Memory Management 6

Overview of the Virtual Memory System

The UNIX system provides a complete set of memory management mechanisms, providing applications complete control over the construction of their address space and permitting a wide variety of operations on both process address spaces and the variety of memory objects in the system.

Process address spaces are composed of a vector of memory pages, each of which can be independently mapped and manipulated. Typically, the system presents the user with mappings that simulate the traditional UNIX process memory environment, but other views of memory are useful as well.

The UNIX memory-management facilities do the following.

- Unify system operations on memory
- Provide a set of kernel mechanisms powerful and general enough to support the implementation of fundamental system services without special-purpose kernel support
- Maintain consistency with the existing environment, in particular using the UNIX file system as the name space for named virtual-memory objects

Virtual Memory, Address Spaces, and Mapping

The system virtual memory (VM) consists of all available physical memory resources. Examples include local and remote file systems, processor primary memory, swap space, and other random-access devices. Named objects in the

virtual memory are referenced though the UNIX file system. However, not all file system objects are in the virtual memory; devices that cannot be treated as storage, such as terminal and network device files, are not in the virtual memory. Some virtual memory objects, such as private process memory and shared memory segments, do not have names.

A process address space is defined by mappings onto objects in the system virtual memory (usually files). Each mapping is constrained to be sized and aligned with the page boundaries of the system on which the process is executing. Each page may be mapped (or not) independently. Only process addresses that are mapped to some system object are valid, for there is no memory associated with processes themselves—all memory is represented by objects in the system virtual memory.

Each object in the virtual memory has an object address space defined by some physical storage. A reference to an object address accesses the physical storage that implements the address within the object. The physical storage associated with virtual memory is thus accessed by transforming process addresses to object addresses, and then to the physical store.

A given process page may map to only one object, although a given object address may be the subject of many process mappings. An important characteristic of a mapping is that the object to which the mapping is made is not affected by the existence of the mapping. Thus, it cannot, in general, be expected that an object has an "awareness" of having been mapped, or of which portions of its address space are accessed by mappings; in particular, the notion of a "page" is not a property of the object. Establishing a mapping to an object simply provides the potential for a process to access or change the object's contents.

The establishment of mappings provides an access method that renders an object directly addressable by a process. Applications may find it advantageous to access the storage resources they use directly rather than indirectly through `read` and `write`. Potential advantages include efficiency (elimination of unnecessary data copying) and reduced complexity (single-step updates rather than the `read`, modify buffer, `write` cycle). The ability to access an object and have it retain its identity over the course of the access is unique to this access method, and facilitates the sharing of common code and data.

Networking, Heterogeneity, and Coherence

The VM system is designed to fit well with the larger UNIX heterogeneous environment. This environment extensively uses networking to access file systems—file systems that are now part of the system virtual memory.

Networks are not constrained to consist of similar hardware or to be based upon a common operating system; in fact, the opposite is encouraged, for such constraints create serious barriers to accommodating heterogeneity.

Although a given set of processes might *apply* a set of mechanisms to establish and maintain the properties of various system objects—properties such as page sizes and the ability of objects to synchronize their own use—a given operating system should not *impose* such mechanisms on the rest of the network.

As it stands, the access method view of a virtual memory maintains the potential for a given object (say a text file) to be mapped by systems running the UNIX memory management system and also to be accessed by systems for which virtual memory and storage management techniques such as paging are totally foreign, such as PC-DOS. Such systems can continue to share access to the object, each using and providing its programs with the access method appropriate to that system.

Another consideration arises when applications use an object as a communications channel, or otherwise attempt to access it simultaneously. In both of these cases, the object is being shared, and the applications must use some synchronization mechanism to guarantee the coherence of their transactions with it. The scope and nature of the synchronization mechanism is best left to the application to decide.

For example, file access on systems that do not support virtual memory access methods must be indirect, by way of `read` and `write`. Applications sharing files on such systems must coordinate their access using semaphores, file locking, or some application-specific protocols.

