12

Shared Memory Introduction

12.1 Introduction

Shared memory is the fastest form of IPC available. Once the memory is mapped into the address space of the processes that are sharing the memory region, no kernel involvement occurs in passing data between the processes. What is normally required, however, is some form of synchronization between the processes that are storing and fetching information to and from the shared memory region. In Part 3, we discussed various forms of synchronization: mutexes, condition variables, read–write locks, record locks, and semaphores.

> What we mean by ''no kernel involvement'' is that the processes do not execute any system calls into the kernel to pass the data. Obviously, the kernel must establish the memory mappings that allow the processes to share the memory, and then manage this memory over time (handle page faults, and the like).

Consider the normal steps involved in the client–server file copying program that we used as an example for the various types of message passing (Figure 4.1).

- The server reads from the input file. The file data is read by the kernel into its memory and then copied from the kernel to the process.
- The server writes this data in a message, using a pipe, FIFO, or message queue. These forms of IPC normally require the data to be copied from the process to the kernel.

> We use the qualifier *normally* because Posix message queues can be implemented using memory-mapped I/O (the `mmap` function that we describe in this chapter), as we showed in Section 5.8 and as we show in the solution to Exercise 12.2. In Figure 12.1, we assume

that Posix message queues are implemented within the kernel, which is another possibility. But pipes, FIFOs, and System V message queues all involve copying the data from the process to the kernel for a `write` or `msgsnd`, or copying the data from the kernel to the process for a `read` or `msgrcv`.

- The client reads the data from the IPC channel, normally requiring the data to be copied from the kernel to the process.
- Finally, the data is copied from the client's buffer, the second argument to the `write` function, to the output file.

A total of four copies of the data are normally required. Additionally, these four copies are done between the kernel and a process, often an expensive copy (more expensive than copying data within the kernel, or copying data within a single process). Figure 12.1 depicts this movement of the data between the client and server, through the kernel.

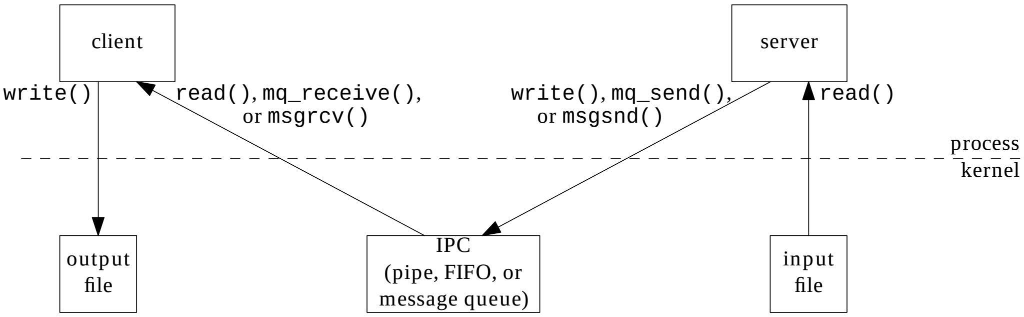

client
server
write()
read(), mq_receive(), or msgrcv()
write(), mq_send(), or msgsnd()
read()
process
kernel
output file
IPC (pipe, FIFO, or message queue)
input file

Figure 12.1 Flow of file data from server to client.

The problem with these forms of IPC—pipes, FIFOs, and message queues—is that for two processes to exchange information, the information has to go through the kernel.

Shared memory provides a way around this by letting two or more processes share a region of memory. The processes must, of course, coordinate or synchronize the use of the shared memory among themselves. (Sharing a common piece of memory is similar to sharing a disk file, such as the sequence number file used in all the file locking examples.) Any of the techniques described in Part 3 can be used for this synchronization.

The steps for the client–server example are now as follows:

- The server gets access to a shared memory object using (say) a semaphore.
- The server reads from the input file into the shared memory object. The second argument to the `read`, the address of the data buffer, points into the shared memory object.
- When the read is complete, the server notifies the client, using a semaphore.
- The client writes the data from the shared memory object to the output file.

This scenario is depicted in Figure 12.2.

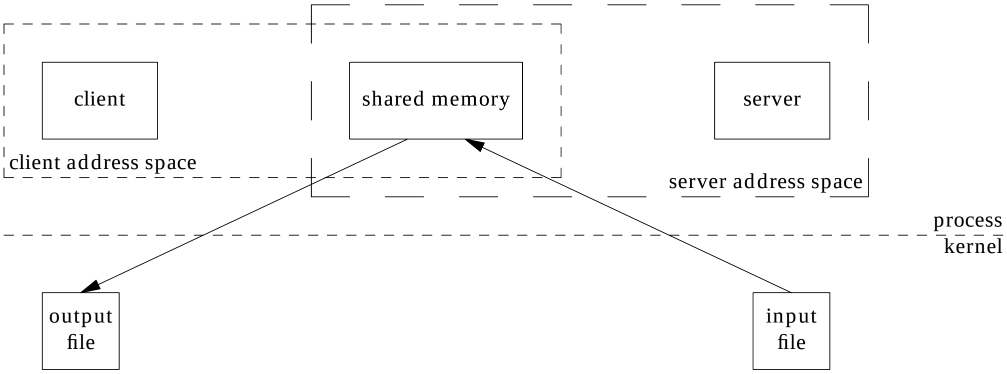


Figure 12.2 Copying file data from server to client using shared memory.

In this figure the data is copied only twice—from the input file into shared memory and from shared memory to the output file. We draw one dashed box enclosing the client and the shared memory object, and another dashed box enclosing the server and the shared memory object, to reinforce that the shared memory object appears in the address space of both the client and the server.

