



Bahir Dar University
Bahir Dar Institution of Technology
Faculty of Computing
Department of Software Engineering
Operating system and System Programming
Individual Assignment -2

Title- System call (mprotect)

Prepared by Jemal Workie

ID-BDU1307712

Date of Submission: -17/11/2014E.C
Submitted to Instructor Wondmu Baye

Contents

INTRODUCTION	1
1-What / Why / How, mprotect() system call?.....	1
What is mprotect() function?	1
Why mprotect?	1
How does mprotect work?.....	1
How does Mprotect () protect memory?.....	1
2- List of parametres and flags.....	3
3- List of the flags, with their purpose and code implementation.	4

INTRODUCTION

A system call is a **way for programs to interact with the operating system**. System calls are function invocations made from user space into the kernel in order to request some service or resource from the operating system. A computer program makes a system call when it makes a request to the operating system's kernel. System call provides the services of the operating system to the user programs via Application Program Interface (API).

1-What / Why / How, mprotect() system call?

What is mprotect() function?

Name: *mprotect* -set protection on a region of memory

Synopsis: `#include <sys/mman.h>`

`int mprotect(void *addr, size_t len, int prot);`

The *mprotect()* function is used to change the access protection of a memory mapping to that specified by *prot*. All whole pages of the process's address space, that were established by the *mmap()* function, addressed from *addr* continuing for a length of *len* will be affected by the change of access protection. It has been used to specify or change the required protection for the process's memory page(s).

Why mprotect?

Because the function *mprotect()* changes the access protections to be that specified by *prot* for those whole pages containing any part of the address space of the process starting at address *addr* and continuing for *len* bytes.

On architectures with a memory management unit (MMU), the address that *mprotect()* takes as an argument is a virtual address. Each process has its own independent virtual address space, so there's only two possibilities:

- The requested address is within the process's own address range; or
- The requested address is within the kernel's address range (which is mapped into every process).

How does mprotect work?

mprotect() works internally by altering the flags attached to a virtual memory address (VMA). The first thing it must do is look up the VMA corresponding to the address that was passed - if the passed address was within the kernel's address range, then there is no VMA, and so this search will fail. This is exactly the same thing happens if you try to change the protections on an area of the address space that is not mapped.

How does Mprotect () protect memory?

The program allocates a page of memory by mapping */dev/zero* and writing a value to the allocated page to obtain a private copy. The program protects the memory by calling *mprotect* with the *PROT_NONE* permission.

mprotect() changes the access protections for the calling process's memory pages containing any part of the address range in the interval [*addr*, *addr+len-1*]. *addr* must be aligned to a page boundary.

If the calling process tries to access memory in a manner that violates the protections, then the kernel generates a SIGSEGV signal for the process.

Return Value

On success, **mprotect()** returns zero. On error, this system call return -1, and *errno* is set to indicate the error.

Errors

EACCESS

The memory cannot be given the specified access. This can happen, for example, if you mmap a file to which you have read-only access, then ask **mprotect()** to mark it **PROT_WRITE**.

EINVAL

addr is not a valid pointer, or not a multiple of the system page size.

EINVAL

Both **PROT_GROWSUP** and **PROT_GROWSDOWN** were specified in *prot*.

EINVAL

Invalid flags specified in *prot*.

EINVAL

(PowerPC architecture) **PROT_SAO** was specified in *prot*, but SAO hardware feature is not available.

ENOMEM

Internal kernel structures could not be allocated.

ENOMEM

Addresses in the range [*addr*, *addr+len-1*] are invalid for the address space of the process, or specify one or more pages that are not mapped. (Before kernel 2.4.19, the error **EFAULT** was incorrectly produced for these cases.)

ENOMEM

Changing the protection of a memory region would result in the total number of mappings with distinct attributes (e.g., read versus read/write protection) exceeding the allowed maximum. (For example, making the protection of a range **PROT_READ** in the middle of a region currently protected as **PROT_READ|PROT_WRITE** would result in three mappings: two read/write mappings at each end and a read-only mapping in the middle.)

On Linux, it is always permissible to call **mprotect()** on any address in a process's address space (except for the kernel vsyscall area). In particular, it can be used to change existing code mappings to be writable.

