the two with a *pipe*, represented by the "|" symbol. A pipe permits one-way communication between two related processes. The `ls` process writes data into the pipe, and the `lpr` process reads data from the pipe.

In this chapter, we discuss five types of interprocess communication:

- Shared memory permits processes to communicate by simply reading and writing to a specified memory location.
- Mapped memory is similar to shared memory, except that it is associated with a file in the filesystem.
- Pipes permit sequential communication from one process to a related process.
- FIFOs are similar to pipes, except that unrelated processes can communicate because the pipe is given a name in the filesystem.
- Sockets support communication between unrelated processes even on different computers.

These types of IPC differ by the following criteria:

- Whether they restrict communication to related processes (processes with a common ancestor), to unrelated processes sharing the same filesystem, or to any computer connected to a network
- Whether a communicating process is limited to only write data or only read data
- The number of processes permitted to communicate
- Whether the communicating processes are synchronized by the IPC—for example, a reading process halts until data is available to read

In this chapter, we omit discussion of IPC permitting communication only a limited number of times, such as communicating via a child's exit value.

5.1 Shared Memory

One of the simplest interprocess communication methods is using shared memory. Shared memory allows two or more processes to access the same memory as if they all called `malloc` and were returned pointers to the same actual memory. When one process changes the memory, all the other processes see the modification.

5.1.1 Fast Local Communication

Shared memory is the fastest form of interprocess communication because all processes share the same piece of memory. Access to this shared memory is as fast as accessing a process's nonshared memory, and it does not require a system call or entry to the kernel. It also avoids copying data unnecessarily.

Because the kernel does not synchronize accesses to shared memory, you must provide your own synchronization. For example, a process should not read from the memory until after data is written there, and two processes must not write to the same memory location at the same time. A common strategy to avoid these race conditions is to use semaphores, which are discussed in the next section. Our illustrative programs, though, show just a single process accessing the memory, to focus on the shared memory mechanism and to avoid cluttering the sample code with synchronization logic.

5.1.2 The Memory Model

To use a shared memory segment, one process must allocate the segment. Then each process desiring to access the segment must attach the segment. After finishing its use of the segment, each process detaches the segment. At some point, one process must deallocate the segment.

Understanding the Linux memory model helps explain the allocation and attachment process. Under Linux, each process's virtual memory is split into pages. Each process maintains a mapping from its memory addresses to these virtual memory pages, which contain the actual data. Even though each process has its own addresses, multiple processes' mappings can point to the same page, permitting sharing of memory. Memory pages are discussed further in Section 8.8, "The `mlock` Family: Locking Physical Memory," of Chapter 8, "Linux System Calls."

Allocating a new shared memory segment causes virtual memory pages to be created. Because all processes desire to access the same shared segment, only one process should allocate a new shared segment. Allocating an existing segment does not create new pages, but it does return an identifier for the existing pages. To permit a process to use the shared memory segment, a process attaches it, which adds entries mapping from its virtual memory to the segment's shared pages. When finished with the segment, these mapping entries are removed. When no more processes want to access these shared memory segments, exactly one process must deallocate the virtual memory pages.

All shared memory segments are allocated as integral multiples of the system's *page size*, which is the number of bytes in a page of memory. On Linux systems, the page size is 4KB, but you should obtain this value by calling the `getpagesize` function.

5.1.3 Allocation

A process allocates a shared memory segment using `shmget` ("SHared Memory GET"). Its first parameter is an integer key that specifies which segment to create. Unrelated processes can access the same shared segment by specifying the same key value. Unfortunately, other processes may have also chosen the same fixed key, which could lead to conflict. Using the special constant `IPC_PRIVATE` as the key value guarantees that a brand new memory segment is created.

Its second parameter specifies the number of bytes in the segment. Because segments are allocated using pages, the number of actually allocated bytes is rounded up to an integral multiple of the page size.