The concepts involved in using shared memory are similar for both the Posix interface and the System V interface. We describe the former in Chapter 13 and the latter in Chapter 14.

In this chapter, we return to our sequence-number-increment example that we started in Chapter 9. But we now store the sequence number in memory instead of in a file.

We first reiterate that memory is *not* shared by default between a parent and child across a `fork`. The program in Figure 12.3 has a parent and child increment a global integer named `count`.

Create and initialize semaphore

12–14 We create and initialize a semaphore that protects what we think is a shared variable (the global `count`). Since this assumption is false, this semaphore is not really needed. Notice that we remove the semaphore name from the system by calling `sem_unlink`, but although this removes the pathname, it has no effect on the semaphore that is already open. We do this so that the pathname is removed from the filesystem even if the program aborts.

Set standard output unbuffered and `fork`

15 We set standard output unbuffered because both the parent and child will be writing to it. This prevents interleaving of the output from the two processes.

16–29 The parent and child each execute a loop that increments the counter the specified number of times, being careful to increment the variable only when the semaphore is held.

shm/incr1.c

```
#include    "unpipc.h"

#define SEM_NAME    "mysem"

int     count = 0;

int
main(int argc, char **argv)
{
    int     i, nloop;
    sem_t  *mutex;

    if (argc != 2)
        err_quit("usage: incr1 <#loops>");
    nloop = atoi(argv[1]);

        /* create, initialize, and unlink semaphore */
    mutex = Sem_open(Px_ipc_name(SEM_NAME), O_CREAT | O_EXCL, FILE_MODE, 1);
    Sem_unlink(Px_ipc_name(SEM_NAME));

    setbuf(stdout, NULL);       /* stdout is unbuffered */
    if (Fork() == 0) {          /* child */
        for (i = 0; i < nloop; i++) {
            Sem_wait(mutex);
            printf("child: %d\n", count++);
            Sem_post(mutex);
        }
        exit(0);
    }
        /* parent */
    for (i = 0; i < nloop; i++) {
        Sem_wait(mutex);
        printf("parent: %d\n", count++);
        Sem_post(mutex);
    }
    exit(0);
}
```

shm/incr1.c

Figure 12.3 Parent and child both increment the same global.

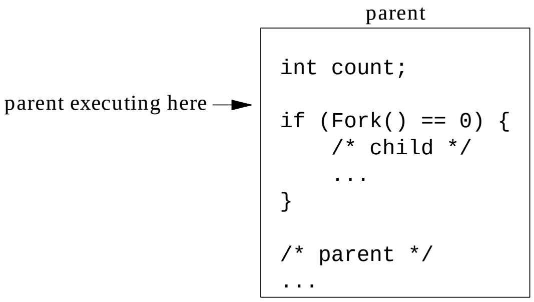
If we run this program and look only at the output when the system switches between the parent and child, we have the following:

```
child: 0              child runs first, counter starts at 0
child: 1
. . .
child: 678
child: 679
parent: 0             child is stopped, parent runs, counter starts at 0
parent: 1
. . .
parent: 1220
parent: 1221
child: 680            parent is stopped, child runs
```

```
child: 681
. . .
child: 2078
child: 2079
parent: 1222          child is stopped, parent runs
parent: 1223

                      and so on
```

As we can see, both processes have their own copy of the global `count`. Each starts with the value of this variable as 0, and each increments its own copy of this variable. Figure 12.4 shows the parent before calling `fork`.



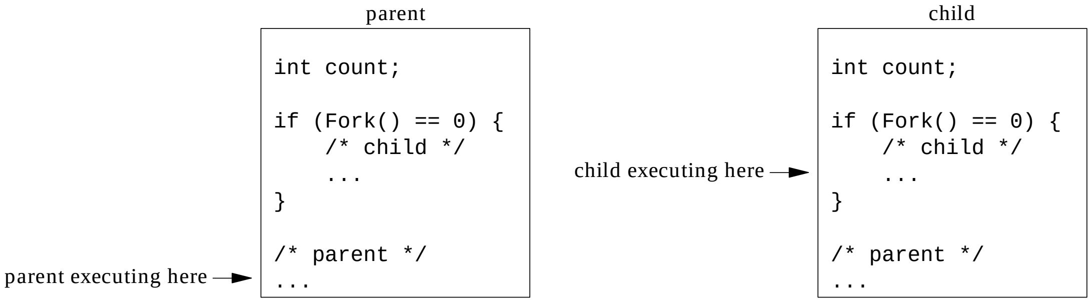
```
int count;

if (Fork() == 0) {
    /* child */
    ...
}

/* parent */
...
```

Figure 12.4 Parent before calling `fork`.

When `fork` is called, the child starts with its own copy of the parent's data space. Figure 12.5 shows the two processes after `fork` returns.



```
int count;

if (Fork() == 0) {
    /* child */
    ...
}

/* parent */
...
```

```
int count;

if (Fork() == 0) {
    /* child */
    ...
}

/* parent */
...
```

Figure 12.5 Parent and child after `fork` returns.

We see that the parent and child each have their own copy of the variable `count`.

12.2 `mmap`, `munmap`, and `msync` Functions

The `mmap` function maps either a file or a Posix shared memory object into the address space of a process. We use this function for three purposes:

1. with a regular file to provide memory-mapped I/O (Section 12.3),
2. with special files to provide anonymous memory mappings (Sections 12.4 and 12.5), and
3. with `shm_open` to provide Posix shared memory between unrelated processes (Chapter 13).

