50

VIRTUAL MEMORY OPERATIONS

This chapter looks at various system calls that perform operations on a process's virtual address space:

- The *mprotect()* system call changes the protection on a region of virtual memory.
- The *mlock()* and *mlockall()* system calls lock a region of virtual memory into physical memory, thus preventing it from being swapped out.
- The *mincore()* system call allows a process to determine whether the pages in a region of virtual memory are resident in physical memory.
- The *madvise()* system call allows a process to advise the kernel about its future patterns of usage of a virtual memory region.

Some of these system calls find particular use in conjunction with shared memory regions (Chapters 48, 49, and 54), but they can be applied to any region of a process's virtual memory.

> The techniques described in this chapter are not in fact about IPC at all, but we include them in this part of the book because they are sometimes used with shared memory.

50.1 Changing Memory Protection: *mprotect()*

The *mprotect()* system call changes the protection on the virtual memory pages in the range starting at *addr* and continuing for *length* bytes.

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

int mprotect(void *addr, size_t length, int prot);
                                        Returns 0 on success, or –1 on error
```

The value given in *addr* must be a multiple of the system page size (as returned by *sysconf(_SC_PAGESIZE)*). (SUSv3 specified that *addr must* be page-aligned. SUSv4 says that an implementation *may* require this argument to be page-aligned.) Because protections are set on whole pages, *length* is, in effect, rounded up to the next multiple of the system page size.

The *prot* argument is a bit mask specifying the new protection for this region of memory. It must be specified as either PROT_NONE or a combination created by ORing together one or more of PROT_READ, PROT_WRITE, and PROT_EXEC. All of these values have the same meaning as for *mmap()* (Table 49-2, on page 1020).

If a process attempts to access a region of memory in a manner that violates the memory protection, the kernel generates a SIGSEGV signal for the process.

One use of *mprotect()* is to change the protection of a region of mapped memory originally set in a call to *mmap()*, as shown in Listing 50-1. This program creates an anonymous mapping that initially has all access denied (PROT_NONE). The program then changes the protection on the region to read plus write. Before and after making the change, the program uses the *system()* function to execute a shell command that displays the line from the /proc/*PID*/maps file corresponding to the mapped region, so that we can see the change in memory protection. (We could have obtained the mapping information by directly parsing /proc/self/maps, but we used the call to *system()* because it results in a shorter program.) When we run this program, we see the following:

```
$ ./t_mprotect
Before mprotect()
b7cde000-b7dde000 ---s 00000000 00:04 18258    /dev/zero (deleted)
After mprotect()
b7cde000-b7dde000 rw-s 00000000 00:04 18258    /dev/zero (deleted)
```

From the last line of output, we can see that *mprotect()* has changed the permissions of the memory region to PROT_READ | PROT_WRITE. (For an explanation of the (deleted) string that appears after /dev/zero in the shell output, refer to Section 48.5.)

Listing 50-1: Changing memory protection with *mprotect()*

vmem/t_mprotect.c

```
#define _BSD_SOURCE         /* Get MAP_ANONYMOUS definition from <sys/mman.h> */
#include <sys/mman.h>
#include "tlpi_hdr.h"

#define LEN (1024 * 1024)

#define SHELL_FMT "cat /proc/%ld/maps | grep zero"
#define CMD_SIZE (sizeof(SHELL_FMT) + 20)
                            /* Allow extra space for integer string */
```

```
int
main(int argc, char *argv[])
{
    char cmd[CMD_SIZE];
    char *addr;

    /* Create an anonymous mapping with all access denied */

    addr = mmap(NULL, LEN, PROT_NONE, MAP_SHARED | MAP_ANONYMOUS, -1, 0);
    if (addr == MAP_FAILED)
        errExit("mmap");

    /* Display line from /proc/self/maps corresponding to mapping */

    printf("Before mprotect()\n");
    snprintf(cmd, CMD_SIZE, SHELL_FMT, (long) getpid());
    system(cmd);