What is required in an environment where mapping replaces `read` and `write` as the access method is an operation, such as `fsync`, that supports atomic update operations.

The nature and scope of synchronization over shared objects is application-defined from the outset. If the system attempted to impose any automatic semantics for sharing, it might prohibit other useful forms of mapped access that have nothing whatsoever to do with communication or sharing.

By providing the mechanism to support coherency, and leaving it to cooperating applications to apply the mechanism, the needs of applications are met without erecting barriers to heterogeneity. Note that this design does not prohibit the creation of libraries that provide coherent abstractions for common application needs.

Memory Management Interfaces

The applications programmer gains access to the facilities of the virtual memory system through several sets of functions. This section summarizes these calls and provides examples of their use. For details, see the *man Pages(2): System Calls*.

Creating and Using Mappings

```
caddr_t
mmap(caddr_t addr, size_t len, int prot, int flags, int fd,
     off_t off);
```

`mmap` establishes a mapping between a process address space and an object in the system virtual memory. It is the system's most fundamental function for defining the contents of an address space—all other system functions that contribute to the definition of an address space are built from `mmap`. The format of an `mmap` call is:

```
paddr = mmap(addr, len, prot, flags, fd, off);
```

`mmap` establishes a mapping from the process address space at an address *paddr* for *len* bytes to the object specified by *fd* at offset *off* for *len* bytes. The value returned by `mmap` is an implementation-dependent function of the parameter *addr* and the setting of the `MAP_FIXED` bit of *flags*, as described below. A successful call to `mmap` returns *paddr* as its result. The address range *[paddr, paddr + len)*[1] must be valid for the address space of the process and the range *[off, off + len)* must be valid for the virtual memory object.

Note – The mapping established by `mmap` replaces any previous mappings for the process pages in the range [*paddr, paddr + len*].

1. Read the notation *[lower, lower + upper)* as "from and including the lower boundary up to, but not including, the upper boundary."

The parameter *prot* determines whether read, execute, write, or some combination of accesses are permitted to the pages being mapped. To deny all access, set *prot* to `PROT_NONE`. Otherwise, specify permissions by an OR of `PROT_READ`, `PROT_EXECUTE`, and `PROT_WRITE` (note that `PROT_EXECUTE` is specific to the SPARC architecture). A write access will fail if `PROT_WRITE` has not been set, though the behavior of the write can be influenced by setting `MAP_PRIVATE` in the *flags* parameter, as described below.

The *flags* parameter provides other information about the handling of mapped pages.

- `MAP_SHARED` and `MAP_PRIVATE` specify the mapping type, and one of them must be specified. The mapping type describes the disposition of store operations made by *this* process into the address range defined by the mapping operation.

  If `MAP_SHARED` is specified, write references will modify the mapped object. No further operations on the object are necessary to effect a change—the act of storing into a `MAP_SHARED` mapping is equivalent to doing a `write` function.

  On the other hand, if `MAP_PRIVATE` is specified, an initial write reference to a page in the mapped area will create a copy of that page and redirect the initial and successive write references to that copy. This operation is sometimes referred to as *copy-on-write* and occurs invisibly to the process causing the store. Only pages actually modified have copies made in this manner.

  The mapping type is retained across a `fork`.

Note – The private copy is not created until the first write; until then, other users who have the object mapped `MAP_SHARED` can change the object. That is, if one user has an object mapped `MAP_PRIVATE` and another user has the same object mapped `MAP_SHARED`, and the `MAP_SHARED` user changes the object before the `MAP_PRIVATE` user does the first write, then the changes appear in the `MAP_PRIVATE` user's copy that the system makes on the first write. If an application needs isolation from changes made by other processes, it should use `read` to make a copy of the data it is isolating.

MAP_PRIVATE mappings are used by system functions such as exec(2) when mapping files containing programs for execution. This permits operations by programs such as debuggers to modify the "text" (code) of the program without affecting the file from which the program is obtained.

- MAP_FIXED informs the system that the value returned by mmap must be exactly *addr*. The use of MAP_FIXED is discouraged, as it can prevent an implementation from making the most effective use of system resources.

