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

$ man mlockall.2

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

NAME

    mlock, mlock2, munlock, mlockall, munlockall - lock and unlock memory

SYNOPSIS

    #include <sys/mman.h>

    int mlock(const void *addr, size_t len);

    int mlock2(const void *addr, size_t len, int flags);

    int munlock(const void *addr, size_t len);

    int mlockall(int flags);

    int munlockall(void);

DESCRIPTION

    mlock(), mlock2(), and mlockall() lock part or all of the calling

    process's virtual address space into RAM, preventing that memory from

    being paged to the swap area.

    munlock() and munlockall() perform the converse operation, unlocking

    part or all of the calling process's virtual address space, so that

    pages in the specified virtual address range may once more to be

    swapped out if required by the kernel memory manager.

    Memory locking and unlocking are performed in units of whole pages.

  mlock(), mlock2(), and munlock()

    mlock() locks pages in the address range starting at addr and continu?

    ing for len bytes. All pages that contain a part of the specified ad?

    dress range are guaranteed to be resident in RAM when the call returns

successfully; the pages are guaranteed to stay in RAM until later un?

locked.

mlock2() also locks pages in the specified range starting at addr and

continuing for len bytes. However, the state of the pages contained in

that range after the call returns successfully will depend on the value

in the flags argument.

The flags argument can be either 0 or the following constant:

MLOCK_ONFAULT

Lock pages that are currently resident and mark the entire range

so that the remaining nonresident pages are locked when they are

populated by a page fault.

If flags is 0, mlock2() behaves exactly the same as mlock().

munlock() unlocks pages in the address range starting at addr and con?

tinuing for len bytes. After this call, all pages that contain a part

of the specified memory range can be moved to external swap space again

by the kernel.

mlockall() and munlockall()

mlockall() locks all pages mapped into the address space of the calling

process. This includes the pages of the code, data and stack segment,

as well as shared libraries, user space kernel data, shared memory, and

memory-mapped files. All mapped pages are guaranteed to be resident in

RAM when the call returns successfully; the pages are guaranteed to

stay in RAM until later unlocked.

The flags argument is constructed as the bitwise OR of one or more of

the following constants:

MCL_CURRENT

Lock all pages which are currently mapped into the address space

of the process.

MCL_FUTURE

Lock all pages which will become mapped into the address space

of the process in the future. These could be, for instance, new

pages required by a growing heap and stack as well as new mem?

ory-mapped files or shared memory regions.

MCL_ONFAULT (since Linux 4.4)

      Used together with MCL_CURRENT, MCL_FUTURE, or both. Mark all

      current (with MCL_CURRENT) or future (with MCL_FUTURE) mappings

      to lock pages when they are faulted in. When used with MCL_CUR?

      RENT, all present pages are locked, but mlockall() will not

      fault in non-present pages. When used with MCL_FUTURE, all fu?

      ture mappings will be marked to lock pages when they are faulted

      in, but they will not be populated by the lock when the mapping

      is created. MCL_ONFAULT must be used with either MCL_CURRENT or

      MCL_FUTURE or both.

If MCL_FUTURE has been specified, then a later system call (e.g.,

mmap(2), sbrk(2), malloc(3)), may fail if it would cause the number of

locked bytes to exceed the permitted maximum (see below). In the same

circumstances, stack growth may likewise fail: the kernel will deny

stack expansion and deliver a SIGSEGV signal to the process.

munlockall() unlocks all pages mapped into the address space of the

calling process.

RETURN VALUE

On success, these system calls return 0. On error, -1 is returned, er?

rno is set appropriately, and no changes are made to any locks in the

address space of the process.

ERRORS

ENOMEM (Linux 2.6.9 and later) the caller had a nonzero RLIMIT_MEMLOCK

      soft resource limit, but tried to lock more memory than the

      limit permitted.   This limit is not enforced if the process is

      privileged (CAP_IPC_LOCK).

ENOMEM (Linux 2.4 and earlier) the calling process tried to lock more

      than half of RAM.

EPERM The caller is not privileged, but needs privilege (CAP_IPC_LOCK)

      to perform the requested operation.

For mlock(), mlock2(), and munlock():

EAGAIN Some or all of the specified address range could not be locked.

EINVAL The result of the addition addr+len was less than addr (e.g.,

the addition may have resulted in an overflow).

EINVAL (Not on Linux) addr was not a multiple of the page size.

ENOMEM Some of the specified address range does not correspond to mapped pages in the address space of the process.

ENOMEM Locking or unlocking a region would result in the total number of mappings with distinct attributes (e.g., locked versus un‐ locked) exceeding the allowed maximum. (For example, unlocking a range in the middle of a currently locked mapping would result in three mappings: two locked mappings at each end and an un‐ locked mapping in the middle.)

For mlock2():

EINVAL Unknown flags were specified.

For mlockall():

EINVAL Unknown flags were specified or MCL_ONFAULT was specified with‐ out either MCL_FUTURE or MCL_CURRENT.

For munlockall():

EPERM (Linux 2.6.8 and earlier) The caller was not privileged (CAP_IPC_LOCK).

VERSIONS

mlock2() is available since Linux 4.4; glibc support was added in ver‐ sion 2.27.

CONFORMING TO

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

mlock2() is Linux specific.

