

NAME

mremap – remap a virtual memory address

LIBRARY

Standard C library (*libc*, *-lc*)

SYNOPSIS

```
#define _GNU_SOURCE      /* See feature_test_macros(7) */
#include <sys/mman.h>

void *mremap(void old_address[old_size], size_t old_size,
             size_t new_size, int flags, ... /* void *new_address */);
```

DESCRIPTION

mremap() expands (or shrinks) an existing memory mapping, potentially moving it at the same time (controlled by the *flags* argument and the available virtual address space).

old_address is the old address of the virtual memory block that you want to expand (or shrink). Note that *old_address* has to be page aligned. *old_size* is the old size of the virtual memory block. *new_size* is the requested size of the virtual memory block after the resize. An optional fifth argument, *new_address*, may be provided; see the description of **MREMAP_FIXED** below.

If the value of *old_size* is zero, and *old_address* refers to a shareable mapping (see **mmap(2)** **MAP_SHARED**), then **mremap()** will create a new mapping of the same pages. *new_size* will be the size of the new mapping and the location of the new mapping may be specified with *new_address*; see the description of **MREMAP_FIXED** below. If a new mapping is requested via this method, then the **MREMAP_MAYMOVE** flag must also be specified.

The *flags* bit-mask argument may be 0, or include the following flags:

MREMAP_MAYMOVE

By default, if there is not sufficient space to expand a mapping at its current location, then **mremap()** fails. If this flag is specified, then the kernel is permitted to relocate the mapping to a new virtual address, if necessary. If the mapping is relocated, then absolute pointers into the old mapping location become invalid (offsets relative to the starting address of the mapping should be employed).

MREMAP_FIXED (since Linux 2.3.31)

This flag serves a similar purpose to the **MAP_FIXED** flag of **mmap(2)**. If this flag is specified, then **mremap()** accepts a fifth argument, *void *new_address*, which specifies a page-aligned address to which the mapping must be moved. Any previous mapping at the address range specified by *new_address* and *new_size* is unmapped.

If **MREMAP_FIXED** is specified, then **MREMAP_MAYMOVE** must also be specified.

MREMAP_DONTUNMAP (since Linux 5.7)

This flag, which must be used in conjunction with **MREMAP_MAYMOVE**, remaps a mapping to a new address but does not unmap the mapping at *old_address*.

The **MREMAP_DONTUNMAP** flag can be used only with private anonymous mappings (see the description of **MAP_PRIVATE** and **MAP_ANONYMOUS** in **mmap(2)**).

After completion, any access to the range specified by *old_address* and *old_size* will result in a page fault. The page fault will be handled by a **userfaultfd(2)** handler if the address is in a range previously registered with **userfaultfd(2)**. Otherwise, the kernel allocates a zero-filled page to handle the fault.

The **MREMAP_DONTUNMAP** flag may be used to atomically move a mapping while leaving the source mapped. See NOTES for some possible applications of **MREMAP_DONTUNMAP**.

If the memory segment specified by *old_address* and *old_size* is locked (using **mlock(2)** or similar), then this lock is maintained when the segment is resized and/or relocated. As a consequence, the amount of memory locked by the process may change.

RETURN VALUE

On success **mremap()** returns a pointer to the new virtual memory area. On error, the value **MAP_FAILED** (that is, *(void *) -1*) is returned, and *errno* is set to indicate the error.

ERRORS

EAGAIN

The caller tried to expand a memory segment that is locked, but this was not possible without exceeding the **RLIMIT_MEMLOCK** resource limit.

EFAULT

Some address in the range *old_address* to *old_address+old_size* is an invalid virtual memory address for this process. You can also get **EFAULT** even if there exist mappings that cover the whole address space requested, but those mappings are of different types.

EINVAL

An invalid argument was given. Possible causes are:

- *old_address* was not page aligned;
- a value other than **MREMAP_MAYMOVE** or **MREMAP_FIXED** or **MREMAP_DONTUNMAP** was specified in *flags*;
- *new_size* was zero;
- *new_size* or *new_address* was invalid;
- the new address range specified by *new_address* and *new_size* overlapped the old address range specified by *old_address* and *old_size*;
- **MREMAP_FIXED** or **MREMAP_DONTUNMAP** was specified without also specifying **MREMAP_MAYMOVE**;
- **MREMAP_DONTUNMAP** was specified, but one or more pages in the range specified by *old_address* and *old_size* were not private anonymous;
- **MREMAP_DONTUNMAP** was specified and *old_size* was not equal to *new_size*;
- *old_size* was zero and *old_address* does not refer to a shareable mapping (but see BUGS);
- *old_size* was zero and the **MREMAP_MAYMOVE** flag was not specified.

ENOMEM

Not enough memory was available to complete the operation. Possible causes are:

- The memory area cannot be expanded at the current virtual address, and the **MREMAP_MAYMOVE** flag is not set in *flags*. Or, there is not enough (virtual) memory available.
- **MREMAP_DONTUNMAP** was used causing a new mapping to be created that would exceed the (virtual) memory available. Or, it would exceed the maximum number of allowed mappings.

STANDARDS

This call is Linux-specific, and should not be used in programs intended to be portable.

NOTES

mremap() changes the mapping between virtual addresses and memory pages. This can be used to implement a very efficient **realloc(3)**.

In Linux, memory is divided into pages. A process has (one or) several linear virtual memory segments. Each virtual memory segment has one or more mappings to real memory pages (in the page table). Each virtual memory segment has its own protection (access rights), which may cause a segmentation violation (**SIGSEGV**) if the memory is accessed incorrectly (e.g., writing to a read-only segment). Accessing virtual memory outside of the segments will also cause a segmentation violation.

If **mremap()** is used to move or expand an area locked with **mlock(2)** or equivalent, the **mremap()** call will

make a best effort to populate the new area but will not fail with **ENOMEM** if the area cannot be populated.

Prior to glibc 2.4, glibc did not expose the definition of **MREMAP_FIXED**, and the prototype for **mremap()** did not allow for the *new_address* argument.

MREMAP_DONTUNMAP use cases

Possible applications for **MREMAP_DONTUNMAP** include:

- Non-cooperative **userfaultfd(2)**: an application can yank out a virtual address range using **MREMAP_DONTUNMAP** and then employ a **userfaultfd(2)** handler to handle the page faults that subsequently occur as other threads in the process touch pages in the yanked range.
- Garbage collection: **MREMAP_DONTUNMAP** can be used in conjunction with **userfaultfd(2)** to implement garbage collection algorithms (e.g., in a Java virtual machine). Such an implementation can be cheaper (and simpler) than conventional garbage collection techniques that involve marking pages with protection **PROT_NONE** in conjunction with the use of a **SIGSEGV** handler to catch accesses to those pages.

BUGS

Before Linux 4.14, if *old_size* was zero and the mapping referred to by *old_address* was a private mapping (**mmap(2)** **MAP_PRIVATE**), **mremap()** created a new private mapping unrelated to the original mapping. This behavior was unintended and probably unexpected in user-space applications (since the intention of **mremap()** is to create a new mapping based on the original mapping). Since Linux 4.14, **mremap()** fails with the error **EINVAL** in this scenario.

SEE ALSO

brk(2), **getpagesize(2)**, **getrlimit(2)**, **mlock(2)**, **mmap(2)**, **sbrk(2)**, **malloc(3)**, **realloc(3)**

Your favorite text book on operating systems for more information on paged memory (e.g., *Modern Operating Systems* by Andrew S. Tanenbaum, *Inside Linux* by Randolph Bentson, *The Design of the UNIX Operating System* by Maurice J. Bach)