Red Hat Enterprise Linux Release 9.2 Manual Pages on 'munmap.2' command

$ man munmap.2

MMAP(2)   Linux Programmer's Manual   MMAP(2)

NAME

   mmap, 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);

   int munmap(void *addr, size_t length);

   See NOTES for information on feature test macro requirements.

DESCRIPTION

   mmap() creates a new mapping in the virtual address space of the call?

   ing process. The starting address for the new mapping is specified in

   addr.   The length argument specifies the length of the mapping (which

   must be greater than 0).

   If addr is NULL, then the kernel chooses the (page-aligned) address at

   which to create the mapping; this is the most portable method of creat?

   ing 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 kernel will pick a

   nearby page boundary (but always above or equal to the value specified

   by /proc/sys/vm/mmap_min_addr) and attempt to create the mapping there.

   If another mapping already exists there, the kernel picks a new address

   that may or may not depend on the hint. 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 de?

scriptor fd. offset must be a multiple of the page size as returned by

sysconf(_SC_PAGE_SIZE).

After the mmap() call has returned, the file descriptor, fd, can be

closed immediately without invalidating the mapping.

The prot argument describes the desired memory protection of the map?

ping (and must not conflict with the open mode of the file). It is ei?

ther 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

The flags argument determines whether updates to the mapping are visi?

ble 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 mapping the same region, and (in the case of file-

   backed mappings) are carried through to the underlying file.

   (To precisely control when updates are carried through to the

   underlying file requires the use of msync(2).)

MAP_SHARED_VALIDATE (since Linux 4.15)

   This flag provides the same behavior as MAP_SHARED except that

   MAP_SHARED mappings ignore unknown flags in flags. By contrast,

   when creating a mapping using MAP_SHARED_VALIDATE, the kernel

   verifies all passed flags are known and fails the mapping with

   the error EOPNOTSUPP for unknown flags. This mapping type is

   also required to be able to use some mapping flags (e.g.,

   MAP_SYNC).

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 unspeci?

    fied whether changes made to the file after the mmap() call are

    visible in the mapped region.

Both MAP_SHARED and MAP_PRIVATE are described in POSIX.1-2001 and

POSIX.1-2008. MAP_SHARED_VALIDATE is a Linux extension.

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 ad?

    dress space. This flag is supported only on x86-64, for 64-bit

    programs.   It was added to allow thread stacks to be allocated

    somewhere in the first 2 GB 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; provided for compatibility with other

    implementations.

MAP_ANONYMOUS

    The mapping is not backed by any file; its contents are initial?

    ized to zero. The fd argument is ignored; however, some imple?

    mentations require fd to be -1 if MAP_ANONYMOUS (or MAP_ANON) is

    specified, and portable applications should ensure this.   The

    offset argument should be zero. The use of MAP_ANONYMOUS in

    conjunction with MAP_SHARED is supported on Linux only since

    kernel 2.4.

MAP_DENYWRITE

    This flag is ignored. (Long ago?Linux 2.0 and earlier?it sig?

    naled that attempts to write to the underlying file should fail

    with ETXTBSY.   But this was a source of denial-of-service at?

tacks.)

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 suitably aligned: for most architec?

tures a multiple of the page size is sufficient; however, some

architectures may impose additional restrictions. 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.

Software that aspires to be portable should use the MAP_FIXED

flag with care, keeping in mind that the exact layout of a

process's memory mappings is allowed to change significantly be?

tween kernel versions, C library versions, and operating system

releases. Carefully read the discussion of this flag in NOTES!

MAP_FIXED_NOREPLACE (since Linux 4.17)

This flag provides behavior that is similar to MAP_FIXED with

respect to the addr enforcement, but differs in that

MAP_FIXED_NOREPLACE never clobbers a preexisting mapped range.

If the requested range would collide with an existing mapping,

then this call fails with the error EEXIST. This flag can

therefore be used as a way to atomically (with respect to other

threads) attempt to map an address range: one thread will suc?

ceed; all others will report failure.

