and updating the page table. When the page fault handler returns, the CPU restarts the faulting instruction, which sends *A* to the MMU again. This time, the MMU translates *A* normally, without generating a page fault.

9.8 Memory Mapping

Linux initializes the contents of a virtual memory area by associating it with an *object* on disk, a process known as *memory mapping*. Areas can be mapped to one of two types of objects:

1. *Regular file in the Linux file system:* An area can be mapped to a contiguous section of a regular disk file, such as an executable object file. The file section is divided into page-size pieces, with each piece containing the initial contents of a virtual page. Because of demand paging, none of these virtual pages is actually swapped into physical memory until the CPU first *touches* the page (i.e., issues a virtual address that falls within that page's region of the address space). If the area is larger than the file section, then the area is padded with zeros.
2. *Anonymous file:* An area can also be mapped to an anonymous file, created by the kernel, that contains all binary zeros. The first time the CPU touches a virtual page in such an area, the kernel finds an appropriate victim page in physical memory, swaps out the victim page if it is dirty, overwrites the victim page with binary zeros, and updates the page table to mark the page as resident. Notice that no data are actually transferred between disk and memory. For this reason, pages in areas that are mapped to anonymous files are sometimes called *demand-zero pages*.

In either case, once a virtual page is initialized, it is swapped back and forth between a special *swap file* maintained by the kernel. The swap file is also known as the *swap space* or the *swap area*. An important point to realize is that at any point in time, the swap space bounds the total amount of virtual pages that can be allocated by the currently running processes.

9.8.1 Shared Objects Revisited

The idea of memory mapping resulted from a clever insight that if the virtual memory system could be integrated into the conventional file system, then it could provide a simple and efficient way to load programs and data into memory.

As we have seen, the process abstraction promises to provide each process with its own private virtual address space that is protected from errant writes or reads by other processes. However, many processes have identical read-only code areas. For example, each process that runs the Linux shell program `bash` has the same code area. Further, many programs need to access identical copies of read-only run-time library code. For example, every C program requires functions from the standard C library such as `printf`. It would be extremely wasteful for each process to keep duplicate copies of these commonly used codes in physical memory. Fortunately, memory mapping provides us with a clean mechanism for controlling how objects are shared by multiple processes.

An object can be mapped into an area of virtual memory as either a *shared object* or a *private object*. If a process maps a shared object into an area of its virtual address space, then any writes that the process makes to that area are visible to any other processes that have also mapped the shared object into their virtual memory. Further, the changes are also reflected in the original object on disk.

Changes made to an area mapped to a private object, on the other hand, are not visible to other processes, and any writes that the process makes to the area are *not* reflected back to the object on disk. A virtual memory area into which a shared object is mapped is often called a *shared area*. Similarly for a *private area*.

Suppose that process 1 maps a shared object into an area of its virtual memory, as shown in Figure 9.29(a). Now suppose that process 2 maps the same shared ob-

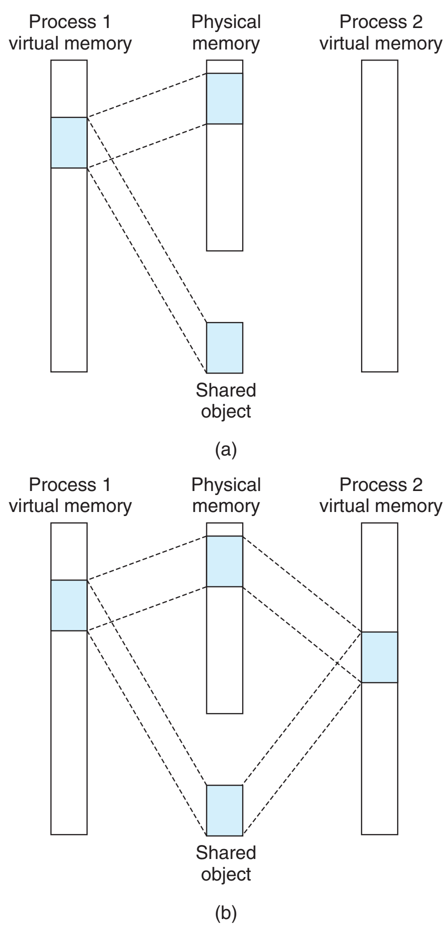

Figure 9.29
A shared object. (a) After process 1 maps the shared object. (b) After process 2 maps the same shared object. (Note that the physical pages are not necessarily contiguous.)

