Full credit is given to the above companies including the OS that this PDF file was generated!

Red Hat Enterprise Linux Release 9.2 Manual Pages on 'mmap.3p' command

$ man mmap.3p

MMAP(3P) POSIX Programmer's Manual MMAP(3P)

PROLOG

This manual page is part of the POSIX Programmer's Manual. The Linux implementation of this interface may differ (consult the corresponding Linux manual page for details of Linux behavior), or the interface may not be implemented on Linux.

NAME

mmap ? map pages of memory

SYNOPSIS

```
#include <sys/mman.h>
void *mmap(void *addr, size_t len, int prot, int flags,
    int fildes, off_t off);
```

DESCRIPTION

The mmap() function shall establish a mapping between an address space of a process and a memory object.

The mmap() function shall be supported for the following memory ob? jects:

* Regular files
* Shared memory objects
* Typed memory objects

Support for any other type of file is unspecified.

The format of the call is as follows:

pa=mmap(addr, len, prot, flags, fildes, off);

The mmap() function shall establish a mapping between the address space of the process at an address pa for len bytes to the memory object rep? resented by the file descriptor fildes at offset off for len bytes. The value of pa is an implementation-defined function of the parameter addr and the values of flags, further described below. A successful mmap() call shall return pa as its result. The address range starting at pa and continuing for len bytes shall be legitimate for the possible (not necessarily current) address space of the process. The range of bytes starting at off and continuing for len bytes shall be legitimate for the possible (not necessarily current) offsets in the memory object represented by fildes.

If fildes represents a typed memory object opened with either the POSIX_TYPED_MEM_ALLOCATE flag or the POSIX_TYPED_MEM_ALLOCATE_CONTIG flag, the memory object to be mapped shall be that portion of the typed memory object allocated by the implementation as specified below. In this case, if off is non-zero, the behavior of mmap() is undefined. If fildes refers to a valid typed memory object that is not accessible from the calling process, mmap() shall fail.

The mapping established by mmap() shall replace any previous mappings for those whole pages containing any part of the address space of the process starting at pa and continuing for len bytes.

If the size of the mapped file changes after the call to mmap() as a result of some other operation on the mapped file, the effect of refer? ences to portions of the mapped region that correspond to added or re? moved portions of the file is unspecified.

If len is zero, mmap() shall fail and no mapping shall be established.

The parameter prot determines whether read, write, execute, or some combination of accesses are permitted to the data being mapped. The prot shall be either PROT_NONE or the bitwise-inclusive OR of one or more of the other flags in the following table, defined in the <sys/mman.h> header.

?Symbolic Constant ?	Description ?

??

?PROT_READ ? Data can be read. ?

?PROT_WRITE ? Data can be written. ?

?PROT_EXEC ? Data can be executed. ?

?PROT_NONE ? Data cannot be accessed. ?

??

If an implementation cannot support the combination of access types specified by prot, the call to mmap() shall fail.

An implementation may permit accesses other than those specified by prot; however, the implementation shall not permit a write to succeed where PROT_WRITE has not been set and shall not permit any access where PROT_NONE alone has been set. The implementation shall support at least the following values of prot: PROT_NONE, PROT_READ, PROT_WRITE, and the bitwise-inclusive OR of PROT_READ and PROT_WRITE. The file descriptor fildes shall have been opened with read permission, regardless of the protection options specified. If PROT_WRITE is specified, the applica? tion shall ensure that it has opened the file descriptor fildes with write permission unless MAP_PRIVATE is specified in the flags parameter as described below.

The parameter flags provides other information about the handling of the mapped data. The value of flags is the bitwise-inclusive OR of these options, defined in <sys/mman.h>:

??

?Symbolic Constant ? Description ?

??

?MAP_SHARED ? Changes are shared. ?

?MAP_PRIVATE ? Changes are private. ?

?MAP_FIXED ? Interpret addr exactly. ?

??

It is implementation-defined whether MAP_FIXED shall be supported. MAP_FIXED shall be supported on XSI-conformant systems.