On POSIX systems on which mlock() and munlock() are available, _POSIX_MEMLOCK_RANGE is defined in <unistd.h> and the number of bytes in a page can be determined from the constant PAGESIZE (if defined) in <limits.h> or by calling sysconf(_SC_PAGESIZE).

On POSIX systems on which mlockall() and munlockall() are available, _POSIX_MEMLOCK is defined in <unistd.h> to a value greater than 0. (See also sysconf(3).)

NOTES

Memory locking has two main applications: real-time algorithms and

high-security data processing. Real-time applications require deter?
ministic timing, and, like scheduling, paging is one major cause of un?
expected program execution delays. Real-time applications will usually
also switch to a real-time scheduler with sched_setscheduler(2). Cryp?
tographic security software often handles critical bytes like passwords
or secret keys as data structures. As a result of paging, these se?
crets could be transferred onto a persistent swap store medium, where
they might be accessible to the enemy long after the security software
has erased the secrets in RAM and terminated. (But be aware that the
suspend mode on laptops and some desktop computers will save a copy of
the system's RAM to disk, regardless of memory locks.)

Real-time processes that are using mlockall() to prevent delays on page
faults should reserve enough locked stack pages before entering the
time-critical section, so that no page fault can be caused by function
calls. This can be achieved by calling a function that allocates a
sufficiently large automatic variable (an array) and writes to the mem?
ory occupied by this array in order to touch these stack pages. This
way, enough pages will be mapped for the stack and can be locked into
RAM. The dummy writes ensure that not even copy-on-write page faults
can occur in the critical section.

Memory locks are not inherited by a child created via fork(2) and are
automatically removed (unlocked) during an execve(2) or when the
process terminates. The mlockall() MCL_FUTURE and MCL_FUTURE | MCL_ON?
FAULT settings are not inherited by a child created via fork(2) and are
cleared during an execve(2).

Note that fork(2) will prepare the address space for a copy-on-write
operation. The consequence is that any write access that follows will
cause a page fault that in turn may cause high latencies for a real-
time process. Therefore, it is crucial not to invoke fork(2) after an
mlockall() or mlock() operation?not even from a thread which runs at a
low priority within a process which also has a thread running at ele?
vated priority.

The memory lock on an address range is automatically removed if the ad?

dress range is unmapped via munmap(2).

Memory locks do not stack, that is, pages which have been locked sev‐
eral times by calls to mlock(), mlock2(), or mlockall() will be un‐
locked by a single call to munlock() for the corresponding range or by
munlockall(). Pages which are mapped to several locations or by sev‐
eral processes stay locked into RAM as long as they are locked at least
at one location or by at least one process.

If a call to mlockall() which uses the MCL_FUTURE flag is followed by
another call that does not specify this flag, the changes made by the
MCL_FUTURE call will be lost.

The mlock2() MLOCK_ONFAULT flag and the mlockall() MCL_ONFAULT flag al‐
low efficient memory locking for applications that deal with large map‐
pings where only a (small) portion of pages in the mapping are touched.
In such cases, locking all of the pages in a mapping would incur a sig‐
nificant penalty for memory locking.

Linux notes

Under Linux, mlock(), mlock2(), and munlock() automatically round addr
down to the nearest page boundary. However, the POSIX.1 specification
of mlock() and munlock() allows an implementation to require that addr
is page aligned, so portable applications should ensure this.

The VmLck field of the Linux-specific /proc/[pid]/status file shows how
many kilobytes of memory the process with ID PID has locked using
mlock(), mlock2(), mlockall(), and mmap(2) MAP_LOCKED.

Limits and permissions

In Linux 2.6.8 and earlier, a process must be privileged (CAP_IPC_LOCK)
in order to lock memory and the RLIMIT_MEMLOCK soft resource limit de‐
fines a limit on how much memory the process may lock.

Since Linux 2.6.9, no limits are placed on the amount of memory that a
privileged process can lock and the RLIMIT_MEMLOCK soft resource limit
instead defines a limit on how much memory an unprivileged process may
lock.

BUGS

In Linux 4.8 and earlier, a bug in the kernel's accounting of locked

memory for unprivileged processes (i.e., without CAP_IPC_LOCK) meant that if the region specified by addr and len overlapped an existing lock, then the already locked bytes in the overlapping region were counted twice when checking against the limit. Such double accounting could incorrectly calculate a "total locked memory" value for the process that exceeded the RLIMIT_MEMLOCK limit, with the result that mlock() and mlock2() would fail on requests that should have succeeded. This bug was fixed in Linux 4.9.

In the 2.4 series Linux kernels up to and including 2.4.17, a bug caused the mlockall() MCL_FUTURE flag to be inherited across a fork(2). This was rectified in kernel 2.4.18.

Since kernel 2.6.9, if a privileged process calls mlockall(MCL_FUTURE) and later drops privileges (loses the CAP_IPC_LOCK capability by, for example, setting its effective UID to a nonzero value), then subsequent memory allocations (e.g., mmap(2), brk(2)) will fail if the RLIMIT_MEM? LOCK resource limit is encountered.

SEE ALSO

mincore(2), mmap(2), setrlimit(2), shmctl(2), sysconf(3), proc(5), ca? pabilities(7)

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   2020-04-11   MLOCK(2)