    /* Change protection on memory to allow read and write access */

    if (mprotect(addr, LEN, PROT_READ | PROT_WRITE) == -1)
        errExit("mprotect");

    printf("After mprotect()\n");
    system(cmd);                /* Review protection via /proc/self/maps */

    exit(EXIT_SUCCESS);
}
```
vmem/t_mprotect.c

50.2 Memory Locking: *mlock()* and *mlockall()*

In some applications, it is useful to lock part or all of a process's virtual memory so that it is guaranteed to always be in physical memory. One reason for doing this is to improve performance. Accesses to locked pages are guaranteed never to be delayed by a page fault. This is useful for applications that must ensure rapid response times.

Another reason for locking memory is security. If a virtual memory page containing sensitive data is never swapped out, then no copy of the page is ever written to the disk. If the page was written to the disk, it could, in theory, be read directly from the disk device at some later time. (An attacker could deliberately engineer this situation by running a program that consumes a large amount of memory, thus forcing the memory of other processes to be swapped out to disk.) Reading information from the swap space could even be done after the process has terminated, since the kernel makes no guarantees about zeroing out the data held in swap space. (Normally, only privileged processes would be able to read from the swap device.)

> The suspend mode on laptop computers, as well some desktop systems, saves a copy of a system's RAM to disk, regardless of memory locks.

In this section, we look at the system calls used for locking and unlocking part or all of a process's virtual memory. However, before doing this, we first look at a resource limit that governs memory locking.

The RLIMIT_MEMLOCK resource limit

In Section 36.3, we briefly described the RLIMIT_MEMLOCK limit, which defines a limit on the number of bytes that a process can lock into memory. We now consider this limit in more detail.

In Linux kernels before 2.6.9, only privileged processes (CAP_IPC_LOCK) can lock memory, and the RLIMIT_MEMLOCK soft resource limit places an upper limit on the number of bytes that a privileged process can lock.

Starting with Linux 2.6.9, changes to the memory locking model allow unprivileged processes to lock small amounts of memory. This is useful for an application that needs to place a small piece of sensitive information in locked memory in order to ensure that it is never written to the swap space on disk; for example, *gpg* does this with pass phrases. As a result of these changes:

- no limits are placed on the amount of memory that a privileged process can lock (i.e., RLIMIT_MEMLOCK is ignored); and
- an unprivileged process is now able to lock memory up to the soft limit defined by RLIMIT_MEMLOCK.

The default value for both the soft and hard RLIMIT_MEMLOCK limits is 8 pages (i.e., 32,768 bytes on x86-32).

The RLIMIT_MEMLOCK limit affects:

- *mlock()* and *mlockall()*;
- the *mmap()* MAP_LOCKED flag, which is used to lock a memory mapping when it is created, as described in Section 49.6; and
- the *shmctl()* SHM_LOCK operation, which is used to lock System V shared memory segments, as described in Section 48.7.

Since virtual memory is managed in units of pages, memory locks apply to complete pages. When performing limit checks, the RLIMIT_MEMLOCK limit is rounded *down* to the nearest multiple of the system page size.

Although this resource limit has a single (soft) value, in effect, it defines two separate limits:

- For *mlock()*, *mlockall()*, and the *mmap()* MAP_LOCKED operation, RLIMIT_MEMLOCK defines a per-process limit on the number of bytes of its virtual address space that a process may lock.
- For the *shmctl()* SHM_LOCK operation, RLIMIT_MEMLOCK defines a per-user limit on the number of bytes in shared memory segments that may be locked by the real user ID of this process. When a process performs a *shmctl()* SHM_LOCK operation, the kernel checks the total number of bytes of System V shared memory that are already recorded as being locked by the real user ID of the calling process. If the size of the to-be-locked segment would not push that total over the process's RLIMIT_MEMLOCK limit, the operation succeeds.

The reason RLIMIT_MEMLOCK has different semantics for System V shared memory is that a shared memory segment can continue to exist even when it is not attached by any process. (It is removed only after an explicit *shmctl()* IPC_RMID operation, and then only after all processes have detached it from their address space.)

Locking and unlocking memory regions

A process can use *mlock()* and *munlock()* to lock and unlock regions of memory.

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

int mlock(void *addr, size_t length);
int munlock(void *addr, size_t length);
```

