

**Bahir Dar University**

**Bahir Dar Institution of Technology**

**Faculty of Computing**

**Department of Software Engineering**

**Operating system and system programming**

**Individual Assignment Two(2)**

**mremap System call**

**Name :- Ephrem Abera**

**ID No :- 1307655**

**Date of submission :- 17/11/2014E.C**

**Submitted to :- Wendimu Baye**

***mremap - remap a virtual memory address***

A memory mapping can be moved while being expanded (or contracted) by the mremap() function (dependent on the flags supplied and the available virtual address space).

The former address of the virtual memory block that you want to enlarge is old address (or shrink). Note that old address needs to be positioned on the page. Old size refers to the virtual memory block's previous size. The requested size for the virtual memory block following the resize is known as new size. The fifth input, new address, is an optional one(MREMAP\_FIXED).

The identical pages will be recreated in a new mapping by mremap() if old size is 0 and old address refers to a shared mapping. The new mapping's size will be determined by new size, and its location can be determined by new address.

This method must also provide the MREMAP MAYMOVE option in order to request a new mapping.

The mapping between virtual addresses and memory pages is altered via mremap(). This can be used to implement a realloc that is quite effective.

Pages make up the memory structure in Linux. There are one or more linear virtual memory segments in a process. There are one or more mappings from each virtual memory segment to actual memory pages. Each virtual memory segment has its own security (access rights), and if the memory is accessed improperly (for example, by writing to a read-only segment), this could result in a segmentation violation (SIGSEGV). Segmentation violations can also result from accessing virtual memory outside of the segments.

The mremap() call will make a best attempt to populate the new area if it is used to move or enlarge an area locked with mlock or an equivalent, but it won't fail with ENOMEM(a type of error which there wasn't enough memory to finish the operation) if the area cannot be populated.

**SYNOPSIS**

#define \_GNU\_SOURCE /\* See feature\_test\_macros(7) \*/

#include <sys/mman.h>

**mremap**

void \*mremap(void \*old\_address, size\_t old\_size,

size\_t new\_size, int flags, ... /\* void \*new\_address \*/);

**How mremap call**

#include <errno.h>

#include <sys/mman.h>

#include <stdarg.h>

#include <stdint.h>

#include <unistd.h>

#include "platform/bionic/macros.h"

extern "C" void\* \_\_mremap(void\*, size\_t, size\_t, int, void\*);

void\* mremap(void\* old\_address, size\_t old\_size, size\_t new\_size, int flags, .../\* void \*new\_address \*/) {

// prevent allocations large enough for `end - start` to overflow

size\_t rounded = \_\_BIONIC\_ALIGN(new\_size, PAGE\_SIZE);

if (rounded < new\_size || rounded > PTRDIFF\_MAX) {

errno = ENOMEM;

return MAP\_FAILED;

}

void\* new\_address = nullptr;

// The optional argument is only valid if the MREMAP\_FIXED flag is set,

// so we assume it's not present otherwise.

if ((flags & MREMAP\_FIXED) != 0) {

va\_list ap;

va\_start(ap, flags);