Note that older kernels which do not recognize the

MAP_FIXED_NOREPLACE flag will typically (upon detecting a colli?

sion with a preexisting mapping) fall back to a "non-MAP_FIXED"

type of behavior: they will return an address that is different

from the requested address. Therefore, backward-compatible   *Page 4/15*

software should check the returned address against the requested address.

MAP_GROWSDOWN

This flag is used for stacks. It indicates to the kernel vir?
tual memory system that the mapping should extend downward in
memory. The return address is one page lower than the memory
area that is actually created in the process's virtual address
space. Touching an address in the "guard" page below the map?
ping will cause the mapping to grow by a page. This growth can
be repeated until the mapping grows to within a page of the high
end of the next lower mapping, at which point touching the
"guard" page will result in a SIGSEGV signal.

MAP_HUGETLB (since Linux 2.6.32)

Allocate the mapping using "huge pages." See the Linux kernel
source   file Documentation/admin-guide/mm/hugetlbpage.rst for
further information, as well as NOTES, below.

MAP_HUGE_2MB, MAP_HUGE_1GB (since Linux 3.8)

Used in conjunction with MAP_HUGETLB to select alternative
hugetlb page sizes (respectively, 2 MB and 1 GB) on systems that
support multiple hugetlb page sizes.

More generally, the desired huge page size can be configured by
encoding the base-2 logarithm of the desired page size in the
six bits at the offset MAP_HUGE_SHIFT. (A value of zero in this
bit field provides the default huge page size; the default huge
page size can be discovered via the Hugepagesize field exposed
by /proc/meminfo.)   Thus, the above two constants are defined
as:

    #define MAP_HUGE_2MB   (21 << MAP_HUGE_SHIFT)

    #define MAP_HUGE_1GB   (30 << MAP_HUGE_SHIFT)

The range of huge page sizes that are supported by the system
can be discovered by listing the subdirectories in /sys/ker?
nel/mm/hugepages.

MAP_LOCKED (since Linux 2.5.37)

Mark the mapped region to be locked in the same way as mlock(2). This implementation will try to populate (prefault) the whole range but the mmap() call doesn't fail with ENOMEM if this fails. Therefore major faults might happen later on. So the semantic is not as strong as mlock(2). One should use mmap() plus mlock(2) when major faults are not acceptable after the initialization of the mapping. The MAP_LOCKED flag is ignored in older kernels.

MAP_NONBLOCK (since Linux 2.5.46)

This flag is meaningful only in conjunction with MAP_POPULATE. Don't perform read-ahead: create page tables entries only for pages that are already present in RAM. Since Linux 2.6.23, this flag causes MAP_POPULATE to do nothing. One day, the combina‐ tion 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 had effect only for private writable mappings.

MAP_POPULATE (since Linux 2.5.46)

Populate (prefault) page tables for a mapping. For a file map‐ ping, this causes read-ahead on the file. This will help to re‐ duce blocking on page faults later. MAP_POPULATE is supported for private mappings only since Linux 2.6.23.

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 on Linux. However, by employing this flag, applications can ensure that they transparently ob‐ tain support if the flag is implemented in the future. Thus, it

is used in the glibc threading implementation to allow for the
fact that some architectures may (later) require special treat?
ment for stack allocations. A further reason to employ this
flag is portability: MAP_STACK exists (and has an effect) on
some other systems (e.g., some of the BSDs).

MAP_SYNC (since Linux 4.15)

This flag is available only with the MAP_SHARED_VALIDATE mapping
type; mappings of type MAP_SHARED will silently ignore this
flag. This flag is supported only for files supporting DAX (di?
rect mapping of persistent memory). For other files, creating a
mapping with this flag results in an EOPNOTSUPP error.
Shared file mappings with this flag provide the guarantee that
while some memory is mapped writable in the address space of the
process, it will be visible in the same file at the same offset
even after the system crashes or is rebooted. In conjunction
with the use of appropriate CPU instructions, this provides
users of such mappings with a more efficient way of making data
modifications persistent.