  When MAP_FIXED is not set, the system uses *addr* as a hint to arrive at *paddr*. The *paddr* so chosen is an area of the address space that the system deems suitable for a mapping of *len* bytes to the specified object. An *addr* value of zero grants the system complete freedom in selecting *paddr*, subject to constraints described below. A non-zero value of *addr* is taken as a suggestion of a process address near which the mapping should be placed.

  When the system selects a value for *paddr*, it never places a mapping at address 0, nor replaces any extant mapping, nor maps into areas considered part of the potential data or stack "segments." The system strives to choose alignments for mappings that maximize the performance of the hardware resources.

- MAP_NORESERVE specifies that no swap space is to be reserved in advance for a mapping. Without this flag, a MAP_PRIVATE mapping has swap space reserved for it when the mapping is first created; this swap space is later used to back the private pages that are created by copy-on-write operations.

  Without this advance reservation, swap space might not be available in the system when a copy-on-write is attempted; the system then fails the write access to the page and sends a SIGBUS signal to the process. However, a process can prevent swap space from being reserved in advance by setting the MAP_NORESERVE flag if that process is willing to handle the case in which swap space is not available.

  The advantage of using this flag is that a process can, for example, create and access a huge data segment on a machine that has a relatively small amount of swap space, as long as the process also provides for the case where writes into the segment might fail. Without MAP_NORESERVE it would be impossible to create this segment.

The file descriptor used in a `mmap` call need not be kept open after the mapping is established. If it is closed, the mapping will remain until such time as it is replaced by another call to `mmap` that explicitly specifies the addresses occupied by this mapping or until the mapping is removed either by process termination or a call to `munmap`.

Although the mapping endures independent of the existence of a file descriptor, changes to the file can influence accesses to the mapped area, even if they do not affect the mapping itself.

For instance, should a file be shortened by a call to `truncate`, such that the mapping now "overhangs" the end of the file, then accesses to that area of the file that no longer exists, SIGBUS signals will result.

It is possible to create the mapping in the first place such that it "overhangs" the end of the file—the only requirement when creating a mapping is that the addresses, lengths, and offsets specified in the operation be *possible* (such as, within the range permitted for the object in question), not that they exist at the time the mapping is created (or subsequently.)

Similarly, if a program accesses an address in a manner inconsistent with how it has been mapped (for instance, by attempting a store operation into a mapping that was established with only `PROT_READ` access), then a `SIGSEGV` signal will result. `SIGSEGV` signals will also result on any attempt to reference an address not defined by any mapping.

In general, if a program references an address that is inconsistent with the mapping (or lack of a mapping) established at that address, the system will respond with a `SIGSEGV` violation.

However, if a program references an address consistent with how the address is mapped, but that address does not evaluate *at* the time of the access to allocated storage in the object being mapped, then the system will respond with a `SIGBUS` violation.

In this manner a program (or user) can distinguish between whether it is the mapping or the object that is inconsistent with the access, and take appropriate remedial action.

Using `mmap` to access system memory objects can simplify programs in a variety of ways. Keeping in mind that `mmap` can really be viewed as just a means to access memory objects, it is possible to program using `mmap` in many cases where you might program with `read` or `write`.

However, it is important to realize that `mmap` can only be used to gain access to *memory* objects—those objects that can be thought of as randomly accessible storage. Thus, terminals and network connections cannot be accessed with `mmap` because they are not "memory." Magnetic tapes, even though they are memory devices, cannot be accessed with `mmap` because storage locations on the tape can only be addressed sequentially.

Some examples of situations that *can* be thought of as candidates for use of `mmap` over more traditional methods of file access include:

- Random access operations—either map the entire file into memory or, if the address space cannot accommodate the file or if the file size is variable, create "windows" of mappings to the object.
- Efficiency—even in situations where access is sequential, if the object being accessed can be accessed via `mmap`, an efficiency gain may be obtained by avoiding the copying operations inherent in accesses via `read` or `write`.
- Structured storage—if the storage being accessed is collected as tables or data structures, algorithms can be more conveniently written if access to the file is treated just as though the tables were in memory.

