[3]);
      if (offset + length > sb.st_size)
         length = sb.st_size - offset;
             /* Can't display bytes past end of file */

   } else {    /* No length arg ==> display to end of file */
      length = sb.st_size - offset;
   }
```

```
    addr = mmap(NULL, length + offset − pa_offset, PROT_READ,
              MAP_PRIVATE, fd, pa_offset);
    if (addr == MAP_FAILED)
       handle_error("mmap");

    s = write(STDOUT_FILENO, addr + offset − pa_offset, length);
    if (s != length) {
       if (s == −1)
          handle_error("write");

       fprintf(stderr, "partial write");
       exit(EXIT_FAILURE);
    }

    exit(EXIT_SUCCESS);
} /* main */
```

SEE ALSO

getpagesize(2), **mincore**(2), **mlock**(2), **mmap2**(2), **mprotect**(2), **mremap**(2), **msync**(2), **remap_file_pages**(2), **setrlimit**(2), **shmat**(2), **shm_open**(3), **shm_overview**(7)
B.O. Gallmeister, POSIX.4, O'Reilly, pp. 128-129 and 389-391.

COLOPHON

This page is part of release 3.24 of the Linux *man-pages* project. A description of the project, and information about reporting bugs, can be found at http://www.kernel.org/doc/man-pages/.