Whether **PROT_EXEC** has any effect different from **PROT_READ** depends on processor architecture, kernel version, and process state. If **READ_IMPLIES_EXEC** is set in the process's personality flags, specifying **PROT_READ** will implicitly add **PROT_EXEC**.

On some hardware architectures (e.g., i386), **PROT_WRITE** implies **PROT_READ**.

POSIX.1 says that an implementation may permit access other than that specified in *prot*, but at a minimum can allow write access only if **PROT_WRITE** has been set, and must not allow any access if **PROT_NONE** has been set.

Applications should be careful when mixing use of **mprotect()** and **pkey_mprotect()**. On x86, when **mprotect()** is used with *prot* set to **PROT_EXEC** a pkey may be allocated and set on the memory implicitly by the kernel, but only when the pkey was 0 previously.

2- List of parametres and flags

The list of parameters used in **mprotect(void **addr*, size_t *len*, int *prot*)**

- I. **addr**- pointer to region in memory. The starting address of the memory region for which the access is to be changed. The *addr* argument must be a multiple of the page size. The **sysconf()** function may be used to determine the system page size.
- II. **len**- The length in bytes of the address range.
- III. **prot**- The desired access protection flag. You may specify **PROT_NONE**, **PROT_READ**, **PROT_WRITE**, or the inclusive or of **PROT_READ** AND **PROT_WRITE** as values for the *protection* argument.

It is a combination of the following access flags: **PROT_NONE** or a bitwise-or of the other values in the following list:

PROT_NONE

An option to disable memory access. The memory cannot be accessed at all.

PROT_READ

The memory can be read.

PROT_WRITE

The memory can be modified.

PROT_EXEC

The memory can be executed.

PROT_SEM (since Linux 2.5.7)

The memory can be used for atomic operations.

PROT_SAO (since Linux 2.6.26)

The memory should have strong access ordering.

Additionally (since Linux 2.6.0), *prot* can have one of the following flags set:

PROT_GROWSUP

Apply the protection mode up to the end of a mapping that grows upwards.

PROT_GROWSDOWN

Apply the protection mode down to the beginning of a mapping that grows downward.

3- List of the flags, with their purpose and code implementation.

PROT_NONE

An option to disable memory access. The memory cannot be accessed at all. It allocates a contiguous virtual memory region with no permissions granted. This can be useful, as other have mentioned, to implement guards (pages that on touch cause segfaults, both for bug hunting and security purposes) or "magic" pointers where values within a PROT_NONE mapping are to be interpreted as something other than a pointer.

PROT_READ

The memory can be read.

Code implementation examples

The program below demonstrates the use of **mprotect()**. The program allocates four pages of memory, makes the third of these pages read-only, and then executes a loop that walks upward through the allocated region modifying bytes.

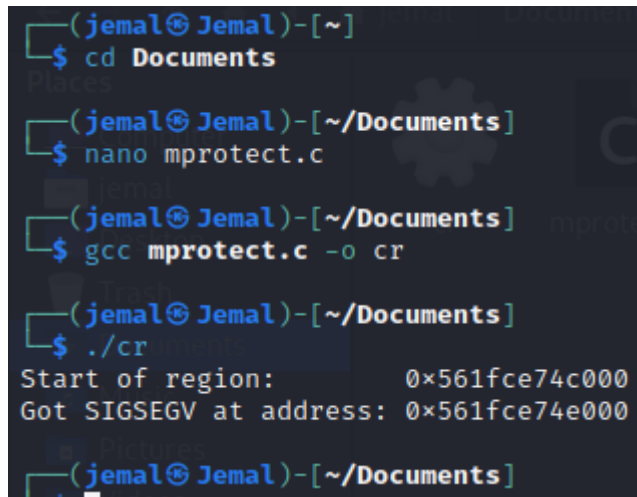
```
#include <unistd.h>
#include <signal.h>
#include <stdio.h>
#include <malloc.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/mman.h>
#define handle_error(msg) \
    do { perror(msg); exit(EXIT_FAILURE); } while (0)
static char *buffer;
static void
handler(int sig, siginfo_t *si, void *unused)
{
    /* Note: calling printf() from a signal handler is not safe
       (and should not be done in production programs), since
       printf() is not async-signal-safe; see signal-safety(7).
       Nevertheless, we use printf() here as a simple way of
       showing that the handler was called. */
    printf("Got SIGSEGV at address: %p\n", si->si_addr);
    exit(EXIT_FAILURE);
}
int
main(int argc, char *argv[])
{
    int pagesize;
    struct sigaction sa;
    sa.sa_flags = SA_SIGINFO;
    sigemptyset(&sa.sa_mask);
    sa.sa_sigaction = handler;
    if (sigaction(SIGSEGV, &sa, NULL) == -1)
        handle_error("sigaction");
```

```

pagesize = sysconf(_SC_PAGE_SIZE);
if (pagesize == -1)
    handle_error("sysconf");
/* Allocate a buffer aligned on a page boundary;
   initial protection is PROT_READ | PROT_WRITE. */
buffer = memalign(pagesize, 4 * pagesize);
if (buffer == NULL)
    handle_error("memalign");
printf("Start of region:      %p\n", buffer);
if (mprotect(buffer + pagesize * 2, pagesize,
    PROT_READ) == -1)
    handle_error("mprotect");
for (char *p = buffer ; ; )
    *(p++) = 'a';
printf("Loop completed\n"); /* Should never happen */
exit(EXIT_SUCCESS);
}