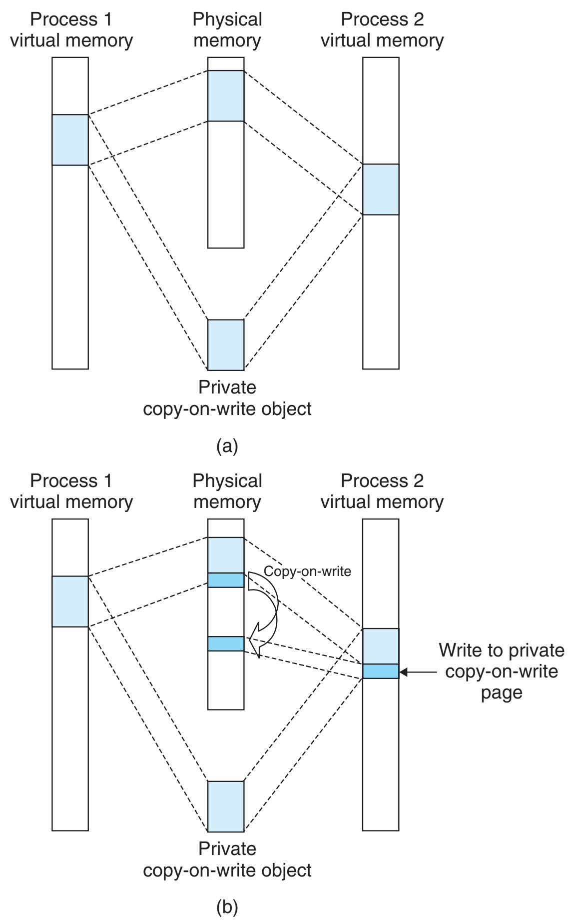

Figure 9.30

A private copy-on-write object. (a) After both processes have mapped the private copy-on-write object. (b) After process 2 writes to a page in the private area.

ject into its address space (not necessarily at the same virtual address as process 1), as shown in Figure 9.29(b).

Since each object has a unique filename, the kernel can quickly determine that process 1 has already mapped this object and can point the page table entries in process 2 to the appropriate physical pages. The key point is that only a single copy of the shared object needs to be stored in physical memory, even though the object is mapped into multiple shared areas. For convenience, we have shown the physical pages as being contiguous, but of course this is not true in general.

Private objects are mapped into virtual memory using a clever technique known as *copy-on-write*. A private object begins life in exactly the same way as a shared object, with only one copy of the private object stored in physical memory. For example, Figure 9.30(a) shows a case where two processes have mapped a private object into different areas of their virtual memories but share the same physical copy of the object. For each process that maps the private object, the page table entries for the corresponding private area are flagged as read-only, and the area struct is flagged as *private copy-on-write*. So long as neither process attempts to write to its respective private area, they continue to share a single copy of the object in physical memory. However, as soon as a process attempts to write to some page in the private area, the write triggers a protection fault.

When the fault handler notices that the protection exception was caused by the process trying to write to a page in a private copy-on-write area, it creates a new copy of the page in physical memory, updates the page table entry to point to the new copy, and then restores write permissions to the page, as shown in Figure 9.30(b). When the fault handler returns, the CPU re-executes the write, which now proceeds normally on the newly created page.

By deferring the copying of the pages in private objects until the last possible moment, copy-on-write makes the most efficient use of scarce physical memory.