  Previously, programs could not simply make storage or table alterations in memory and save them for access in subsequent runs; however, when the addresses of the table are defined by mappings to a file, then changes to the storage *are* changes to the file, and are thus automatically recorded in it.
- Scattered storage—if a program requires scattered regions of storage, such as multiple heaps or stack areas, such areas can be defined by mapping operations during program operation.

The remainder of this section illustrates some other concepts surrounding mapping creation and use.

Mapping `/dev/zero` gives the calling program a block of zero-filled virtual memory of the size specified in the call to `mmap`. `/dev/zero` is a special device, that responds to `read` as an infinite source of bytes with the value 0, but when mapped creates an unnamed object to back the mapped region of memory.

The following code fragment demonstrates a use of this to create a block of scratch storage in a program, at an address that the system chooses.

```
/*
 * Function to allocate a block of zeroed storage. Parameter is the
 * number of bytes desired. The storage is mapped as MAP_SHARED, so
 * that if a fork occurs, the child process will be able to access
 * and modify the storage. If we wished to cause the child's
 * modifications (as well as those by the parent) to be invisible to
 * the ancestry of processes, we would use MAP_PRIVATE.
 */
caddr_t
get_zero_storage(int len);
{
    int fd;
    caddr_t result;

    if ((fd = open("/dev/zero", O_RDWR)) == -1)
        return ((caddr_t)-1);
    result = mmap(0, len, PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0);
    (void) close(fd);
    return (result);
}
```

As written, this function permits a hierarchy of processes to use the area of allocated storage as a region of communication (for *implicit* interprocess communication purposes).

In some cases, devices or files are useful only if accessed via mapping. An example of this is frame buffer devices used to support bit-mapped displays, where display management algorithms function best if they can operate randomly on the addresses of the display directly.

Finally, it is important to remember that mappings can be operated upon at the granularity of a single page. Even though a mapping operation may define multiple pages of an address space, there is *absolutely* no restriction that subsequent operations on those addresses must operate on the same number of pages.

For instance, an `mmap` operation defining ten pages of an address space may be followed by subsequent `munmap` (see below) operations that remove every other page from the address space, leaving five mapped pages each followed by an unmapped page.

Those unmapped pages may subsequently be mapped to different locations in the same or different objects, or the whole range of pages (or any partition, superset, or subset of the pages) used in other `mmap` or other memory management operations.

Further, any mapping operation that operates on more than a single page can partially succeed in that some parts of the address range can be affected even though the call returns an overall failure.

Thus, an `mmap` operation that replaces another mapping, if it fails, might have deleted the previous mapping and failed to replace it. Similarly, other operations (unless specifically stated otherwise) might process some pages in the range successfully before operating on a page where the operation fails.

Not all device drivers support memory mapping. `mmap` fails if you try to map a device that does not support mapping.

Removing Mappings

```
int
munmap(caddr_t addr, size_t len);
```

`munmap` removes all mappings for pages in the range *[addr, addr + len)* from the address space of the calling process.

It is not an error to remove mappings from addresses that do not have them, and *any* mapping, no matter how it was established, can be removed with `munmap`. `munmap` does not in any way affect the objects that were mapped at those addresses.

Cache Control

The UNIX memory management system can be thought of as a form of "cache management," in which processor primary memory is used as a cache for pages from objects from the system virtual memory. Thus, there are a number of operations that control or interrogate the status of this cache, as described in this section

```
int
mincore(caddr_t addr, size_t len, char *vec);
```

`mincore` determines the residency of the memory pages in the address space covered by mappings in the range *[addr, addr + len)*.

Using the cache concept described earlier, this function can be viewed as an operation that interrogates the status of the cache, and returns an indication of what is currently resident in the cache. The status is returned as a char-per-page in the character array referenced by **vec* (which the system assumes to be large enough to encompass all the pages in the address range).