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

void *mmap(void *addr, size_t len, int prot, int flags, int fd, off_t offset);

                    Returns: starting address of mapped region if OK, MAP_FAILED on error
```

addr can specify the starting address within the process of where the descriptor *fd* should be mapped. Normally, this is specified as a null pointer, telling the kernel to choose the starting address. In any case, the return value of the function is the starting address of where the descriptor has been mapped.

len is the number of bytes to map into the address space of the process, starting at *offset* bytes from the beginning of the file. Normally, *offset* is 0. Figure 12.6 shows this mapping.

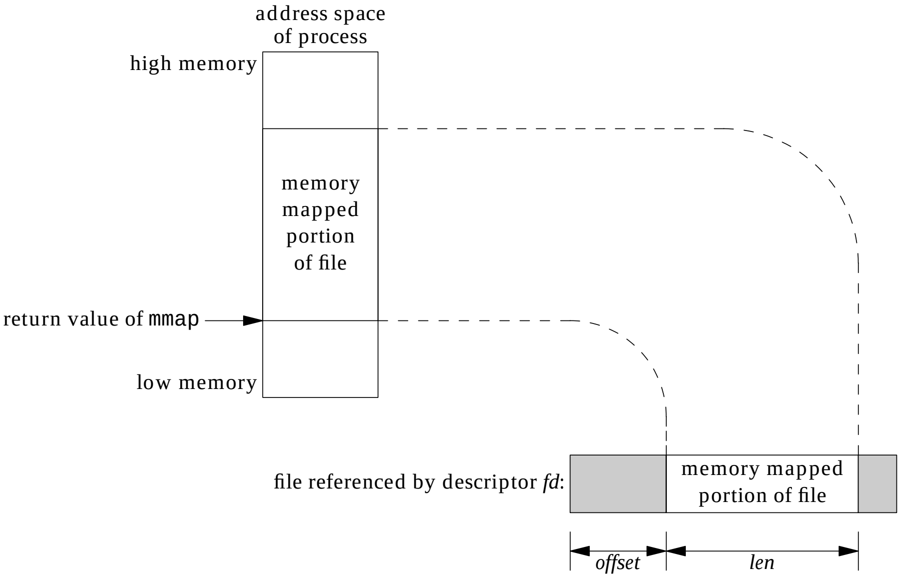


Figure 12.6 Example of memory-mapped file.

The protection of the memory-mapped region is specified by the *prot* argument using the constants in Figure 12.7. The common value for this argument is `PROT_READ | PROT_WRITE` for read–write access.

prot	Description
PROT_READ	data can be read
PROT_WRITE	data can be written
PROT_EXEC	data can be executed
PROT_NONE	data cannot be accessed

Figure 12.7 *prot* argument for mmap.h>.

flags	Description
MAP_SHARED	changes are shared
MAP_PRIVATE	changes are private
MAP_FIXED	interpret the *addr* argument exactly

Figure 12.8 *flags* argument for mmap.

The *flags* are specified by the constants in Figure 12.8. Either the MAP_SHARED or the MAP_PRIVATE flag must be specified, optionally ORed with MAP_FIXED. If MAP_PRIVATE is specified, then modifications to the mapped data by the calling process are visible only to that process and do not change the underlying object (either a file object or a shared memory object). If MAP_SHARED is specified, modifications to the mapped data by the calling process are visible to all processes that are sharing the object, and these changes do modify the underlying object.

For portability, MAP_FIXED should not be specified. If it is not specified, but *addr* is not a null pointer, then it is implementation dependent as to what the implementation does with *addr*. The nonnull value of *addr* is normally taken as a hint about where the memory should be located. Portable code should specify *addr* as a null pointer and should not specify MAP_FIXED.

One way to share memory between a parent and child is to call mmap with MAP_SHARED before calling fork. Posix.1 then guarantees that memory mappings in the parent are retained in the child. Furthermore, changes made by the parent are visible to the child and vice versa. We show an example of this shortly.

After mmap returns success, the *fd* argument can be closed. This has no effect on the mapping that was established by mmap.

To remove a mapping from the address space of the process, we call munmap.

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

int munmap(void *addr, size_t len);
```

Returns: 0 if OK, −1 on error

The *addr* argument is the address that was returned by mmap, and the *len* is the size of that mapped region. Further references to these addresses result in the generation of a SIGSEGV signal to the process (assuming, of course, that a later call to mmap does not reuse this portion of the address space).

If the mapped region was mapped using MAP_PRIVATE, the changes made are discarded.

In Figure 12.6, the kernel's virtual memory algorithm keeps the memory-mapped file (typically on disk) synchronized with the memory-mapped region in memory, assuming a MAP_SHARED segment. That is, if we modify a location in memory that is memory-mapped to a file, then at some time later the kernel will update the file accordingly. But sometimes, we want to make certain that the file on disk corresponds to what is in the memory-mapped region, and we call msync to perform this synchronization.