Both return 0 on success, or −1 on error

The *mlock()* system call locks all of the pages of the calling process's virtual address range starting at *addr* and continuing for *length* bytes. Unlike the corresponding argument passed to several other memory-related system calls, *addr* does not need to be page-aligned: the kernel locks pages starting at the next page boundary below *addr*. However, SUSv3 optionally allows an implementation to require that *addr* be a multiple of the system page size, and portable applications should ensure that this is so when calling *mlock()* and *munlock()*.

Because locking is done in units of whole pages, the end of the locked region is the next page boundary greater than *length* plus *addr*. For example, on a system where the page size is 4096 bytes, the call *mlock(2000, 4000)* will lock bytes 0 through to 8191.

> We can find out how much memory a process currently has locked by inspecting the VmLck entry of the Linux-specific /proc/*PID*/status file.

After a successful *mlock()* call, all of the pages in the specified range are guaranteed to be locked and resident in physical memory. The *mlock()* system call fails if there is insufficient physical memory to lock all of the requested pages or if the request violates the RLIMIT_MEMLOCK soft resource limit.

We show an example of the use of *mlock()* in Listing 50-2.

The *munlock()* system call performs the converse of *mlock()*, removing a memory lock previously established by the calling process. The *addr* and *length* arguments are interpreted in the same way as for *munlock()*. Unlocking a set of pages doesn't guarantee that they cease to be memory-resident: pages are removed from RAM only in response to memory demands by other processes.

Aside from the explicit use of *munlock()*, memory locks are automatically removed in the following circumstances:

- on process termination;
- if the locked pages are unmapped via *munmap()*; or
- if the locked pages are overlaid using the *mmap()* MAP_FIXED flag.

Details of the semantics of memory locking

In the following paragraphs, we note some details of the semantics of memory locks.

Memory locks are not inherited by a child created via *fork()*, and are not preserved across an *exec()*.

Where multiple processes share a set of pages (e.g., a MAP_SHARED mapping), these pages remain locked in memory as long as at least one of the processes holds a memory lock on the pages.

Memory locks don't nest for a single process. If a process repeatedly calls *mlock()* on a certain virtual address range, only one lock is established, and this lock will be removed by a single call to *munlock()*. On the other hand, if we use *mmap()* to map the same set of pages (i.e., the same file) at several different locations within a single process, and then lock each of these mappings, the pages remain locked in RAM until all of the mappings have been unlocked.

The fact that memory locks are performed in units of pages and can't be nested means that it isn't logically correct to independently apply *mlock()* and *munlock()* calls to different data structures on the same virtual page. For example, suppose we have two data structures within the same virtual memory page pointed to by pointers *p1* and *p2*, and we make the following calls:

```
mlock(*p1, len1);
mlock(*p2, len2);               /* Actually has no effect */
munlock(*p1, len1);
```

All of the above calls will succeed, but at the end of this sequence, the entire page is unlocked; that is, the data structure pointed to by *p2* is not locked into memory.

Note that the semantics of the *shmctl()* SHM_LOCK operation (Section 48.7) differ from those of *mlock()* and *mlockall()*, as follows:

- After a SHM_LOCK operation, pages are locked into memory only as they are faulted in by subsequent references. By contrast, *mlock()* and *mlockall()* fault all of the locked pages into memory before the call returns.
- The SHM_LOCK operation sets a property of the shared memory segment, rather than the process. (For this reason, the value in the /proc/*PID*/status VmLck field doesn't include the size of any attached System V shared memory segments that have been locked using SHM_LOCK.) This means that, once faulted into memory, the pages remain resident even if all processes detach the shared memory segment. By contrast, a region locked into memory using *mlock()* (or *mlockall()*) remains locked only as long as at least one process holds a lock on the region.