The low order bit of each character contains either a 1 (indicating that the page is resident in the system's primary storage), or a 0 (indicating that the page is not resident in primary storage). Other bits in the character are reserved for possible future expansion—therefore, programs testing residency should test only the least significant bit of each character.

Because the status of a page can change after `mincore` checks it, but before `mincore` returns the information, returned information might be outdated. Only locked pages are guaranteed to remain in memory

```
int
mlock(caddr_t addr, size_t len);

int
munlock(caddr_t addr, size_t len);
```

`mlock` causes the pages referenced by the mapping in the range *[addr, addr + len)* to be locked in physical memory. References to those pages (through mappings in this or other processes) will not result in page faults that require an I/O operation to obtain the data needed to satisfy the reference.

Because this operation ties up physical system resources and has the potential to disrupt normal system operation, use of this facility is restricted to the superuser. The system will not permit more than a configuration-dependent limit of pages to be locked in memory simultaneously. The call to `mlock` fails if this limit is exceeded.

`munlock` releases the locks on physical pages. Note that if multiple `mlock` calls are made through the same mapping, only a single `munlock` call is required to release the locks (in other words, locks on a given mapping do not nest).

However, if different mappings to the same pages are processed with `mlock`, then the pages will not be unlocked until the locks on all the mappings are released.

Locks are also released when a mapping is removed, either through being replaced with an `mmap` operation or removed explicitly with `munmap`.

A lock will be transferred between pages on the "copy-on-write" event associated with a MAP_PRIVATE mapping, thus locks on an address range that includes MAP_PRIVATE mappings will be retained transparently along with the copy-on-write redirection (see mmap above for a discussion of this redirection)

```
int
mlockall(int flags);

int
munlockall(void);
```

mlockall and munlockall are similar in purpose and restriction to mlock and munlock, except that they operate on entire address spaces. mlockall accepts a *flags* argument built as a bit-field of values from the set:

MCL_CURRENT	Current mappings
MCL_FUTURE	Future mappings

If *flags* is MCL_CURRENT, the lock is to affect everything currently in the address space. If *flags* is MCL_FUTURE, the lock is to affect everything added in the future. If *flags* is (MCL_CURRENT | MCL_FUTURE), the lock is to affect both current and future mappings.

munlockall removes all locks on all pages in the address space, whether established by mlock or mlockall

```
int
msync(caddr_t addr, size_t len, int flags);
```

msync supports applications that require assertions about the integrity of data in the storage backing their mapping, either for correctness or for coherent communications in a distributed environment.

msync causes all modified copies of pages over the range *[addr, addr + len)* to be flushed to the objects mapped by those addresses. In the cache analogy discussed previously, msync is the cache "write-back," or flush, operation. It is similar in purpose to the fsync operation for files.

msync optionally invalidates each such cache entry so that the first subsequent reference to the page causes the system to obtain it from its permanent storage location.

The *flags* argument provides a bit field of values that influences the behavior of msync. The bit names and their interpretations are:

`MS_SYNC`	synchronized write
`MS_ASYNC`	return immediately
`MS_INVALIDATE`	invalidate caches

`MS_SYNC` causes `msync` to return only after all I/O operations are complete. `MS_ASYNC` causes `msync` to return immediately once all I/O operations are scheduled. `MS_INVALIDATE` causes all cached copies of data from mapped objects to be invalidated, requiring them to be obtained again from object storage upon the next reference.

Other Mapping Functions

```
long
sysconf(_SC_PAGESIZE);
```

`sysconf` returns the system-dependent size of a memory page. For portability, applications should not embed any constants specifying the size of a page, and instead should make use of `sysconf` to obtain that information.

Note that it is not unusual for page sizes to vary even among implementations of the same instruction set, increasing the importance of using this function for portability.

```
int
mprotect(caddr_t addr, size_t len, int prot);
```

`mprotect` has the effect of assigning protection *prot* to all pages in the range of [*addr, addr* + *len*). The protection assigned cannot exceed the permissions allowed on the underlying object.