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

int msync(void *addr, size_t len, int flags);

                                    Returns: 0 if OK, −1 on error
```

The *addr* and *len* arguments normally refer to the entire memory-mapped region of memory, although subsets of this region can also be specified. The *flags* argument is formed from the combination of constants shown in Figure 12.9.

Constant	Description
MS_ASYNC	perform asynchronous writes
MS_SYNC	perform synchronous writes
MS_INVALIDATE	invalidate cached data

Figure 12.9 *flags* for msync function.

One of the two constants MS_ASYNC and MS_SYNC must be specified, but not both. The difference in these two is that MS_ASYNC returns once the write operations are queued by the kernel, whereas MS_SYNC returns only after the write operations are complete. If MS_INVALIDATE is also specified, all in-memory copies of the file data that are inconsistent with the file data are invalidated. Subsequent references will obtain data from the file.

Why Use mmap?

Our description of mmap so far has implied a memory-mapped file: some file that we open and then map into our address space by calling mmap. The nice feature in using a memory-mapped file is that all the I/O is done under the covers by the kernel, and we just write code that fetches and stores values in the memory-mapped region. We never call read, write, or lseek. Often, this can simplify our code.

> Recall our implementation of Posix message queues using mmap and the storing of values into a msg_hdr structure in Figure 5.30 and the fetching of values from a msg_hdr structure in Figure 5.32.

Beware of some caveats, however, in that not all files can be memory mapped. Trying to map a descriptor that refers to a terminal or a socket, for example, generates an error return from `mmap`. These types of descriptors must be accessed using `read` and `write` (or variants thereof).

Another use of `mmap` is to provide shared memory between unrelated processes. In this case, the actual contents of the file become the initial contents of the memory that is shared, and any changes made by the processes to this shared memory are then copied back to the file (providing filesystem persistence). This assumes that `MAP_SHARED` is specified, which is required to share the memory between processes.

Details on the implementation of `mmap` and its relationship to the kernel's virtual memory algorithms are provided in [McKusick et al. 1996] for 4.4BSD and in [Vahalia 1996] and [Goodheart and Cox 1994] for SVR4.

12.3 Increment Counter in a Memory-Mapped File

We now modify Figure 12.3 (which did not work) so that the parent and child share a piece of memory in which the counter is stored. To do so, we use a memory-mapped file: a file that we `open` and then `mmap` into our address space. Figure 12.10 shows the new program.

New command-line argument

11–14 We have a new command-line argument that is the name of a file that will be memory mapped. We open the file for reading and writing, creating the file if it does not exist, and then write an integer with a value of 0 to the file.

`mmap` then close descriptor

15–16 We call `mmap` to map the file that was just opened into the memory of this process. The first argument is a null pointer, telling the system to pick the starting address. The length is the size of an integer, and we specify read–write access. By specifying a fourth argument of `MAP_SHARED`, any changes made by the parent will be seen by the child, and vice versa. The return value is the starting address of the memory region that will be shared, and we store it in `ptr`.

`fork`

20–34 We set standard output unbuffered and call `fork`. The parent and child both increment the integer counter pointed to by `ptr`.

Memory-mapped files are handled specially by `fork`, in that memory mappings created by the parent before calling `fork` are shared by the child. Therefore, what we have done by opening the file and calling `mmap` with the `MAP_SHARED` flag is provide a piece of memory that is shared between the parent and child. Furthermore, since the shared memory is a memory-mapped file, any changes to the shared memory (the piece of memory pointed to by `ptr` of size `sizeof(int)`) are also reflected in the actual file (whose name was specified by the command-line argument).

shm/incr2.c

```
#include    "unpipc.h"

#define SEM_NAME    "mysem"

int
main(int argc, char **argv)
{
    int     fd, i, nloop, zero = 0;
    int    *ptr;
    sem_t  *mutex;

    if (argc != 3)
        err_quit("usage: incr2 <pathname> <#loops>");
    nloop = atoi(argv[2]);

        /* open file, initialize to 0, map into memory */
    fd = Open(argv[1], O_RDWR | O_CREAT, FILE_MODE);
    Write(fd, &zero, sizeof(int));
    ptr = Mmap(NULL, sizeof(int), PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
    Close(fd);

        /* create, initialize, and unlink semaphore */
    mutex = Sem_open(Px_ipc_name(SEM_NAME), O_CREAT | O_EXCL, FILE_MODE, 1);
    Sem_unlink(Px_ipc_name(SEM_NAME));

    setbuf(stdout, NULL);       /* stdout is unbuffered */
    if (Fork() == 0) {          /* child */
        for (i = 0; i < nloop; i++) {
            Sem_wait(mutex);
            printf("child: %d\n", (*ptr)++);
            Sem_post(mutex);
        }
        exit(0);
    }
        /* parent */
    for (i = 0; i < nloop; i++) {
        Sem_wait(mutex);
        printf("parent: %d\n", (*ptr)++);
        Sem_post(mutex);
    }
    exit(0);
}
```

shm/incr2.c

Figure 12.10 Parent and child incrementing a counter in shared memory.

If we execute this program, we see that the memory pointed to by `ptr` is indeed shared between the parent and child. We show only the values when the kernel switches between the two processes.

```
solaris % incr2 /tmp/temp.1 10000
child: 0              child starts first
child: 1
. . .
child: 128
child: 129
parent: 130           child is stopped, parent starts
```