MAP_UNINITIALIZED (since Linux 2.6.33)

Don't clear anonymous pages. This flag is intended to improve
performance on embedded devices. This flag is honored only if
the kernel was configured with the CONFIG_MMAP_ALLOW_UNINITIAL?
IZED option. Because of the security implications, that option
is normally enabled only on embedded devices (i.e., devices
where one has complete control of the contents of user memory).

Of the above flags, only MAP_FIXED is specified in POSIX.1-2001 and
POSIX.1-2008. However, most systems also support MAP_ANONYMOUS (or its
synonym MAP_ANON).

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 (but length need

not be). All pages containing a part of the indicated range are un?

mapped, and subsequent references to these pages will generate SIGSEGV.

It is not an error if the indicated range does not contain any mapped

pages.

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

to indicate the cause of the error.

On success, munmap() returns 0. On failure, it returns -1, and errno

is set to indicate the cause of the error (probably to EINVAL).

ERRORS

EACCES A file descriptor refers to a non-regular file. Or a file map?

ping 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).

EEXIST MAP_FIXED_NOREPLACE was specified in flags, and the range cov?

ered by addr and length clashes with an existing mapping.

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 none of MAP_PRIVATE, MAP_SHARED or

MAP_SHARED_VALIDATE.

ENFILE The system-wide limit on the total number of open files has been

reached.

ENODEV The underlying filesystem of the specified file does not support

memory mapping.

ENOMEM No memory is available.

ENOMEM The process's maximum number of mappings would have been ex‐

    ceeded. This error can also occur for munmap(), when unmapping

    a region in the middle of an existing mapping, since this re‐

    sults in two smaller mappings on either side of the region being

    unmapped.

ENOMEM (since Linux 4.7) The process's RLIMIT_DATA limit, described in

    getrlimit(2), would have been exceeded.

EOVERFLOW

    On 32-bit architecture together with the large file extension

    (i.e., using 64-bit off_t): the number of pages used for length

    plus number of pages used for offset would overflow unsigned

    long (32 bits).

EPERM The prot argument asks for PROT_EXEC but the mapped area belongs

    to a file on a filesystem that was mounted no-exec.

EPERM The operation was prevented by a file seal; see fcntl(2).

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 page of the buffer that lies beyond the

    end of the mapped file. For an explanation of the treatment of

    the bytes in the page that corresponds to the end of a mapped

    file that is not a multiple of the page size, see NOTES.

ATTRIBUTES

For an explanation of the terms used in this section, see at‐

tributes(7).

```
???????????????????????????????????????????????????
?Interface       ? Attribute    ? Value   ?
???????????????????????????????????????????????????
?mmap(), munmap()   ? Thread safety ? MT-Safe ?
```

???

CONFORMING TO

POSIX.1-2001, POSIX.1-2008, SVr4, 4.4BSD.

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

Memory mapped by mmap() is preserved across fork(2), with the same at?

tributes.

A file is mapped in multiples of the page size. For a file that is not

a multiple of the page size, the remaining bytes in the partial page at

the end of the mapping are zeroed when mapped, and modifications 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 corre?

spond to added or removed regions of the file is unspecified.

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).

Certain flags constants are defined only if suitable feature test

macros are defined (possibly by default): _DEFAULT_SOURCE with glibc

2.19 or later; or _BSD_SOURCE or _SVID_SOURCE in glibc 2.19 and ear?

lier. (Employing _GNU_SOURCE also suffices, and requiring that macro

specifically would have been more logical, since these flags are all

Linux-specific.) The relevant flags are: MAP_32BIT, MAP_ANONYMOUS (and

the synonym MAP_ANON), MAP_DENYWRITE, MAP_EXECUTABLE, MAP_FILE,

MAP_GROWSDOWN, MAP_HUGETLB, MAP_LOCKED, MAP_NONBLOCK, MAP_NORESERVE,

MAP_POPULATE, and MAP_STACK.

