NAME

mmap - allocate memory, or map files or devices into memory

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/mman.h>

*void **
mmap(*void *addr, size_t len, int prot, int flags, int fd, off_t offset*);

DESCRIPTION

The **mmap**() system call causes the pages starting at *addr* and continuing for at most *len* bytes to be mapped from the object described by *fd*, starting at byte offset *offset*. If *len* is not a multiple of the page size, the mapped region may extend past the specified range. Any such extension beyond the end of the mapped object will be zero-filled.

If *fd* references a regular file or a shared memory object, the range of bytes starting at *offset* and continuing for *len* bytes must be legitimate for the possible (not necessarily current) offsets in the object. In particular, the *offset* value cannot be negative. If the object is truncated and the process later accesses a page that is wholly within the truncated region, the access is aborted and a SIGBUS signal is delivered to the process.

If *fd* references a device file, the interpretation of the *offset* value is device specific and defined by the device driver. The virtual memory subsystem does not impose any restrictions on the *offset* value in this case, passing it unchanged to the driver.

If *addr* is non-zero, it is used as a hint to the system. (As a convenience to the system, the actual address of the region may differ from the address supplied.) If *addr* is zero, an address will be selected by the system. The actual starting address of the region is returned. A successful *mmap* deletes any previous mapping in the allocated address range.

The protections (region accessibility) are specified in the *prot* argument by *or'*ing the following values:

PROT_NONE Pages may not be accessed.
PROT_READ Pages may be read.
PROT_WRITE Pages may be written.
PROT_EXEC Pages may be executed.

In addition to these protection flags, FreeBSD provides the ability to set the maximum protection of a region allocated by **mmap** and later altered by mprotect(2). This is accomplished by *or*'ing one or more PROT_ values wrapped in the PROT_MAX() macro into the *prot* argument.

The *flags* argument specifies the type of the mapped object, mapping options and whether modifications made to the mapped copy of the page are private to the process or are to be shared with other references. Sharing, mapping type and options are specified in the *flags* argument by *or*'ing the following values:

MAP_32BIT — Request a region in the first 2GB of the current process's address space. If a suitable region cannot be found, **mmap**() will fail.

MAP_ALIGNED(*n*) — Align the region on a requested boundary. If a suitable region cannot be found, **mmap**() will fail. The *n* argument specifies the binary logarithm of the desired alignment.

MAP_ALIGNED_SUPER — Align the region to maximize the potential use of large ("super") pages. If a suitable region cannot be found, **mmap**() will fail. The system will choose a suitable page size based on the size of mapping. The page size used as well as the alignment of the region may both be affected by properties of the file being mapped. In particular, the physical address of existing pages of a file may require a specific alignment. The region is not guaranteed to be aligned on any specific boundary.

MAP_ANON — Map anonymous memory not associated with any specific file. The file descriptor used for creating MAP_ANON must be -1. The *offset* argument must be 0.

MAP_ANONYMOUS — This flag is identical to MAP_ANON and is provided for compatibility.

MAP_EXCL — This flag can only be used in combination with MAP_FIXED. Please see the definition of MAP_FIXED for the description of its effect.

MAP_FIXED — Do not permit the system to select a different address than the one specified. If the specified address cannot be used, **mmap**() will fail. If MAP_FIXED is specified, *addr* must be a multiple of the page size. If MAP_EXCL is not specified, a successful MAP_FIXED request replaces any previous mappings for the process' pages in the range from *addr* to *addr* + *len*. In contrast, if MAP_EXCL is specified, the request will fail if a mapping already exists within the range.

MAP_GUARD — Instead of a mapping, create a guard of the specified size. Guards allow a process to create reservations in its address space, which can later be replaced by actual mappings.

mmap will not create mappings in the address range of a guard unless the request specifies MAP_FIXED. Guards can be destroyed with munmap(2). Any memory access by a thread to the guarded range results in the delivery of a SIGSEGV signal to that thread.

MAP_NOCORE — Region is not included in a core file.

MAP_NOSYNC — Causes data dirtied via this VM map to be flushed to physical media only when necessary (usually by the pager) rather than gratuitously. Typically this prevents the update daemons from flushing pages dirtied through such maps and thus allows efficient sharing of memory across unassociated processes using a file-backed shared memory map. Without this option any VM pages you dirty may be flushed to disk every so often (every 30-60 seconds usually) which can create performance problems if you do not need that to occur (such as when you are using shared file-backed mmap regions for IPC purposes). Dirty data will be flushed automatically when all mappings of an object are removed and all descriptors referencing the object are closed. Note that VM/file system coherency is maintained whether you use MAP_NOSYNC or not. This option is not portable across UNIX platforms (yet), though some may implement the same behavior by default.

WARNING! Extending a file with ftruncate(2), thus creating a big hole, and then filling the hole by modifying a shared **mmap**() can lead to severe file fragmentation. In order to avoid such fragmentation you should always pre-allocate the file’s backing store by **write**()ing zero’s into the newly extended area prior to modifying the area via your **mmap**(). The fragmentation problem is especially sensitive to MAP_NOSYNC pages, because pages may be flushed to disk in a totally random order.

The same applies when using MAP_NOSYNC to implement a file-based shared memory store. It is recommended that you create the backing store by **write**()ing zero’s to the backing file rather than **ftruncate**()ing it. You can test file fragmentation by observing the KB/t (kilobytes per transfer) results from an "iostat 1" while reading a large file sequentially, e.g., using "dd if=filename of=/dev/null bs=32k".