MAP_SHARED and MAP_PRIVATE describe the disposition of write references to the memory object. If MAP_SHARED is specified, write references

shall change the underlying object. If MAP_PRIVATE is specified, modi?
fications to the mapped data by the calling process shall be visible
only to the calling process and shall not change the underlying object.
It is unspecified whether modifications to the underlying object done
after the MAP_PRIVATE mapping is established are visible through the
MAP_PRIVATE mapping. Either MAP_SHARED or MAP_PRIVATE can be specified,
but not both. The mapping type is retained across fork().
The state of synchronization objects such as mutexes, semaphores, bar?
riers, and conditional variables placed in shared memory mapped with
MAP_SHARED becomes undefined when the last region in any process con?
taining the synchronization object is unmapped.
When fildes represents a typed memory object opened with either the
POSIX_TYPED_MEM_ALLOCATE flag or the POSIX_TYPED_MEM_ALLOCATE_CONTIG
flag, mmap() shall, if there are enough resources available, map len
bytes allocated from the corresponding typed memory object which were
not previously allocated to any process in any processor that may ac?
cess that typed memory object. If there are not enough resources avail?
able, the function shall fail. If fildes represents a typed memory ob?
ject opened with the POSIX_TYPED_MEM_ALLOCATE_CONTIG flag, these allo?
cated bytes shall be contiguous within the typed memory object. If
fildes represents a typed memory object opened with the
POSIX_TYPED_MEM_ALLOCATE flag, these allocated bytes may be composed of
non-contiguous fragments within the typed memory object. If fildes rep?
resents a typed memory object opened with neither the
POSIX_TYPED_MEM_ALLOCATE_CONTIG flag nor the POSIX_TYPED_MEM_ALLOCATE
flag, len bytes starting at offset off within the typed memory object
are mapped, exactly as when mapping a file or shared memory object. In
this case, if two processes map an area of typed memory using the same
off and len values and using file descriptors that refer to the same
memory pool (either from the same port or from a different port), both
processes shall map the same region of storage.
When MAP_FIXED is set in the flags argument, the implementation is in?
formed that the value of pa shall be addr, exactly. If MAP_FIXED is

set, mmap() may return MAP_FAILED and set errno to [EINVAL]. If a MAP_FIXED request is successful, then any previous mappings or memory locks for those whole pages containing any part of the address range [pa,pa+len) shall be removed, as if by an appropriate call to munmap(), before the new mapping is established.

When MAP_FIXED is not set, the implementation uses addr in an implementation-defined manner to arrive at pa. The pa so chosen shall be an area of the address space that the implementation deems suitable for a mapping of len bytes to the file. All implementations interpret an addr value of 0 as granting the implementation complete freedom in selecting pa, subject to constraints described below. A non-zero value of addr is taken to be a suggestion of a process address near which the mapping should be placed. When the implementation selects a value for pa, it never places a mapping at address 0, nor does it replace any extant mapping.

If MAP_FIXED is specified and addr is non-zero, it shall have the same remainder as the off parameter, modulo the page size as returned by sysconf() when passed _SC_PAGESIZE or _SC_PAGE_SIZE. The implementation may require that off is a multiple of the page size. If MAP_FIXED is specified, the implementation may require that addr is a multiple of the page size. The system performs mapping operations over whole pages. Thus, while the parameter len need not meet a size or alignment constraint, the system shall include, in any mapping operation, any partial page specified by the address range starting at pa and continuing for len bytes.

The system shall always zero-fill any partial page at the end of an object. Further, the system shall never write out any modified portions of the last page of an object which are beyond its end. References within the address range starting at pa and continuing for len bytes to whole pages following the end of an object shall result in delivery of a SIGBUS signal.

An implementation may generate SIGBUS signals when a reference would cause an error in the mapped object, such as out-of-space condition.

The mmap() function shall add an extra reference to the file associated with the file descriptor fildes which is not removed by a subsequent close() on that file descriptor. This reference shall be removed when there are no more mappings to the file.

The last data access timestamp of the mapped file may be marked for up? date at any time between the mmap() call and the corresponding munmap() call. The initial read or write reference to a mapped region shall cause the file's last data access timestamp to be marked for update if it has not already been marked for update.

The last data modification and last file status change timestamps of a file that is mapped with MAP_SHARED and PROT_WRITE shall be marked for update at some point in the interval between a write reference to the mapped region and the next call to msync() with MS_ASYNC or MS_SYNC for that portion of the file by any process. If there is no such call and if the underlying file is modified as a result of a write reference, then these timestamps shall be marked for update at some time after the write reference.

There may be implementation-defined limits on the number of memory re? gions that can be mapped (per process or per system).