For instance, a read-only mapping to a file that was opened for read-only access cannot be set to be writable with `mprotect` (unless the mapping is of the `MAP_PRIVATE` type, in which case the write access is permitted since the writes will modify copies of pages from the object, and not the object itself).

Address Space Layout

Traditionally, the address space of a UNIX process has consisted of exactly three segments: one each for write-protected program code (text), a heap of dynamically allocated storage (data), and the process stack. Text is read-only and shared, while the data and stack segments are private to the process.

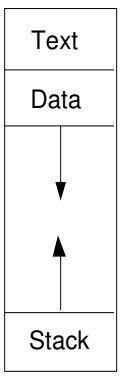

Figure 6-1 Traditional UNIX System Address-Space Layout

In the SunOS 5.x system, a process's address space is simply a vector of pages, and the division between different address-space segments is not so clear-cut. Process text and data spaces are simply groups of pages.[1]

There are often multiple text and data *segments*, some belonging to specific programs and some belonging to code running in shared libraries. The following figure illustrates one possible address space layout.

1. For compatibility, the system maintains address ranges that *should* belong to such segments to support operations such as extending or contracting the data segment's *break*. These are initialized when a program is initiated with `execve()`.

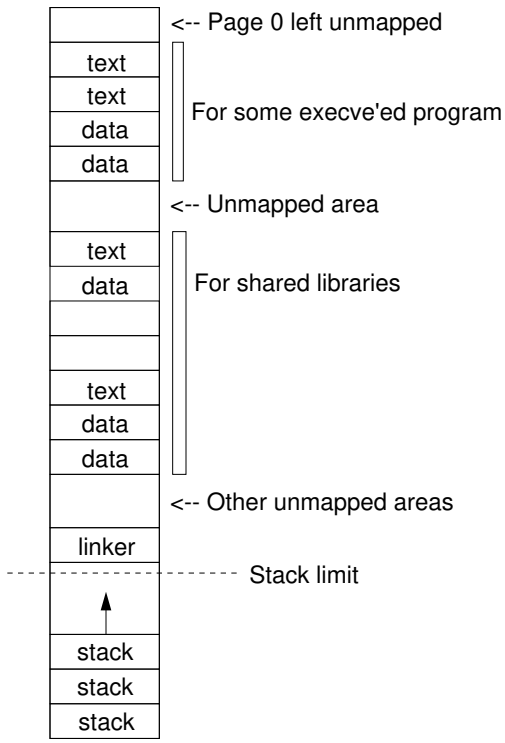

Figure 6-2 Address-Space Layout

Although the system still uses text, data, and stack segments, these should be thought of as constructs provided by the programming environment rather than by the operating system.

As such, it is possible to construct processes that have multiple segments of each *type*, or of types of arbitrary semantic value—programs no longer need to be built only from objects the system can represent directly.

For instance, a process address space may contain multiple text and data segments, some belonging to specific programs and some shared among multiple programs. Text segments from shared libraries, for example, typically appear in the address spaces of many processes.

A process address space is simply a vector of pages, and there is no necessary division between different address space segments. Process text and data spaces are simply groups of pages mapped in ways appropriate to the function they provide the program.

A process address space is usually sparsely populated, with data and text pages intermingled. The precise mechanics of the management of stack space is machine-dependent.

By convention, page 0 is not used. Process address spaces are often constructed through dynamic linking when a program is `exec`'d. Operations such as `exec` and dynamic linking build upon the mapping operations described previously.

Although the system can have multiple areas that can be considered "data" segments, for programming convenience the system maintains operations to operate on an area of storage associated with a process initial "heap storage area."

A process can manipulate this area by calling `brk` and `sbrk`:

```
caddr_t
brk(caddr_t addr);

caddr_t
sbrk(int incr);
```

`brk` sets the system idea of the lowest data segment location not used by the caller to *addr* (rounded up to the next multiple of the system page size).

`sbrk`, the alternate function, adds *incr* bytes to the caller data space and returns a pointer to the start of the new data area.