The third parameter is the bitwise or of flag values that specify options to `shmget`. The flag values include these:

- `IPC_CREAT`—This flag indicates that a new segment should be created. This permits creating a new segment while specifying a key value.
- `IPC_EXCL`—This flag, which is always used with `IPC_CREAT`, causes `shmget` to fail if a segment key is specified that already exists. Therefore, it arranges for the calling process to have an "exclusive" segment. If this flag is not given and the key of an existing segment is used, `shmget` returns the existing segment instead of creating a new one.
- Mode flags—This value is made of 9 bits indicating permissions granted to owner, group, and world to control access to the segment. Execution bits are ignored. An easy way to specify permissions is to use the constants defined in `<sys/stat.h>` and documented in the section 2 `stat` man page.[1] For example, `S_IRUSR` and `S_IWUSR` specify read and write permissions for the owner of the shared memory segment, and `S_IROTH` and `S_IWOTH` specify read and write permissions for others.

For example, this invocation of `shmget` creates a new shared memory segment (or access to an existing one, if `shm_key` is already used) that's readable and writeable to the owner but not other users.

```
int segment_id = shmget (shm_key, getpagesize (),
                         IPC_CREAT | S_IRUSR |  S_IWUSER);
```

If the call succeeds, `shmget` returns a segment identifier. If the shared memory segment already exists, the access permissions are verified and a check is made to ensure that the segment is not marked for destruction.

5.1.4 Attachment and Detachment

To make the shared memory segment available, a process must use `shmat`, "SHared Memory ATtach." Pass it the shared memory segment identifier `SHMID` returned by `shmget`. The second argument is a pointer that specifies where in your process's address space you want to map the shared memory; if you specify NULL, Linux will choose an available address. The third argument is a flag, which can include the following:

- `SHM_RND` indicates that the address specified for the second parameter should be rounded down to a multiple of the page size. If you don't specify this flag, you must page-align the second argument to `shmat` yourself.
- `SHM_RDONLY` indicates that the segment will be only read, not written.

1. These permission bits are the same as those used for files. They are described in Section 10.3, "File System Permissions."

If the call succeeds, it returns the address of the attached shared segment. Children created by calls to `fork` inherit attached shared segments; they can detach the shared memory segments, if desired.

When you're finished with a shared memory segment, the segment should be detached using `shmdt` ("SHared Memory DeTach"). Pass it the address returned by `shmat`. If the segment has been deallocated and this was the last process using it, it is removed. Calls to `exit` and any of the `exec` family automatically detach segments.

5.1.5 Controlling and Deallocating Shared Memory

The `shmctl` ("SHared Memory ConTroL") call returns information about a shared memory segment and can modify it. The first parameter is a shared memory segment identifier.

To obtain information about a shared memory segment, pass `IPC_STAT` as the second argument and a pointer to a `struct shmid_ds`.

To remove a segment, pass `IPC_RMID` as the second argument, and pass NULL as the third argument. The segment is removed when the last process that has attached it finally detaches it.

Each shared memory segment should be explicitly deallocated using `shmctl` when you're finished with it, to avoid violating the systemwide limit on the total number of shared memory segments. Invoking `exit` and `exec` detaches memory segments but does not deallocate them.

See the `shmctl` man page for a description of other operations you can perform on shared memory segments.

5.1.6 An Example Program

The program in Listing 5.1 illustrates the use of shared memory.

Listing 5.1 **(*shm.c*) Exercise Shared Memory**

```
#include <stdio.h>
#include <sys/shm.h>
#include <sys/stat.h>

int main ()
{
  int segment_id;
  char* shared_memory;
  struct shmid_ds shmbuffer;
  int segment_size;
  const int shared_segment_size = 0x6400;

  /* Allocate a shared memory segment.  */
  segment_id = shmget (IPC_PRIVATE, shared_segment_size,
                       IPC_CREAT | IPC_EXCL | S_IRUSR | S_IWUSR);
```