```
parent: 131
. . .
parent: 636
parent: 637
child: 638                parent is stopped, child starts
child: 639
. . .
child: 1517
child: 1518
parent: 1519              child is stopped, parent starts
parent: 1520
. . .
parent: 19999             final line of output
solaris % od -D /tmp/temp.1
0000000 0000020000
0000004
```

Since the file was memory mapped, we can look at the file after the program terminates with the od program and see that the final value of the counter (20,000) is indeed stored in the file.

Figure 12.11 is a modification of Figure 12.5 showing the shared memory, and showing that the semaphore is also shared. We show the semaphore as being in the kernel, but as we mentioned with Posix semaphores, this is not a requirement. Whatever implementation is used, the semaphore must have at least kernel persistence. The semaphore could be stored as another memory-mapped file, as we demonstrated in Section 10.15.

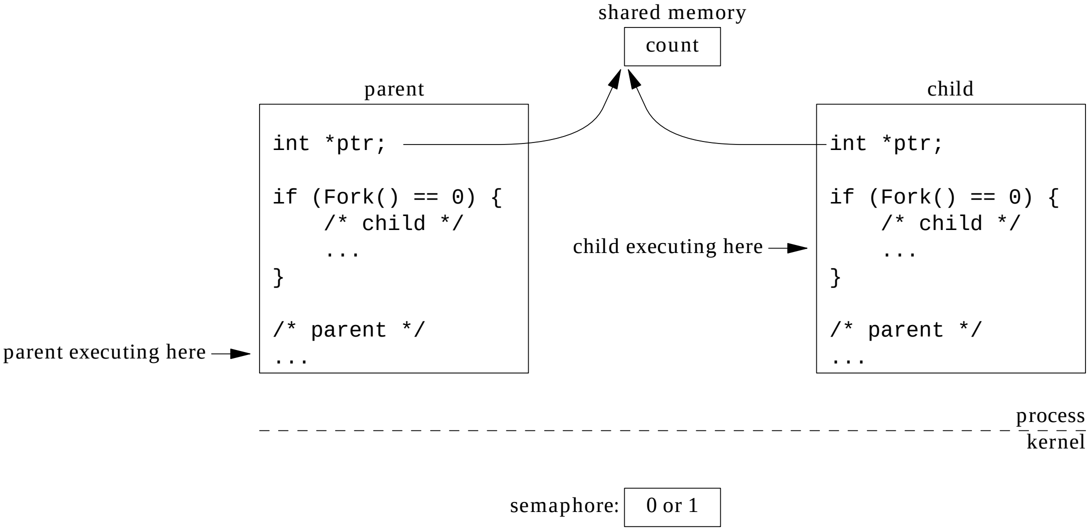


Figure 12.11 Parent and child sharing memory and a semaphore.

We show that the parent and child each have their own copy of the pointer ptr, but each copy points to the same integer in shared memory: the counter that both processes increment.

We now modify our program from Figure 12.10 to use a Posix memory-based semaphore instead of a Posix named semaphore, and store this semaphore in the shared memory. Figure 12.12 is the new program.

shm/incr3.c

```
 1 #include    "unpipc.h"

 2 struct shared {
 3     sem_t   mutex;              /* the mutex: a Posix memory-based semaphore */
 4     int     count;              /* and the counter */
 5 } shared;

 6 int
 7 main(int argc, char **argv)
 8 {
 9     int     fd, i, nloop;
10     struct shared *ptr;

11     if (argc != 3)
12         err_quit("usage: incr3 <pathname> <#loops>");
13     nloop = atoi(argv[2]);

14         /* open file, initialize to 0, map into memory */
15     fd = Open(argv[1], O_RDWR | O_CREAT, FILE_MODE);
16     Write(fd, &shared, sizeof(struct shared));
17     ptr = Mmap(NULL, sizeof(struct shared), PROT_READ | PROT_WRITE,
18                MAP_SHARED, fd, 0);
19     Close(fd);

20         /* initialize semaphore that is shared between processes */
21     Sem_init(&ptr->mutex, 1, 1);

22     setbuf(stdout, NULL);       /* stdout is unbuffered */
23     if (Fork() == 0) {          /* child */
24         for (i = 0; i < nloop; i++) {
25             Sem_wait(&ptr->mutex);
26             printf("child: %d\n", ptr->count++);
27             Sem_post(&ptr->mutex);
28         }
29         exit(0);
30     }
31         /* parent */
32     for (i = 0; i < nloop; i++) {
33         Sem_wait(&ptr->mutex);
34         printf("parent: %d\n", ptr->count++);
35         Sem_post(&ptr->mutex);
36     }
37     exit(0);
38 }
```

shm/incr3.c

Figure 12.12 Counter and semaphore are both in shared memory.

Define structure that will be in shared memory

2–5 We define a structure containing the integer counter and a semaphore to protect it. This structure will be stored in the shared memory object.

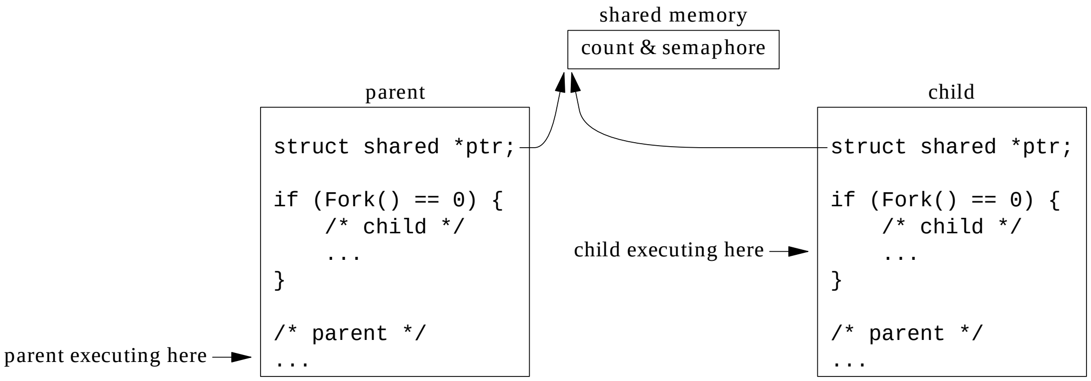
Map the memory

14–19 We create the file that will be mapped, and write a structure of 0 to the file. All we are doing is initializing the counter, because the value of the semaphore will be initialized by the call to `sem_init`. Nevertheless, writing an entire structure of 0 is simpler than to try to write only an integer of 0.

Initialize semaphore

20–21 We are now using a memory-based semaphore, instead of a named semaphore, so we call `sem_init` to initialize its value to 1. The second argument must be nonzero, to indicate that the semaphore is being shared between processes.

Figure 12.13 is a modification of Figure 12.11, noting the change that the semaphore has moved from the kernel into shared memory.