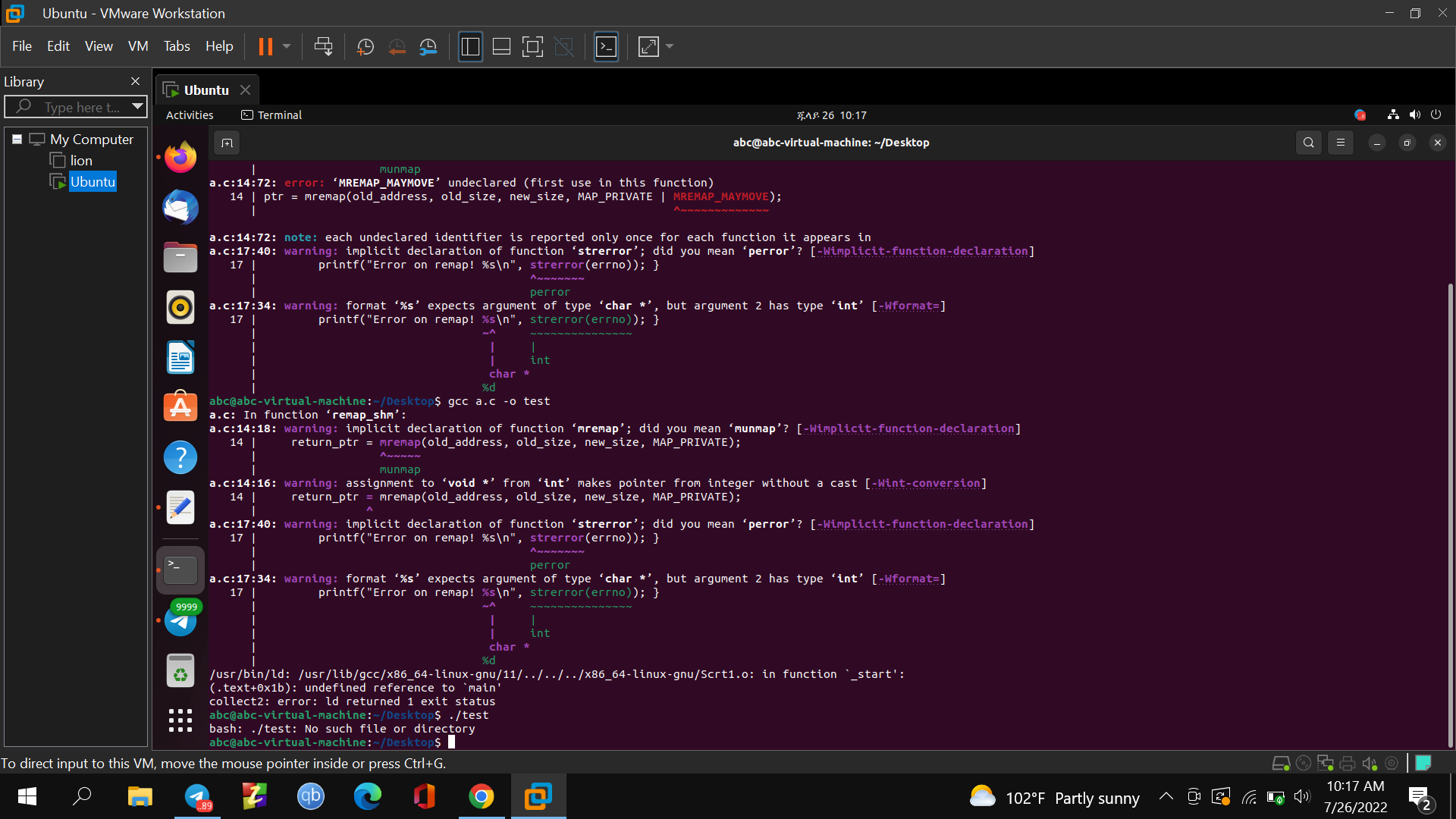
new\_address = va\_arg(ap, void\*);

va\_end(ap);