Locking and unlocking all of a process's memory

A process can use *mlockall()* and *munlockall()* to lock and unlock all of its memory.

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

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

Both return 0 on success, or −1 on error

The *mlockall()* system call locks all of the currently mapped pages in a process's virtual address space, all of the pages mapped in the future, or both, according to the *flags* bit mask, which is specified by ORing together one or both of the following constants:

MCL_CURRENT
: Lock all pages that are currently mapped into the calling process's virtual address space. This includes all pages currently allocated for the program text, data segments, memory mappings, and the stack. After a successful call specifying the MCL_CURRENT flag, all of the pages of the calling process are guaranteed to be memory-resident. This flag doesn't affect pages that are subsequently allocated in the process's virtual address space; for this, we must use MCL_FUTURE.

MCL_FUTURE
: Lock all pages subsequently mapped into the calling process's virtual address space. Such pages may, for example, be part of a shared memory region mapped via *mmap()* or *shmat()*, or part of the upwardly growing heap or downwardly growing stack. As a consequence of specifying the MCL_FUTURE flag, a later memory allocation operation (e.g., *mmap()*, *sbrk()*, or *malloc()*) may fail, or stack growth may yield a SIGSEGV signal, if the system runs out of RAM to allocate to the process or the RLIMIT_MEMLOCK soft resource limit is encountered.

The same rules regarding the constraints, lifetime, and inheritance of memory locks created with *mlock()* also apply for memory locks created via *mlockall()*.

The *munlockall()* system call unlocks all of the pages of the calling process and undoes the effect of any previous *mlockall(MCL_FUTURE)* call. As with *munlock()*, unlocked pages are not guaranteed to be removed from RAM by this call.

> Before Linux 2.6.9, privilege (CAP_IPC_LOCK) was required to call *munlockall()* (inconsistently, privilege was not required for *munlock()*). Since Linux 2.6.9, privilege is no longer required.

50.3 Determining Memory Residence: *mincore()*

The *mincore()* system call is the complement of the memory locking system calls. It reports which pages in a virtual address range are currently resident in RAM, and thus won't cause a page fault if accessed.

SUSv3 doesn't specify *mincore()*. It is available on many, but not all, UNIX implementations. On Linux, *mincore()* has been available since kernel 2.4.

```
#define _BSD_SOURCE           /* Or: #define _SVID_SOURCE */
#include <sys/mman.h>

int mincore(void *addr, size_t length, unsigned char *vec);
```

Returns 0 on success, or −1 on error

The *mincore()* system call returns memory-residence information about pages in the virtual address range starting at *addr* and running for *length* bytes. The address supplied in *addr* must be page-aligned, and, since information is returned about whole pages, *length* is effectively rounded up to the next multiple of the system page size.

Information about memory residency is returned in *vec*, which must be an array of *(length + PAGE_SIZE − 1) / PAGE_SIZE* bytes. (On Linux, *vec* has the type *unsigned char **; on some other UNIX implementations, *vec* has the type *char **.) The least significant bit of each byte is set if the corresponding page is memory-resident. The setting of the other bits is undefined on some UNIX implementations, so portable applications should test only this bit.

The information returned by *mincore()* can change between the time the call is made and the time the elements of *vec* are checked. The only pages guaranteed to remain memory-resident are those locked with *mlock()* or *mlockall()*.

> Prior to Linux 2.6.21, various implementation problems meant that *mincore()* did not correctly report memory-residence information for `MAP_PRIVATE` mappings or for nonlinear mappings (established using *remap_file_pages()*).

Listing 50-2 demonstrates the use of *mlock()* and *mincore()*. After allocating and mapping a region of memory using *mmap()*, this program uses *mlock()* to lock either the entire region or otherwise groups of pages at regular intervals. (Each of the command-line arguments to the program is expressed in terms of pages; the program converts these to bytes, as required for the calls to *mmap()*, *mlock()*, and *mincore()*.) Before and after the *mlock()* call, the program uses *mincore()* to retrieve information about the memory residency of pages in the region and displays this information graphically.