```
struct shared *ptr;

if (Fork() == 0) {
    /* child */
    ...
}

/* parent */
...
```

```
struct shared *ptr;

if (Fork() == 0) {
    /* child */
    ...
}

/* parent */
...
```

Figure 12.13 Counter and semaphore are now in shared memory.

12.4 4.4BSD Anonymous Memory Mapping

Our examples in Figures 12.10 and 12.12 work fine, but we have to create a file in the filesystem (the command-line argument), call `open`, and then `write` zeros to the file to initialize it. When the purpose of calling `mmap` is to provide a piece of mapped memory that will be shared across a `fork`, we can simplify this scenario, depending on the implementation.

1. 4.4BSD provides *anonymous memory mapping*, which completely avoids having to create or open a file. Instead, we specify the *flags* as `MAP_SHARED | MAP_ANON` and the *fd* as −1. The *offset* is ignored. The memory is initialized to 0. We show an example of this in Figure 12.14.
2. SVR4 provides `/dev/zero`, which we `open`, and we use the resulting descriptor in the call to `mmap`. This device returns bytes of 0 when read, and anything written to the device is discarded. We show an example of this in Figure 12.15.

(Many Berkeley-derived implementations, such as SunOS 4.1.x and BSD/OS 3.1, also support /dev/zero.)

Figure 12.14 shows the only portion of Figure 12.10 that changes when we use 4.4BSD anonymous memory mapping.

shm/incr_map_anon.c

```
 3 int
 4 main(int argc, char **argv)
 5 {
 6     int     i, nloop;
 7     int    *ptr;
 8     sem_t  *mutex;

 9     if (argc != 2)
10         err_quit("usage: incr_map_anon <#loops>");
11     nloop = atoi(argv[1]);

12         /* map into memory */
13     ptr = Mmap(NULL, sizeof(int), PROT_READ | PROT_WRITE,
14                MAP_SHARED | MAP_ANON, -1, 0);
```

shm/incr_map_anon.c

Figure 12.14 4.4BSD anonymous memory mapping.

6–11 The automatic variables fd and zero are gone, as is the command-line argument that specified the pathname that was created.

12–14 We no longer open a file. The MAP_ANON flag is specified in the call to mmap, and the fifth argument (the descriptor) is −1.

12.5 SVR4 /dev/zero Memory Mapping

Figure 12.15 shows the only portion of Figure 12.10 that changes when we map /dev/zero.

shm/incr_dev_zero.c

```
int
main(int argc, char **argv)
{
    int     fd, i, nloop;
    int    *ptr;
    sem_t  *mutex;

    if (argc != 2)
        err_quit("usage: incr_dev_zero <#loops>");
    nloop = atoi(argv[1]);

        /* open /dev/zero, map into memory */
    fd = Open("/dev/zero", O_RDWR);
    ptr = Mmap(NULL, sizeof(int), PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
    Close(fd);
```

shm/incr_dev_zero.c

Figure 12.15 SVR4 memory mapping of /dev/zero.

6–11 The automatic variable `zero` is gone, as is the command-line argument that specified the pathname that was created.

12–15 We `open /dev/zero`, and the descriptor is then used in the call to `mmap`. We are guaranteed that the memory-mapped region is initialized to 0.

12.6 Referencing Memory-Mapped Objects

When a regular file is memory mapped, the size of the mapping in memory (the second argument to `mmap`) normally equals the size of the file. For example, in Figure 12.12 the file size is set to the size of our `shared` structure by `write`, and this value is also the size of the memory mapping. But these two sizes—the file size and the memory-mapped size—can differ.

We will use the program shown in Figure 12.16 to explore the `mmap` function in more detail.

shm/test1.c

```
#include    "unpipc.h"

int
main(int argc, char **argv)
{
    int     fd, i;
    char   *ptr;
    size_t  filesize, mmapsize, pagesize;

    if (argc != 4)
        err_quit("usage: test1 <pathname> <filesize> <mmapsize>");
    filesize = atoi(argv[2]);
    mmapsize = atoi(argv[3]);

        /* open file: create or truncate; set file size */
    fd = Open(argv[1], O_RDWR | O_CREAT | O_TRUNC, FILE_MODE);
    Lseek(fd, filesize - 1, SEEK_SET);
    Write(fd, "", 1);

    ptr = Mmap(NULL, mmapsize, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
    Close(fd);

    pagesize = Sysconf(_SC_PAGESIZE);
    printf("PAGESIZE = %ld\n", (long) pagesize);

    for (i = 0; i < max(filesize, mmapsize); i += pagesize) {
        printf("ptr[%d] = %d\n", i, ptr[i]);
        ptr[i] = 1;
        printf("ptr[%d] = %d\n", i + pagesize - 1, ptr[i + pagesize - 1]);
        ptr[i + pagesize - 1] = 1;
    }
    printf("ptr[%d] = %d\n", i, ptr[i]);

    exit(0);
}
```

shm/test1.c

Figure 12.16 Memory mapping when `mmap` equals file size.

Command-line arguments

8–11 The command-line arguments specify the pathname of the file that will be created and memory mapped, the size to which that file is set, and the size of the memory mapping.

Create, open, truncate file; set file size

12–15 The file being opened is created if it does not exist, or truncated to a size of 0 if it already exists. The size of the file is then set to the specified size by seeking to that size minus 1 byte and writing 1 byte.

Memory map file

16–17 The file is memory mapped, using the size specified as the final command-line argument. The descriptor is then closed.

Print page size

18–19 The page size of the implementation is obtained using `sysconf` and printed.

Read and store the memory-mapped region

20–26 The memory-mapped region is read (the first byte of each page and the last byte of each page), and the values printed. We expect the values to all be 0. We also set the first and last bytes of the page to 1. We expect one of the references to generate a signal eventually, which will terminate the program. When the `for` loop terminates, we print the first byte of the next page, expecting this to fail (assuming that the program has not already failed).

The first scenario that we show is when the file size equals the memory-mapped size, but this size is not a multiple of the page size.