9.8.2 The `fork` Function Revisited

Now that we understand virtual memory and memory mapping, we can get a clear idea of how the `fork` function creates a new process with its own independent virtual address space.

When the `fork` function is called by the *current process*, the kernel creates various data structures for the *new process* and assigns it a unique PID. To create the virtual memory for the new process, it creates exact copies of the current process's `mm_struct`, area structs, and page tables. It flags each page in both processes as read-only, and flags each area struct in both processes as private copy-on-write.

When the `fork` returns in the new process, the new process now has an exact copy of the virtual memory as it existed when the fork was called. When either of the processes performs any subsequent writes, the copy-on-write mechanism creates new pages, thus preserving the abstraction of a private address space for each process.

9.8.3 The `execve` Function Revisited

Virtual memory and memory mapping also play key roles in the process of loading programs into memory. Now that we understand these concepts, we can understand how the `execve` function really loads and executes programs. Suppose that the program running in the current process makes the following call:

```
execve("a.out", NULL, NULL);
```

As you learned in Chapter 8, the `execve` function loads and runs the program contained in the executable object file `a.out` within the current process, effectively replacing the current program with the `a.out` program. Loading and running `a.out` requires the following steps:

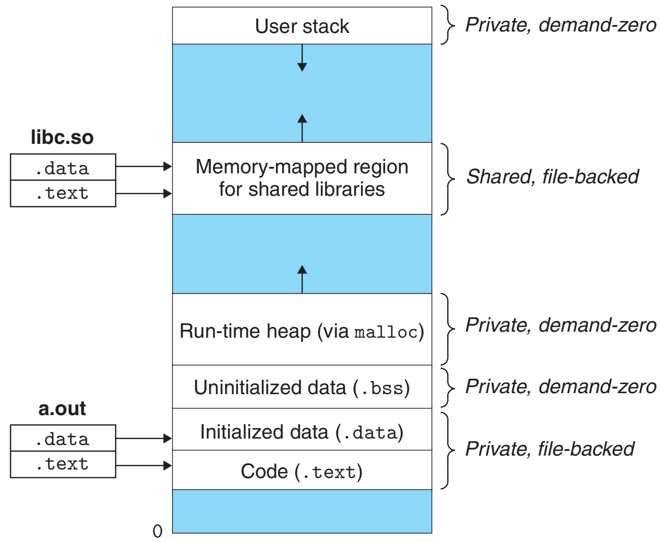

Figure 9.31
How the loader maps the areas of the user address space.

1. *Delete existing user areas.* Delete the existing area structs in the user portion of the current process's virtual address.
2. *Map private areas.* Create new area structs for the code, data, bss, and stack areas of the new program. All of these new areas are private copy-on-write. The code and data areas are mapped to the `.text` and `.data` sections of the `a.out` file. The bss area is demand-zero, mapped to an anonymous file whose size is contained in `a.out`. The stack and heap area are also demand-zero, initially of zero length. Figure 9.31 summarizes the different mappings of the private areas.
3. *Map shared areas.* If the `a.out` program was linked with shared objects, such as the standard C library `libc.so`, then these objects are dynamically linked into the program, and then mapped into the shared region of the user's virtual address space.
4. *Set the program counter (PC).* The last thing that `execve` does is to set the program counter in the current process's context to point to the entry point in the code area.

The next time this process is scheduled, it will begin execution from the entry point. Linux will swap in code and data pages as needed.

9.8.4 User-Level Memory Mapping with the `mmap` Function

Linux processes can use the `mmap` function to create new areas of virtual memory and to map objects into these areas.