Listing 50-2: Using *mlock()* and *mincore()*

vmem/memlock.c

```
#define _BSD_SOURCE     /* Get mincore() declaration and MAP_ANONYMOUS
                           definition from <sys/mman.h> */
#include <sys/mman.h>
#include "tlpi_hdr.h"

/* Display residency of pages in range [addr .. (addr + length - 1)] */

static void
displayMincore(char *addr, size_t length)
{
    unsigned char *vec;
    long pageSize, numPages, j;

    pageSize = sysconf(_SC_PAGESIZE);

    numPages = (length + pageSize - 1) / pageSize;
    vec = malloc(numPages);
    if (vec == NULL)
        errExit("malloc");
```

```
    if (mincore(addr, length, vec) == -1)
        errExit("mincore");

    for (j = 0; j < numPages; j++) {
        if (j % 64 == 0)
            printf("%s%10p: ", (j == 0) ? "" : "\n", addr + (j * pageSize));
        printf("%c", (vec[j] & 1) ? '*' : '.');
    }
    printf("\n");

    free(vec);
}

int
main(int argc, char *argv[])
{
    char *addr;
    size_t len, lockLen;
    long pageSize, stepSize, j;

    if (argc != 4 || strcmp(argv[1], "--help") == 0)
        usageErr("%s num-pages lock-page-step lock-page-len\n", argv[0]);

    pageSize = sysconf(_SC_PAGESIZE);
    if (pageSize == -1)
        errExit("sysconf(_SC_PAGESIZE)");

    len =      getInt(argv[1], GN_GT_0, "num-pages") * pageSize;
    stepSize = getInt(argv[2], GN_GT_0, "lock-page-step") * pageSize;
    lockLen =  getInt(argv[3], GN_GT_0, "lock-page-len") * pageSize;

    addr = mmap(NULL, len, PROT_READ, MAP_SHARED | MAP_ANONYMOUS, -1, 0);
    if (addr == MAP_FAILED)
        errExit("mmap");

    printf("Allocated %ld (%#lx) bytes starting at %p\n",
            (long) len, (unsigned long) len, addr);

    printf("Before mlock:\n");
    displayMincore(addr, len);

    /* Lock pages specified by command line arguments into memory */

    for (j = 0; j + lockLen <= len; j += stepSize)
        if (mlock(addr + j, lockLen) == -1)
            errExit("mlock");

    printf("After mlock:\n");
    displayMincore(addr, len);

    exit(EXIT_SUCCESS);
}
```

vmem/memlock.c

The following shell session shows a sample run of the program in Listing 50-2. In this example, we allocate 32 pages, and in each group of 8 pages, we lock 3 consecutive pages:

```
$ su                                              Assume privilege
Password:
# ./memlock 32 8 3
Allocated 131072 (0x20000) bytes starting at 0x4014a000
Before mlock:
0x4014a000: ..............................
After mlock:
0x4014a000: ***.....***.....***.....***.....
```

In the program output, dots represent pages that are not resident in memory, and asterisks represent pages that are resident in memory. As we can see from the final line of output, 3 out of each group of 8 pages are memory-resident.

In this example, we assumed superuser privilege so that the program can use *mlock()*. This is not necessary in Linux 2.6.9 and later if the amount of memory to be locked falls within the RLIMIT_MEMLOCK soft resource limit.

50.4 Advising Future Memory Usage Patterns: *madvise()*

The *madvise()* system call is used is to improve the performance of an application by informing the kernel about the calling process's likely usage of the pages in the range starting at *addr* and continuing for *length* bytes. The kernel may use this information to improve the efficiency of I/O performed on the file mapping that underlies the pages. (See Section 49.4 for a discussion of file mappings.) On Linux, *madvise()* has been available since kernel 2.4.

```
#define _BSD_SOURCE
#include <sys/mman.h>

int madvise(void *addr, size_t length, int advice);
```

Returns 0 on success, or −1 on error

The value specified in *addr* must be page-aligned, and *length* is effectively rounded up to the next multiple of the system page size. The *advice* argument is one of the following:

MADV_NORMAL
: This is the default behavior. Pages are transferred in clusters (a small multiple of the system page size). This results in some read-ahead and read-behind.