continues

Listing 5.1 **Continued**

```
  /* Attach the shared memory segment.  */
  shared_memory = (char*) shmat (segment_id, 0, 0);
  printf ("shared memory attached at address %p\n", shared_memory);
  /* Determine the segment's size.  */
  shmctl (segment_id, IPC_STAT, &shmbuffer);
  segment_size = shmbuffer.shm_segsz;
  printf ("segment size: %d\n", segment_size);
  /* Write a string to the shared memory segment.  */
  sprintf (shared_memory, "Hello, world.");
  /* Detach the shared memory segment.  */
  shmdt (shared_memory);

  /* Reattach the shared memory segment, at a different address.  */
  shared_memory = (char*) shmat (segment_id, (void*) 0x5000000, 0);
  printf ("shared memory reattached at address %p\n", shared_memory);
  /* Print out the string from shared memory.  */
  printf ("%s\n", shared_memory);
  /* Detach the shared memory segment.  */
  shmdt (shared_memory);

  /* Deallocate the shared memory segment.  */
  shmctl (segment_id, IPC_RMID, 0);

  return 0;
}
```

5.1.7 Debugging

The `ipcs` command provides information on interprocess communication facilities, including shared segments. Use the `-m` flag to obtain information about shared memory. For example, this code illustrates that one shared memory segment, numbered 1627649, is in use:

```
% ipcs -m

------ Shared Memory Segments --------
key        shmid      owner      perms      bytes      nattch     status
0x00000000 1627649    user     640        25600      0
```

If this memory segment was erroneously left behind by a program, you can use the `ipcrm` command to remove it.

```
% ipcrm shm 1627649
```

5.1.8 Pros and Cons

Shared memory segments permit fast bidirectional communication among any number of processes. Each user can both read and write, but a program must establish and follow some protocol for preventing race conditions such as overwriting information before it is read. Unfortunately, Linux does not strictly guarantee exclusive access even if you create a new shared segment with `IPC_PRIVATE`.

Also, for multiple processes to use a shared segment, they must make arrangements to use the same key.

5.2 Processes Semaphores

As noted in the previous section, processes must coordinate access to shared memory. As we discussed in Section 4.4.5, "Semaphores for Threads," in Chapter 4, "Threads," semaphores are counters that permit synchronizing multiple threads. Linux provides a distinct alternate implementation of semaphores that can be used for synchronizing processes (called process semaphores or sometimes System V semaphores). Process semaphores are allocated, used, and deallocated like shared memory segments. Although a single semaphore is sufficient for almost all uses, process semaphores come in sets. Throughout this section, we present system calls for process semaphores, showing how to implement single binary semaphores using them.

5.2.1 Allocation and Deallocation

The calls `semget` and `semctl` allocate and deallocate semaphores, which is analogous to `shmget` and `shmctl` for shared memory. Invoke `semget` with a key specifying a semaphore set, the number of semaphores in the set, and permission flags as for `shmget`; the return value is a semaphore set identifier. You can obtain the identifier of an existing semaphore set by specifying the right key value; in this case, the number of semaphores can be zero.

Semaphores continue to exist even after all processes using them have terminated. The last process to use a semaphore set must explicitly remove it to ensure that the operating system does not run out of semaphores. To do so, invoke `semctl` with the semaphore identifier, the number of semaphores in the set, `IPC_RMID` as the third argument, and any `union semun` value as the fourth argument (which is ignored). The effective user ID of the calling process must match that of the semaphore's allocator (or the caller must be `root`). Unlike shared memory segments, removing a semaphore set causes Linux to deallocate immediately.

Listing 5.2 presents functions to allocate and deallocate a binary semaphore.