```

OUTPUT



```

(jemal@Jemal)-[~]
$ cd Documents
(jemal@Jemal)-[~/Documents]
$ nano mprotect.c
(jemal@Jemal)-[~/Documents]
$ gcc mprotect.c -o cr
(jemal@Jemal)-[~/Documents]
$ ./cr
Start of region:      0x561fce74c000
Got SIGSEGV at address: 0x561fce74e000
(jemal@Jemal)-[~/Documents]

```

```

#include <unistd.h>
#include <signal.h>
#include <stdio.h>
#include <malloc.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/mman.h>

#define handle_error(msg) \
    do { perror(msg); exit(EXIT_FAILURE); } while (0) /*for error handling

static char *buffer; /*pointer to static buffer variable

static void handler(int sig, siginfo_t *si, void *unused) /*function to handle segfaults signals here
{
    printf("Got SIGSEGV at address: 0x%lx\n", (long) si->si_addr);

```

```

    /* print address where program gets segfault. Should be the read only area i.e. buffer+2*pagesize */
    exit(EXIT_FAILURE);
}

int main(int argc, char *argv[])
{
    char *p;
    int pagesize;
    struct sigaction sa;

    sa.sa_flags = SA_SIGINFO;
    sigemptyset(&sa.sa_mask);
    sa.sa_sigaction = handler;           //setup signal handler
    if (sigaction(SIGSEGV, &sa, NULL) == -1)
        handle_error("sigaction");

    pagesize = sysconf(_SC_PAGE_SIZE);   //function to find pagesize of the system
    if (pagesize == -1)
        handle_error("sysconf");
    printf("Pagesize is:    0x%x\n", pagesize);
    /* Allocate a buffer aligned on a page boundary; initial protection is PROT_READ |
    PROT_WRITE */

    buffer = memalign(pagesize, 4 * pagesize); //allocate memory of size 4*pagesize aligned to
    pagesize on heap
    if (buffer == NULL)
        handle_error("memalign");
    printf("Start of region:    0x%lx\n", (long) buffer); //print start of buffer memory region

    if (mprotect(buffer + pagesize * 2, pagesize, PROT_READ) == -1)
        /* Make memory at address buffer+2*pagesize of size [pagesize] to read only */
        handle_error("mprotect");

    for (p = buffer ; ; ) //loop to keep writing in the memory region after buffer
        *(p++) = 'a';
    /* Will give segfault when you it tries to write in region which we make read only with mprotect */

    printf("Loop completed\n"); // Should never happen */
    exit(EXIT_SUCCESS);
}

```


OUTPUT

```
(jema1@Jema1)-[~/Documents]
$ nano mprotect3.c

(jema1@Jema1)-[~/Documents]
$ ls
cr  mprotect1.c  mprotect2.c  mprotect3.c  mprotect.c

(jema1@Jema1)-[~/Documents]
$ gcc mprotect3.c -o cr

(jema1@Jema1)-[~/Documents]
$ ./cr
Pagesize is:          0x1000
Start of region:      0x55bf837e3000
Got SIGSEGV at address: 0x55bf837e5000

(jema1@Jema1)-[~/Documents]
$
```

PROT_WRITE

The memory can be modified.