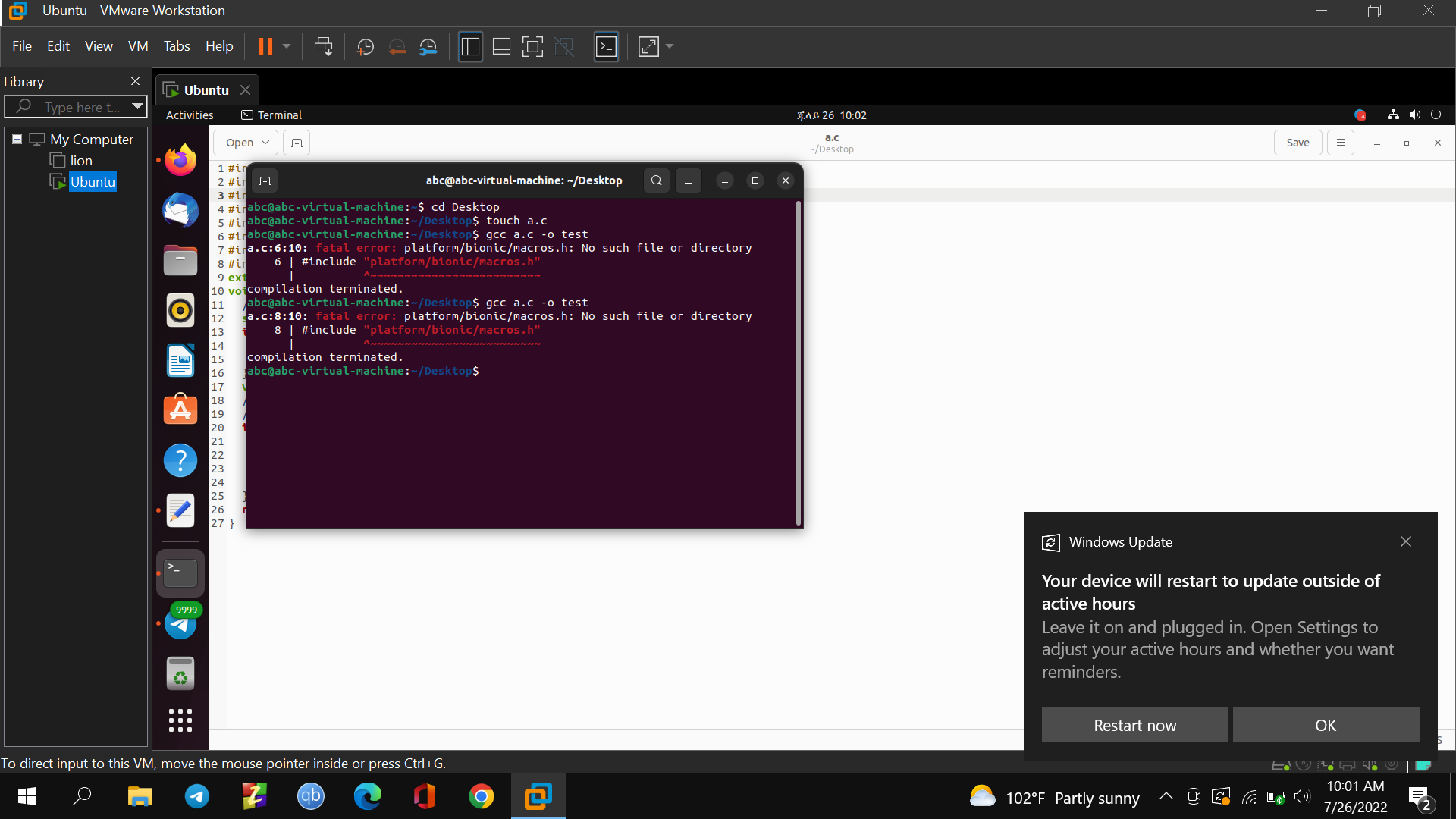
}

return \_\_mremap(old\_address, old\_size, new\_size, flags, new\_address);

}



I have tried many times in a number of ways like by downloading the library of "platform/bionic/macros.h", but the error is intolerable. Eventually I could able to implement mmap.



***Flags***

1. **MREMAP\_MAYMOVE :-** By default, mremap() fails if there is not enough room to enlarge a mapping at its current location. If this flag is set, the kernel is allowed to move the mapping to a different virtual address if it becomes necessary. Absolute pointers into the old mapping location become invalid if the mapping is moved; instead, offsets relative to the mapping's initial address should be used.
2. **MREMAP\_FIXED :-** The MAP FIXED flag of mmap performs a similar function as does this flag. The void \*new address parameter, which specifies a page-aligned address to which the mapping must be transferred, is accepted as a fifth argument by mremap() if this flag is set. The address range defined by new address and new size is unmapped of any prior mapping.

MREMAP FIXED must be provided in conjunction with MREMAP MAYMOVE.

***Example of mmap system call for memory allocation***

#include <stdio.h>

#include <sys/mman.h>

int main(){

int N=5;

int \*ptr = mmap ( NULL, N\*sizeof(int),

PROT\_READ | PROT\_WRITE, MAP\_PRIVATE | MAP\_ANONYMOUS, 0, 0 );

if(ptr == MAP\_FAILED){

printf("Mapping Failed\n");

return 1;

}

for(int i=0; i<N; i++)

ptr[i] = i\*10;

for(int i=0; i<N; i++)

printf("[%d] ",ptr[i]);

printf("\n");

int err = munmap(ptr, 10\*sizeof(int));

if(err != 0){

printf("UnMapping Failed\n");

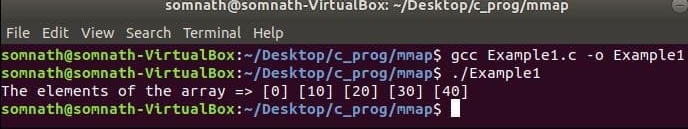
return 1;

}

return 0;

}

**Output**



**Here is mmap implemented with this example**

**2.1. mmap MAP\_FIXED :-** Place the mapping at that specific address rather than taking the addr as a clue. Addr needs to be properly aligned; for the majority of systems, a multiple of the page size is adequate; however, some architectures might have additional constraints. The overlapped portion of any existing mapping(s) will be removed if the memory region indicated by addr and length overlaps any of the pages of any existing mapping(s). Mmap() will fail if the given address cannot be used.