The fsync(2) system call will flush all dirty data and metadata associated with a file, including dirty NOSYNC VM data, to physical media. The sync(8) command and sync(2) system call generally do not flush dirty NOSYNC VM data. The msync(2) system call is usually not needed since BSD implements a coherent file system buffer cache. However, it may be used to associate dirty VM pages with file system buffers and thus cause them to be flushed to physical media sooner rather than later.

MAP_PREFAULT_READ Immediately update the calling process's lowest-level virtual address translation structures, such as its page table, so that every memory resident page within the region is mapped for read access. Ordinarily these structures are updated lazily. The effect of this option is to eliminate any soft faults that would otherwise occur on the initial read accesses to the region. Although this option does not preclude *prot* from including PROT_WRITE, it does not eliminate soft faults on the initial write accesses to the region.

MAP_PRIVATE Modifications are private.

MAP_SHARED Modifications are shared.

MAP_STACK Creates both a mapped region that grows downward on demand and an adjoining guard that both reserves address space for the mapped region to grow into and limits the mapped region's growth. Together, the mapped region and the guard occupy *len* bytes of the address space. The guard starts at the returned address, and the mapped region ends at the returned address plus *len* bytes. Upon access to the guard, the mapped region automatically grows in size, and the guard shrinks by an equal amount. Essentially, the boundary between the guard and the mapped region moves downward so that the access falls within the enlarged mapped region. However, the guard will never shrink to less than the number of pages specified by the sysctl security.bsd.stack_guard_page, thereby ensuring that a gap for detecting stack overflow always exists between the downward growing mapped region and the closest mapped region beneath it.

MAP_STACK implies MAP_ANON and *offset* of 0. The *fd* argument must be -1 and *prot* must include at least PROT_READ and PROT_WRITE. The size of the guard, in pages, is specified by sysctl security.bsd.stack_guard_page.

The close(2) system call does not unmap pages, see munmap(2) for further information.

NOTES

Although this implementation does not impose any alignment restrictions on the *offset* argument, a portable program must only use page-aligned values.

Large page mappings require that the pages backing an object be aligned in matching blocks in both the virtual address space and RAM. The system will automatically attempt to use large page mappings when mapping an object that is already backed by large pages in RAM by aligning the mapping request in the virtual address space to match the alignment of the large physical pages. The system may also use large page mappings when mapping portions of an object that are not yet backed by pages in RAM. The MAP_ALIGNED_SUPER flag is an optimization that will align the mapping request to the size of a large page similar to MAP_ALIGNED, except that the system will override this alignment if an object already uses large pages so that the mapping will be consistent with the existing large pages. This flag is mostly useful for maximizing the use of large pages on the first mapping of objects that do not yet have pages present in RAM.

RETURN VALUES

Upon successful completion, **mmap**() returns a pointer to the mapped region. Otherwise, a value of MAP_FAILED is returned and *errno* is set to indicate the error.

ERRORS

The **mmap**() system call will fail if:

[EACCES] The flag PROT_READ was specified as part of the *prot* argument and *fd* was not open for reading. The flags MAP_SHARED and PROT_WRITE were specified as part of the *flags* and *prot* argument and *fd* was not open for writing.

[EBADF] The *fd* argument is not a valid open file descriptor.

[EINVAL] An invalid (negative) value was passed in the *offset* argument, when *fd* referenced a regular file or shared memory.

[EINVAL] An invalid value was passed in the *prot* argument.

[EINVAL] An undefined option was set in the *flags* argument.

[EINVAL] Both MAP_PRIVATE and MAP_SHARED were specified.

[EINVAL] None of MAP_ANON, MAP_GUARD, MAP_PRIVATE, MAP_SHARED, or

MAP_STACK was specified. At least one of these flags must be included.

[EINVAL] MAP_STACK was specified and *len* is less than or equal to the guard size.

[EINVAL] MAP_FIXED was specified and the *addr* argument was not page aligned, or part of the desired address space resides out of the valid address space for a user process.

[EINVAL] Both MAP_FIXED and MAP_32BIT were specified and part of the desired address space resides outside of the first 2GB of user address space.

[EINVAL] The *len* argument was equal to zero.

[EINVAL] MAP_ALIGNED was specified and the desired alignment was either larger than the virtual address size of the machine or smaller than a page.

[EINVAL] MAP_ANON was specified and the *fd* argument was not -1.

[EINVAL] MAP_ANON was specified and the *offset* argument was not 0.

[EINVAL] Both MAP_FIXED and MAP_EXCL were specified, but the requested region is already used by a mapping.

[EINVAL] MAP_EXCL was specified, but MAP_FIXED was not.

[EINVAL] MAP_GUARD was specified, but the *offset* argument was not zero, the *fd* argument was not -1, or the *prot* argument was not PROT_NONE.

[EINVAL] MAP_GUARD was specified together with one of the flags MAP_ANON, MAP_PREFAULT, MAP_PREFAULT_READ, MAP_PRIVATE, MAP_SHARED, MAP_STACK.

[ENODEV] MAP_ANON has not been specified and *fd* did not reference a regular or character special file.

[ENOMEM] MAP_FIXED was specified and the *addr* argument was not available. MAP_ANON was specified and insufficient memory was available.

[ENOTSUP] The *prot* argument contains protections which are not a subset of the specified maximum protections.

SEE ALSO

madvise(2), mincore(2), minherit(2), mlock(2), mprotect(2), msync(2), munlock(2), munmap(2), getpagesize(3), getpagesizes(3)

HISTORY

The **mmap** system call was first documented in 4.2BSD and implemented in 4.4BSD.

The PROT_MAX functionality was introduced in FreeBSD 13.