```
solaris % ls -l foo
foo: No such file or directory
solaris % test1 foo 5000 5000
PAGESIZE = 4096
ptr[0] = 0
ptr[4095] = 0
ptr[4096] = 0
ptr[8191] = 0
Segmentation Fault(coredump)
solaris % ls -l foo
-rw-r--r--   1 rstevens other1      5000 Mar 20 17:18 foo
solaris % od -b -A d foo
0000000 001 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000
0000016 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000
*
0004080 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 001
0004096 001 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000
0004112 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000
*
0005000
```

The page size is 4096 bytes, and we are able to read the entire second page (indexes 4096 through 8191), but a reference to the third page (index 8192) generates `SIGSEGV`, which the shell prints as "Segmentation Fault." Even though we set `ptr[8191]` to 1, this value is not written to the file, and the file's size remains 5000. The kernel lets us read and write that portion of the final page beyond our mapping (since the kernel's memory protection works with pages), but anything that we write to this extension is not written to the file. The other 3 bytes that we set to 1, indexes 0, 4095, and 4096, are copied back to the file, which we verify with the `od` command. (The `-b` option says to print the bytes in octal, and the `-A d` option says to print the addresses in decimal.) Figure 12.17 depicts this example.

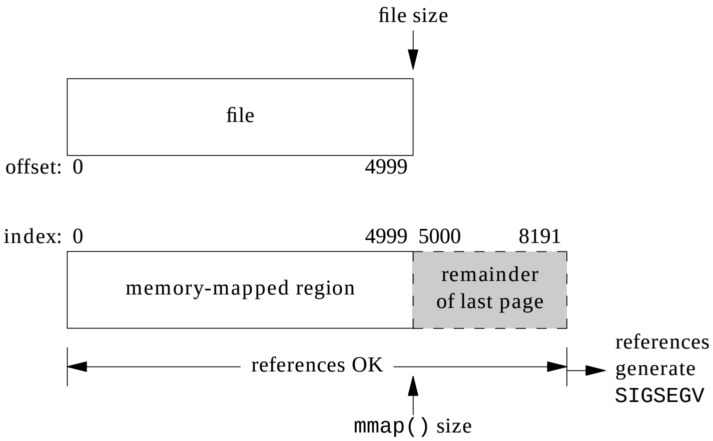


Figure 12.17 Memory mapping when `mmap` size equals file size.

If we run our example under Digital Unix, we see similar results, but the page size is now 8192.

```
alpha % ls -l foo
foo not found
alpha % test1 foo 5000 5000
PAGESIZE = 8192
ptr[0] = 0
ptr[8191] = 0
Memory fault(coredump)
alpha % ls -l foo
-rw-r--r--   1 rstevens operator    5000 Mar 21 08:40 foo
```

We are still able to reference beyond the end of our memory-mapped region but within that page of memory (indexes 5000 through 8191). Referencing `ptr[8192]` generates `SIGSEGV`, as we expect.

In our next example with Figure 12.16, we specify a memory mapping (15000 bytes) that is larger than the file size (5000 bytes).

```
solaris % rm foo
solaris % test1 foo 5000 15000
PAGESIZE = 4096
ptr[0] = 0
ptr[4095] = 0
ptr[4096] = 0
```

```
ptr[8191] = 0
Bus Error(coredump)
solaris % ls -l foo
-rw-r--r--   1 rstevens other1      5000 Mar 20 17:37 foo
```

The results are similar to our earlier example when the file size and the memory map size were the same (both 5000). This example generates SIGBUS (which the shell prints as "Bus Error"), whereas the previous example generated SIGSEGV. The difference is that SIGBUS means we have referenced within our memory-mapped region but beyond the size of the underlying object. The SIGSEGV in the previous example meant we had referenced beyond the end of our memory-mapped region. What we have shown here is that the kernel knows the size of the underlying object that is mapped (the file foo in this case), even though we have closed the descriptor for that object. The kernel allows us to specify a size to mmap that is larger than the size of this object, but we cannot reference beyond its end (except for the bytes within the final page that are beyond the end of the object, indexes 5000 through 8191). Figure 12.18 depicts this example.

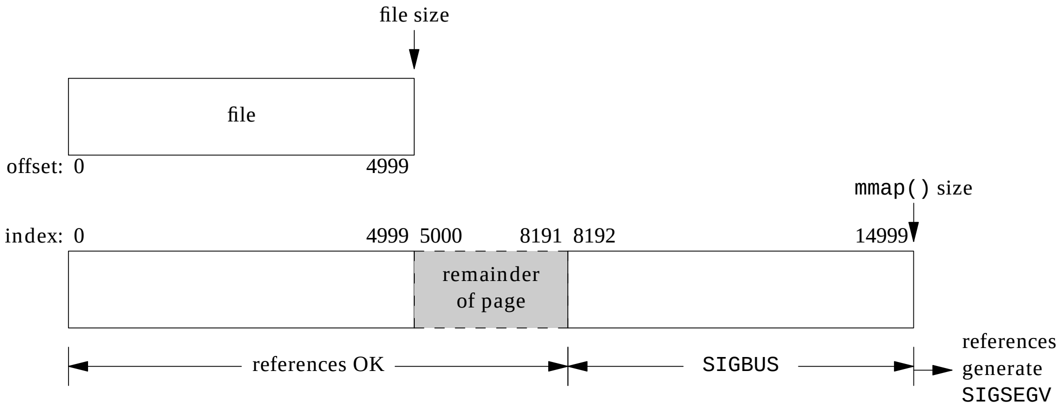


Figure 12.18 Memory mapping when mmap size exceeds file size.

Our next program is shown in Figure 12.19. It shows a common technique for handling a file that is growing: specify a memory-map size that is larger than the file, keep track of the file's current size (making certain not to reference beyond the current end-of-file), and then just let the file's size increase as more data is written to the file.

Open file

9–11 We open a file, creating it if it does not exist or truncating it if it already exists. The file is memory mapped with a size of 32768, even though the file's current size is 0.

Increase file size

12–16 We increase the size of the file, 4096 bytes at a time, by calling ftruncate (Section 13.3), and fetch the byte that is now the final byte of the file.

shm/test2.c