**3. MREMAP\_DONTUNMAP :-** This flag, which needs to be used with MREMAP MAYMOVE, remaps a mapping to a new location but leaves the mapping at the old address in place.

Only private anonymous mappings are compatible with the MREMAP DONTUNMAP flag (for further information, see mmap of MAP PRIVATE and MAP ANONYMOUS).

**3.1. MAP\_PRIVATE :-** A private copy-on-write mapping should be made. Other processes mapping the same file cannot see updates to the mapping, and the updates are not transferred over to the underlying file. If modifications to the file occur after the mmap() call, they may not be seen in the mapped region.

**3.2. MAP\_ANON :-** Synonym for MAP\_ANONYMOUS; provided for compatibility with other implementations.

Following completion, a page fault will occur on any access to the range defined by old address and old size. If the address is within a previously registered range with userfaultfd, a userfaultfd handler will be used to handle the page fault. If not, the kernel creates a page that is zeroed out to handle the fault.

**userfaultfd** - create a file descriptor for handling page faults in user space.

***SYNOPSIS for userfaultfd***

#include <fcntl.h> /\* Definition of O\_\* constants \*/

#include <sys/syscall.h> /\* Definition of SYS\_\* constants \*/

#include <unistd.h>

**System call for userfaultfd**

int syscall(SYS\_userfaultfd, int flags);

**userfaultfd use :-** A thread in a multithreaded program can handle user-space paging for the other threads in the process thanks to the userfaultfd mechanism. The faulting thread is put to sleep and an event is produced that can be read using the userfaultfd file descriptor if a page fault happens for one of the areas registered to the userfaultfd object.

**The MREMAP\_DONTUNMAP flag** may be used to atomically move mapping while leaving the source mapped.

When a memory segment is resized, relocated, or locked using old address and old size, the lock is preserved if it was previously locked using ***mlock*** or a similar mechanism. As a result, the process may lock different amounts of memory.

**mlock() :-** preventing that memory from being paged to the swap area. Pages in the address range beginning at addr and extending for len bytes are locked by the mlock() function. When the call successfully completes, all pages that contain a portion of the specified address range are guaranteed to be resident in RAM. These pages are also guaranteed to remain in RAM until they are later unlocked.

**SYNOPSIS for mlock()**

#include <sys/mman.h>

int mlock(const void \*addr, size\_t len);

**RETURN VALUE :-** Mremap() returns a pointer to the new virtual memory area if it is successful. Errors are indicated by setting ***errno*** to the value MAP FAILED, which is equivalent to (void \*) -1.

***erron :-*** number of last error.

**SYNOPSIS for erron :-**

#include <errno.h>

The <errno.h> header file defines the integer variable errno, which is set by system calls and some library functions in the event of an error to indicate what went wrong. A function that succeeds is permitted to alter errno, but only when the return value of the call indicated an error (typically, -1 from most system calls and -1 or NULL from most library functions). No library function or system call ever sets the value of errno to zero.

**EXAMPLE**

Mremap() returns a pointer to the new virtual memory area if it is successful. Errors result in the return of the value MAP FAILED, which is (void \*) -1, and the corresponding setting of errno.

Here is my mremap code. The values are set as follows upon entry:

old\_address: (void \*) 0x7ffff4957000 (140737296822272)

old\_size: 8003584

new\_size: 16007168

After the call, the pointer return ptr is the old size value (8003584), which is not a valid pointer value, at 0x7a2000, which is a void \* value.

Here is the code:

void \* remap\_shm(void \*old\_address, size\_t old\_size, size\_t new\_size)

{

void \* return\_ptr;

return\_ptr = mremap(old\_address, old\_size, new\_size, MAP\_PRIVATE | MREMAP\_MAYMOVE);

if (return\_ptr <= 0){

printf("Error on remap! %s\n", strerror(errno)); }

return return\_ptr;

}

It fails if I don't include the MREMAP MAYMOVE flag. I could able to explain why the pointer to the new memory mapping is returned with the old size value by omitting MAP\_PRIVATE. The MAP PRIVATE flag for mremap is invalid. On my system, MAP PRIVATE == MREMAP FIXED, which indicates that mremap would erroneously anticipate a fifth input that you do not supply.