If such a limit is imposed, whether the number of memory regions that can be mapped by a process is decreased by the use of shmat() is imple? mentation-defined.

If mmap() fails for reasons other than [EBADF], [EINVAL], or [ENOTSUP], some of the mappings in the address range starting at addr and continu? ing for len bytes may have been unmapped.

RETURN VALUE

Upon successful completion, the mmap() function shall return the ad? dress at which the mapping was placed (pa); otherwise, it shall return a value of MAP_FAILED and set errno to indicate the error. The symbol MAP_FAILED is defined in the <sys/mman.h> header. No successful return from mmap() shall return the value MAP_FAILED.

ERRORS

The mmap() function shall fail if:

EACCES The fildes argument is not open for read, regardless of the pro?
tection specified, or fildes is not open for write and
PROT_WRITE was specified for a MAP_SHARED type mapping.

EAGAIN The mapping could not be locked in memory, if required by mlock?
all(), due to a lack of resources.

EBADF The fildes argument is not a valid open file descriptor.

EINVAL The value of len is zero.

EINVAL The value of flags is invalid (neither MAP_PRIVATE nor
MAP_SHARED is set).

EMFILE The number of mapped regions would exceed an implementation-de?
fined limit (per process or per system).

ENODEV The fildes argument refers to a file whose type is not supported
by mmap().

ENOMEM MAP_FIXED was specified, and the range [addr,addr+len) exceeds
that allowed for the address space of a process; or, if
MAP_FIXED was not specified and there is insufficient room in
the address space to effect the mapping.

ENOMEM The mapping could not be locked in memory, if required by mlock?
all(), because it would require more space than the system is
able to supply.

ENOMEM Not enough unallocated memory resources remain in the typed mem?
ory object designated by fildes to allocate len bytes.

ENOTSUP
MAP_FIXED or MAP_PRIVATE was specified in the flags argument and
the implementation does not support this functionality.
The implementation does not support the combination of ac?
cesses requested in the prot argument.

ENXIO Addresses in the range [off,off+len) are invalid for the object
specified by fildes.

ENXIO MAP_FIXED was specified in flags and the combination of addr,
len, and off is invalid for the object specified by fildes.

ENXIO The fildes argument refers to a typed memory object that is not
accessible from the calling process.

EOVERFLOW

The file is a regular file and the value of off plus len exceeds the offset maximum established in the open file description as? sociated with fildes.

The mmap() function may fail if:

EINVAL The addr argument (if MAP_FIXED was specified) or off is not a multiple of the page size as returned by sysconf(), or is con? sidered invalid by the implementation.

The following sections are informative.

EXAMPLES

None.

APPLICATION USAGE

Use of mmap() may reduce the amount of memory available to other memory allocation functions.

Use of MAP_FIXED may result in unspecified behavior in further use of malloc() and shmat(). The use of MAP_FIXED is discouraged, as it may prevent an implementation from making the most effective use of re? sources. Most implementations require that off and addr are multiples of the page size as returned by sysconf().

The application must ensure correct synchronization when using mmap() in conjunction with any other file access method, such as read() and write(), standard input/output, and shmat().

The mmap() function allows access to resources via address space manip? ulations, instead of read()/write(). Once a file is mapped, all a process has to do to access it is use the data at the address to which the file was mapped. So, using pseudo-code to illustrate the way in which an existing program might be changed to use mmap(), the follow? ing:

```
fildes = open(...)
lseek(fildes, some_offset)
read(fildes, buf, len)
/* Use data in buf. */
```

becomes:

```
fildes = open(...)
address = mmap(0, len, PROT_READ, MAP_PRIVATE, fildes, some_offset)
/* Use data at address. */
```

RATIONALE

After considering several other alternatives, it was decided to adopt the mmap() definition found in SVR4 for mapping memory objects into process address spaces. The SVR4 definition is minimal, in that it de? scribes only what has been built, and what appears to be necessary for a general and portable mapping facility.

Note that while mmap() was first designed for mapping files, it is ac? tually a general-purpose mapping facility. It can be used to map any appropriate object, such as memory, files, devices, and so on, into the address space of a process.

When a mapping is established, it is possible that the implementation may need to map more than is requested into the address space of the process because of hardware requirements. An application, however, can? not count on this behavior. Implementations that do not use a paged ar? chitecture may simply allocate a common memory region and return the address of it; such implementations probably do not allocate any more than is necessary. References past the end of the requested area are unspecified.

