NAME

mlock, munlock, mlockall, munlockall − lock and unlock memory

SYNOPSIS

#include <sys/mman.h>

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

int mlockall(int *flags***);**
int munlockall(void);

DESCRIPTION

mlock() and **mlockall**() respectively 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, respectively 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() and munlock()

mlock() locks pages in the address range starting at *addr* and continuing for *len* bytes. All pages that contain a part of the specified address range are guaranteed to be resident in RAM when the call returns successfully; the pages are guaranteed to stay in RAM until later unlocked.

munlock() unlocks pages in the address range starting at *addr* and continuing 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 memory mapped files or shared memory regions.

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, *errno* 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 non-zero **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
(Linux 2.6.9 and later) the caller was not privileged (**CAP_IPC_LOCK**) and its **RLIMIT_MEMLOCK** soft resource limit was 0.

EPERM
(Linux 2.6.8 and earlier) The calling process has insufficient privilege to call **munlockall**(). Under Linux the **CAP_IPC_LOCK** capability is required.

For **mlock**() and **munlock**():

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

EINVAL
len was negative.

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.

For **mlockall**():

EINVAL
Unknown *flags* were specified.

For **munlockall**():

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

CONFORMING TO

POSIX.1-2001, SVr4.

AVAILABILITY

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 deterministic timing, and, like scheduling, paging is one major cause of unexpected program execution delays. Real-time applications will usually also switch to a real-time scheduler with **sched_setscheduler**(2). Cryptographic security software often handles critical bytes like passwords or secret keys as data structures. As a result of paging, these secrets 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 memory 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 memory lock on an address range is automatically removed if the address range is unmapped via **munmap**(2).

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

Linux Notes

Under Linux, **mlock**() and **munlock**() automatically round *addr* down to the nearest page boundary. However, POSIX.1-2001 allows an implementation to require that *addr* is page aligned, so portable applications should ensure this.

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 defines 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 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 non-zero value), then subsequent memory allocations (e.g., **mmap**(2), **brk**(2)) will fail if the **RLIMIT_MEMLOCK** resource limit is encountered.

SEE ALSO

mmap(2), **setrlimit**(2), **shmctl**(2), **sysconf**(3), **capabilities**(7)

COLOPHON

This page is part of release 3.22 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/.