```
#include    "unpipc.h"

#define FILE    "test.data"
#define SIZE    32768

int
main(int argc, char **argv)
{
    int     fd, i;
    char   *ptr;

        /* open: create or truncate; then mmap file */
    fd = Open(FILE, O_RDWR | O_CREAT | O_TRUNC, FILE_MODE);
    ptr = Mmap(NULL, SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);

    for (i = 4096; i <= SIZE; i += 4096) {
        printf("setting file size to %d\n", i);
        Ftruncate(fd, i);
        printf("ptr[%d] = %d\n", i - 1, ptr[i - 1]);
    }

    exit(0);
}
```

shm/test2.c

Figure 12.19 Memory map example that lets the file size grow.

When we run this program, we see that as we increase the size of the file, we are able to reference the new data through our established memory map.

```
alpha % ls -l test.data
test.data: No such file or directory
alpha % test2
setting file size to 4096
ptr[4095] = 0
setting file size to 8192
ptr[8191] = 0
setting file size to 12288
ptr[12287] = 0
setting file size to 16384
ptr[16383] = 0
setting file size to 20480
ptr[20479] = 0
setting file size to 24576
ptr[24575] = 0
setting file size to 28672
ptr[28671] = 0
setting file size to 32768
ptr[32767] = 0
alpha % ls -l test.data
-rw-r--r--   1 rstevens other1     32768 Mar 20 17:53 test.data
```

This example shows that the kernel keeps track of the size of the underlying object that is memory mapped (the file test.data in this example), and we are always able to reference bytes that are within the current file size that are also within our memory map. We obtain identical results under Solaris 2.6.

This section has dealt with memory-mapped files and mmap. In Exercise 13.1, we modify our two programs to work with Posix shared memory and see the same results.

12.7 Summary

Shared memory is the fastest form of IPC available, because one copy of the data in the shared memory is available to all the threads or processes that share the memory. Some form of synchronization is normally required, however, to coordinate the various threads or processes that are sharing the memory.

This chapter has focused on the mmap function and the mapping of regular files into memory, because this is one way to share memory between related or unrelated processes. Once we have memory mapped a file, we no longer use read, write, or lseek to access the file; instead, we just fetch or store the memory locations that have been mapped to the file by mmap. Changing explicit file I/O into fetches and stores of memory can often simplify our programs and sometimes increase performance.

When the memory is to be shared across a subsequent fork, this can be simplified by not creating a regular file to map, but using anonymous memory mapping instead. This involves either a new flag of MAP_ANON (for Berkeley-derived kernels) or mapping /dev/zero (for SVR4-derived kernels).

Our reason for covering mmap in such detail is both because memory mapping of files is a useful technique and because mmap is used for Posix shared memory, which is the topic of the next chapter.

Also available are four additional functions (that we do not cover) defined by Posix dealing with memory management:

- mlockall causes all of the memory of the process to be memory resident. munlockall undoes this locking.
- mlock causes a specified range of addresses of the process to be memory resident, where the function arguments are a starting address and a number of bytes from that address. munlock unlocks a specified region of memory.

Exercises

12.1 What would happen in Figure 12.19 if we executed the code within the for loop one more time?

12.2 Assume that we have two processes, a sender and a receiver, with the former just sending messages to the latter. Assume that System V message queues are used and draw a diagram of how the messages go from the sender to the receiver. Now assume that our implementation of Posix message queues from Section 5.8 is used, and draw a diagram of the transfer of messages.

12.3 With mmap and MAP_SHARED, we said that the kernel virtual memory algorithm updates the actual file with any modifications that are made to the memory image. Read the manual page for /dev/zero to determine what happens when the kernel writes the changes back to the file.

12.4 Modify Figure 12.10 to specify MAP_PRIVATE instead of MAP_SHARED, and verify that the results are similar to the results from Figure 12.3. What are the contents of the file that is memory mapped?

12.5 In Section 6.9, we mentioned that one way to select on a System V message queue is to create a piece of anonymous shared memory, create a child, and let the child block in its call to msgrcv, reading the message into shared memory. The parent also creates two pipes; one is used by the child to notify the parent that a message is ready in shared memory, and the other pipe is used by the parent to notify the child that the shared memory is now available. This allows the parent to select on the read end of the pipe, along with any other descriptors on which it wants to select. Code this solution. Call our my_shm function (Figure A.46) to allocate the anonymous shared memory object. Use our msgcreate and msgsnd programs from Section 6.6 to create the message queue, and then place records onto the queue. The parent should just print the size and type of each message that the child reads.