If an application requests a mapping that overlaps existing mappings in the process, it might be desirable that an implementation detect this and inform the application. However, if the program specifies a fixed address mapping (which requires some implementation knowledge to deter? mine a suitable address, if the function is supported at all), then the program is presumed to be successfully managing its own address space and should be trusted when it asks to map over existing data struc? tures. Furthermore, it is also desirable to make as few system calls as possible, and it might be considered onerous to require an munmap() be? fore an mmap() to the same address range. This volume of POSIX.1?2017 specifies that the new mapping replaces any existing mappings (implying an automatic munmap() on the address range), following existing prac?

tice in this regard. The standard developers also considered whether there should be a way for new mappings to overlay existing mappings, but found no existing practice for this.

It is not expected that all hardware implementations are able to sup? port all combinations of permissions at all addresses. Implementations are required to disallow write access to mappings without write permis? sion and to disallow access to mappings without any access permission. Other than these restrictions, implementations may allow access types other than those requested by the application. For example, if the ap? plication requests only PROT_WRITE, the implementation may also allow read access. A call to mmap() fails if the implementation cannot sup? port allowing all the access requested by the application. For example, some implementations cannot support a request for both write access and execute access simultaneously. All implementations must support re? quests for no access, read access, write access, and both read and write access. Strictly conforming code must only rely on the required checks. These restrictions allow for portability across a wide range of hardware.

The MAP_FIXED address treatment is likely to fail for non-page-aligned values and for certain architecture-dependent address ranges. Conform? ing implementations cannot count on being able to choose address values for MAP_FIXED without utilizing non-portable, implementation-defined knowledge. Nonetheless, MAP_FIXED is provided as a standard interface conforming to existing practice for utilizing such knowledge when it is available.

Similarly, in order to allow implementations that do not support vir? tual addresses, support for directly specifying any mapping addresses via MAP_FIXED is not required and thus a conforming application may not count on it.

The MAP_PRIVATE function can be implemented efficiently when memory protection hardware is available. When such hardware is not available, implementations can implement such ``mappings'' by simply making a real copy of the relevant data into process private memory, though this

tends to behave similarly to read().

The function has been defined to allow for many different models of us? ing shared memory. However, all uses are not equally portable across all machine architectures. In particular, the mmap() function allows the system as well as the application to specify the address at which to map a specific region of a memory object. The most portable way to use the function is always to let the system choose the address, speci? fying NULL as the value for the argument addr and not to specify MAP_FIXED.

If it is intended that a particular region of a memory object be mapped at the same address in a group of processes (on machines where this is even possible), then MAP_FIXED can be used to pass in the desired map? ping address. The system can still be used to choose the desired ad? dress if the first such mapping is made without specifying MAP_FIXED, and then the resulting mapping address can be passed to subsequent pro? cesses for them to pass in via MAP_FIXED. The availability of a spe? cific address range cannot be guaranteed, in general.

The mmap() function can be used to map a region of memory that is larger than the current size of the object. Memory access within the mapping but beyond the current end of the underlying objects may result in SIGBUS signals being sent to the process. The reason for this is that the size of the object can be manipulated by other processes and can change at any moment. The implementation should tell the applica? tion that a memory reference is outside the object where this can be detected; otherwise, written data may be lost and read data may not re? flect actual data in the object.

Note that references beyond the end of the object do not extend the ob? ject as the new end cannot be determined precisely by most virtual mem? ory hardware. Instead, the size can be directly manipulated by ftrun? cate().

Process memory locking does apply to shared memory regions, and the MCL_FUTURE argument to mlockall() can be relied upon to cause new shared memory regions to be automatically locked.

Existing implementations of mmap() return the value -1 when unsuccess?
ful. Since the casting of this value to type void * cannot be guaran?
teed by the ISO C standard to be distinct from a successful value, this
volume of POSIX.1?2017 defines the symbol MAP_FAILED, which a conform?
ing implementation does not return as the result of a successful call.

FUTURE DIRECTIONS

None.

SEE ALSO

exec, fcntl(), fork(), lockf(), msync(), munmap(), mprotect(),
posix_typed_mem_open(), shmat(), sysconf()
The Base Definitions volume of POSIX.1?2017, <sys_mman.h>

COPYRIGHT