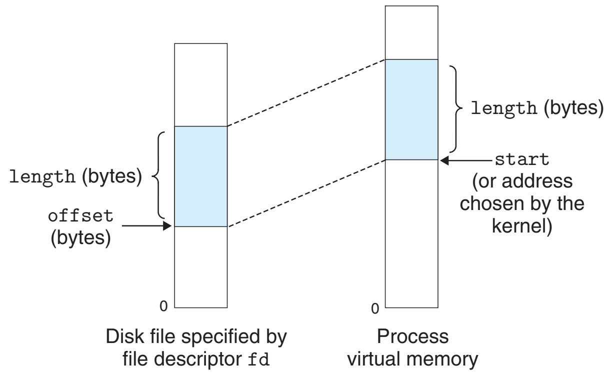

Figure 9.32
Visual interpretation of mmap arguments.

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

void  *mmap(void *start, size_t length, int prot, int flags,
            int fd, off_t offset);
```

Returns: pointer to mapped area if OK, MAP_FAILED (−1) on error

The mmap function asks the kernel to create a new virtual memory area, preferably one that starts at address start, and to map a contiguous chunk of the object specified by file descriptor fd to the new area. The contiguous object chunk has a size of length bytes and starts at an offset of offset bytes from the beginning of the file. The start address is merely a hint, and is usually specified as NULL. For our purposes, we will always assume a NULL start address. Figure 9.32 depicts the meaning of these arguments.

The prot argument contains bits that describe the access permissions of the newly mapped virtual memory area (i.e., the vm_prot bits in the corresponding area struct).

PROT_EXEC. Pages in the area consist of instructions that may be executed by the CPU.

PROT_READ. Pages in the area may be read.

PROT_WRITE. Pages in the area may be written.

PROT_NONE. Pages in the area cannot be accessed.

The flags argument consists of bits that describe the type of the mapped object. If the MAP_ANON flag bit is set, then the backing store is an anonymous object and the corresponding virtual pages are demand-zero. MAP_PRIVATE indicates a private copy-on-write object, and MAP_SHARED indicates a shared object. For example,

```
bufp = Mmap(NULL, size, PROT_READ, MAP_PRIVATE|MAP_ANON, 0, 0);
```

asks the kernel to create a new read-only, private, demand-zero area of virtual memory containing `size` bytes. If the call is successful, then `bufp` contains the address of the new area.

The `munmap` function deletes regions of virtual memory:

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

int munmap(void *start, size_t length);
```

Returns: 0 if OK, −1 on error

The `munmap` function deletes the area starting at virtual address `start` and consisting of the next `length` bytes. Subsequent references to the deleted region result in segmentation faults.

Practice Problem 9.5 (solution page 918)

(solution page 918)

Write a C program `mmapcopy.c` that uses `mmap` to copy an arbitrary-size disk file to `stdout`. The name of the input file should be passed as a command-line argument.

9.9 Dynamic Memory Allocation

While it is certainly possible to use the low-level `mmap` and `munmap` functions to create and delete areas of virtual memory, C programmers typically find it more convenient and more portable to use a *dynamic memory allocator* when they need to acquire additional virtual memory at run time.

A dynamic memory allocator maintains an area of a process's virtual memory known as the *heap* (Figure 9.33). Details vary from system to system, but without loss of generality, we will assume that the heap is an area of demand-zero memory that begins immediately after the uninitialized data area and grows upward (toward higher addresses). For each process, the kernel maintains a variable `brk` (pronounced "break") that points to the top of the heap.

An allocator maintains the heap as a collection of various-size *blocks*. Each block is a contiguous chunk of virtual memory that is either *allocated* or *free*. An allocated block has been explicitly reserved for use by the application. A free block is available to be allocated. A free block remains free until it is explicitly allocated by the application. An allocated block remains allocated until it is freed, either explicitly by the application or implicitly by the memory allocator itself.

Allocators come in two basic styles. Both styles require the application to explicitly allocate blocks. They differ about which entity is responsible for freeing allocated blocks.

- *Explicit allocators* require the application to explicitly free any allocated blocks. For example, the C standard library provides an explicit allocator called the `malloc` package. C programs allocate a block by calling the `malloc`