MADV_RANDOM
: Pages in this region will be accessed randomly, so read-ahead will yield no benefit. Thus, the kernel should fetch the minimum amount of data on each read.

MADV_SEQUENTIAL
: Pages in this range will be accessed once, sequentially. Thus, the kernel can aggressively read ahead, and pages can be quickly freed after they have been accessed.

MADV_WILLNEED
: Read pages in this region ahead, in preparation for future access. The MADV_WILLNEED operation has an effect similar to the Linux-specific *readahead()* system call and the *posix_fadvise()* POSIX_FADV_WILLNEED operation.

MADV_DONTNEED
: The calling process no longer requires the pages in this region to be memory-resident. The precise effect of this flag varies across UNIX implementations. We first note the behavior on Linux. For a MAP_PRIVATE region, the mapped pages are explicitly discarded, which means that modifications to the pages are lost. The virtual memory address range remains accessible, but the next access of each page will result in a page fault reinitializing the page, either with the contents of the file from which it is mapped or with zeros in the case of an anonymous mapping. This can be used as a means of explicitly reinitializing the contents of a MAP_PRIVATE region. For a MAP_SHARED region, the kernel *may* discard modified pages in some circumstances, depending on the architecture (this behavior doesn't occur on x86). Some other UNIX implementations also behave in the same way as Linux. However, on some UNIX implementations, MADV_DONTNEED simply informs the kernel that the specified pages can be swapped out if necessary. Portable applications should not rely on the Linux's destructive semantics for MADV_DONTNEED.

  Linux 2.6.16 added three new nonstandard *advice* values: MADV_DONTFORK, MADV_DOFORK, and MADV_REMOVE. Linux 2.6.32 and 2.6.33 added another four nonstandard *advice* values: MADV_HWPOISON, MADV_SOFT_OFFLINE, MADV_MERGEABLE, and MADV_UNMERGEABLE. These values are used in special circumstances and are described in the *madvise(2)* manual page.

Most UNIX implementations provide a version of *madvise()*, typically allowing at least the *advice* constants described above. However, SUSv3 standardizes this API under a different name, *posix_madvise()*, and prefixes the corresponding *advice* constants with the string POSIX_. Thus, the constants are POSIX_MADV_NORMAL, POSIX_MADV_RANDOM, POSIX_MADV_SEQUENTIAL, POSIX_MADV_WILLNEED, and POSIX_MADV_DONTNEED. This alternative interface is implemented in *glibc* (version 2.2 and later) by calls to *madvise()*, but it is not available on all UNIX implementations.

> SUSv3 says that *posix_madvise()* should not affect the semantics of a program. However, in *glibc* versions before 2.7, the POSIX_MADV_DONTNEED operation is implemented using *madvise()* MADV_DONTNEED, which does affect the semantics of a program, as described earlier. Since *glibc* 2.7, the *posix_madvise()* wrapper implements POSIX_MADV_DONTNEED to do nothing, so that it does not affect the semantics of a program.

50.5 Summary

In this chapter, we considered various operations that can be performed on a process's virtual memory:

- The *mprotect()* system call changes the protection on a region of virtual memory.
- The *mlock()* and *mlockall()* system calls lock part or all of a process's virtual address space, respectively, into physical memory.
- The *mincore()* system call reports which pages in a virtual memory region are currently resident in physical memory.
- The *madvise()* system call and the *posix_madvise()* function allow a process to advise the kernel about the process's expected patterns of memory use.

50.6 Exercises

50-1. Verify the effect of the `RLIMIT_MEMLOCK` resource limit by writing a program that sets a value for this limit and then attempts to lock more memory than the limit.

50-2. Write a program to verify the operation of the *madvise()* `MADV_DONTNEED` operation for a writable `MAP_PRIVATE` mapping.