Here I am to use mprotect against reading first, and then writing.

```
#include <sys/types.h>
#include <sys/mman.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(void)
{
    int pagesize = sysconf(_SC_PAGE_SIZE);
    int *a;
    if (posix_memalign((void*)&a, pagesize, sizeof(int)) != 0)
        perror("memalign");

    *a = 42;
    if (mprotect(a, pagesize, PROT_WRITE) == -1) /* Resp. PROT_READ */
        perror("mprotect");

    printf("a = %d\n", *a);
    *a = 24;
    printf("a = %d\n", *a);
    free (a);
    return 0;
}
```

OUTPUT

```
(jema1@Jema1)-[~/Documents]
$ pwd
/home/jema1/Documents
(jema1@Jema1)-[~/Documents]
$ nano PROT_WRITE.c
(jema1@Jema1)-[~/Documents]
$ ls
cr  PROT_WRITE.c  PROT_NONE.c  PROT_WRITE.c
(jema1@Jema1)-[~/Documents]
$ gcc PROT_WRITE.c -o cr
(jema1@Jema1)-[~/Documents]
$ ./cr
a = 42
a = 24
(jema1@Jema1)-[~/Documents]
$
```

PROT_EXEC

The memory can be executed.

Implementation Example for PROT_EXEC

```
#include <stdio.h>
#include <stdint.h>
#include <unistd.h>
#include <string.h>
#include <errno.h>
#include <sys/mman.h>
static uint8_t code[] = { 0xB8,0x2A,0x00,0x00,0x00, /* mov eax,0x2a */
                          0xC3, /* ret */
};
int main(void)
{
    const size_t len = sizeof(code);
    /* mmap a region for our code */
    void *p = mmap(NULL, len, PROT_READ|PROT_WRITE, /* No PROT_EXEC */
                  MAP_PRIVATE|MAP_ANONYMOUS, -1, 0);
    if (p==MAP_FAILED) {
        perror("mmap() failed");
        return 2;
    }
    /* Copy it in (still not executable) */
    memcpy(p, code, len);
    /* Now make it execute-only */
    if (mprotect(p, len, PROT_EXEC) < 0) {
        perror("mprotect failed to mark exec-only");
        return 2;
    }
    /* Go! */
    int (*func)(void) = p;
```

```

printf("(dynamic) code returned %d\n", func());
return 0;
}

```

OUTPUT

```

(jemal@Jemal)-[~/Documents]
$ pwd
/home/jemal/Documents
(jemal@Jemal)-[~/Documents]
$ nano mprotect2.c
(jemal@Jemal)-[~/Documents]
$ ls
cr mprotect1.c mprotect2.c mprotect.c
(jemal@Jemal)-[~/Documents]
$ gcc mprotect2.c -o cr
(jemal@Jemal)-[~/Documents]
$ ./cr
(dynamic) code returned 42
(jemal@Jemal)-[~/Documents]
$

```

PROT_SEM (since Linux 2.5.7)

The memory can be used for atomic operations. This flag was introduced as part of the `futex(2)` implementation (in order to guarantee the ability to perform atomic operations required by commands such as **FUTEX_WAIT**), **but is not currently used in on any architecture.**

Additionally (since Linux 2.6.0), *prot* can have one of the following flags set:

PROT_SAO (since Linux 2.6.26)

The memory should have strong access ordering. This feature is specific to the PowerPC architecture (version 2.06 of the architecture specification adds the SAO CPU feature, and it is available on POWER 7 or PowerPC A2, for example).

Additionally (since Linux 2.6.0), *prot* can have one of the following flags set:

PROT_GROWSUP

Apply the protection mode up to the end of a mapping that grows upwards. (Such mappings are created for the stack area on architectures—for example, HP-PARISC—that have an upwardly growing stack.)

PROT_GROWSDOWN

Apply the protection mode down to the beginning of a mapping that grows downward (which should be a stack segment or a segment mapped with the **MAP_GROWSDOWN** flag set).

References- <https://www.geeksforgeeks.org/linux-system-call-in-detail/>

- <https://thevivekpandey.github.io/posts/2017-09-25-linux-system-calls.html>

- https://linuxhint.com/list_of_linux_syscalls/