An application can determine which pages of a mapping are currently resident in the buffer/page cache using mincore(2).

Using MAP_FIXED safely

The only safe use for MAP_FIXED is where the address range specified by addr and length was previously reserved using another mapping; other‐wise, the use of MAP_FIXED is hazardous because it forcibly removes preexisting mappings, making it easy for a multithreaded process to corrupt its own address space.

For example, suppose that thread A looks through /proc/<pid>/maps in order to locate an unused address range that it can map using MAP_FIXED, while thread B simultaneously acquires part or all of that same address range. When thread A subsequently employs mmap(MAP_FIXED), it will effectively clobber the mapping that thread B created. In this scenario, thread B need not create a mapping di‐rectly; simply making a library call that, internally, uses dlopen(3) to load some other shared library, will suffice. The dlopen(3) call will map the library into the process's address space. Furthermore, almost any library call may be implemented in a way that adds memory mappings to the address space, either with this technique, or by simply allocating memory. Examples include brk(2), malloc(3), pthread_cre‐ate(3), and the PAM libraries ?http://www.linux-pam.org?.

Since Linux 4.17, a multithreaded program can use the MAP_FIXED_NORE‐PLACE flag to avoid the hazard described above when attempting to cre‐ate a mapping at a fixed address that has not been reserved by a preex‐isting mapping.

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 be‐

fore a subsequent msync(2) with the MS_SYNC or MS_ASYNC flag, if one
occurs.

Huge page (Huge TLB) mappings

For mappings that employ huge pages, the requirements for the arguments
of mmap() and munmap() differ somewhat from the requirements for map‐
pings that use the native system page size.

For mmap(), offset must be a multiple of the underlying huge page size.

The system automatically aligns length to be a multiple of the underly‐
ing huge page size.

For munmap(), addr, and length must both be a multiple of the underly‐
ing huge page size.

C library/kernel differences

This page describes the interface provided by the glibc mmap() wrapper
function.   Originally, this function invoked a system call of the same
name. Since kernel 2.4, that system call has been superseded by
mmap2(2), and nowadays the glibc mmap() wrapper function invokes
mmap2(2) with a suitably adjusted value for offset.

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 has effect only 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.

POSIX specifies that the system shall always zero fill any partial page
at the end of the object and that system will never write any modifica‐
tion of the object beyond its end. On Linux, when you write data to
such partial page after the end of the object, the data stays in the
page cache even after the file is closed and unmapped and even though
the data is never written to the file itself, subsequent mappings may

see the modified content. In some cases, this could be fixed by call?

ing msync(2) before the unmap takes place; however, this doesn't work

on tmpfs(5) (for example, when using the POSIX shared memory interface

documented in shm_overview(7)).

EXAMPLES

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 de?

sired bytes.

Program source

```
#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);

    }
```

```c
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);

}

munmap(addr, length + offset - pa_offset);

close(fd);
```

```
            exit(EXIT_SUCCESS);

    }

SEE ALSO

    ftruncate(2), getpagesize(2),  memfd_create(2),  mincore(2),  mlock(2),

    mmap2(2),  mprotect(2), mremap(2), msync(2), remap_file_pages(2), setr?

    limit(2), shmat(2), userfaultfd(2), shm_open(3), shm_overview(7)

    The descriptions of the following files in  proc(5):  /proc/[pid]/maps,

    /proc/[pid]/map_files, and /proc/[pid]/smaps.

    B.O. Gallmeister, POSIX.4, O'Reilly, pp. 128?129 and 389?391.

COLOPHON

    This  page  is  part of release 5.10 of the Linux man-pages project.  A

    description of the project, information about reporting bugs,  and  the

    latest    version    of    this    page,    can    be    found    at

    https://www.kernel.org/doc/man-pages/.

Linux                         2020-12-21                         MMAP(2)
```