Listing 5.2 **(*sem_all_deall.c*) Allocating and Deallocating a Binary Semaphore**

```
#include <sys/ipc.h>
#include <sys/sem.h>
#include <sys/types.h>

/* We must define union semun ourselves.  */

union semun {
  int val;
  struct semid_ds *buf;
  unsigned short int *array;
  struct seminfo *__buf;
};

/* Obtain a binary semaphore's ID, allocating if necessary.  */

int binary_semaphore_allocation (key_t key, int sem_flags)
{
  return semget (key, 1, sem_flags);
}

/* Deallocate a binary semaphore.  All users must have finished their
   use.  Returns -1 on failure.  */

int binary_semaphore_deallocate (int semid)
{
  union semun ignored_argument;
  return semctl (semid, 1, IPC_RMID, ignored_argument);
}
```

5.2.2 Initializing Semaphores

Allocating and initializing semaphores are two separate operations. To initialize a semaphore, use `semctl` with zero as the second argument and `SETALL` as the third argument. For the fourth argument, you must create a `union semun` object and point its `array` field at an array of unsigned short values. Each value is used to initialize one semaphore in the set.

Listing 5.3 presents a function that initializes a binary semaphore.

Listing 5.3 **(*sem_init.c*) Initializing a Binary Semaphore**

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
```

```
/* We must define union semun ourselves.  */

union semun {
  int val;
  struct semid_ds *buf;
  unsigned short int *array;
  struct seminfo *__buf;
};

/* Initialize a binary semaphore with a value of 1.  */

int binary_semaphore_initialize (int semid)
{
  union semun argument;
  unsigned short values[1];
  values[0] = 1;
  argument.array = values;
  return semctl (semid, 0, SETALL, argument);
}
```

5.2.3 Wait and Post Operations

Each semaphore has a non-negative value and supports wait and post operations. The `semop` system call implements both operations. Its first parameter specifies a semaphore set identifier. Its second parameter is an array of `struct sembuf` elements, which specify the operations you want to perform. The third parameter is the length of this array.

The fields of `struct sembuf` are listed here:

- `sem_num` is the semaphore number in the semaphore set on which the operation is performed.
- `sem_op` is an integer that specifies the semaphore operation.

  If `sem_op` is a positive number, that number is added to the semaphore value immediately.

  If `sem_op` is a negative number, the absolute value of that number is subtracted from the semaphore value. If this would make the semaphore value negative, the call blocks until the semaphore value becomes as large as the absolute value of `sem_op` (because some other process increments it).

  If `sem_op` is zero, the operation blocks until the semaphore value becomes zero.
- `sem_flg` is a flag value. Specify `IPC_NOWAIT` to prevent the operation from blocking; if the operation would have blocked, the call to `semop` fails instead. If you specify `SEM_UNDO`, Linux automatically undoes the operation on the semaphore when the process exits.

Listing 5.4 illustrates wait and post operations for a binary semaphore.

Listing 5.4 **(*sem_pv.c*) Wait and Post Operations for a Binary Semaphore**

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

/* Wait on a binary semaphore.  Block until the semaphore value is positive, then
   decrement it by 1.  */

int binary_semaphore_wait (int semid)
{
  struct sembuf operations[1];
  /* Use the first (and only) semaphore.  */
  operations[0].sem_num = 0;
  /* Decrement by 1.  */
  operations[0].sem_op = -1;
  /* Permit undo'ing.  */
  operations[0].sem_flg = SEM_UNDO;

  return semop (semid, operations, 1);
}

/* Post to a binary semaphore: increment its value by 1.
   This returns immediately.  */

int binary_semaphore_post (int semid)
{
  struct sembuf operations[1];
  /* Use the first (and only) semaphore.  */
  operations[0].sem_num = 0;
  /* Increment by 1.  */
  operations[0].sem_op = 1;
  /* Permit undo'ing.  */
  operations[0].sem_flg = SEM_UNDO;

  return semop (semid, operations, 1);
}
```

Specifying the `SEM_UNDO` flag permits dealing with the problem of terminating a process while it has resources allocated through a semaphore. When a process terminates, either voluntarily or involuntarily, the semaphore's values are automatically adjusted to "undo" the process's effects on the semaphore. For example, if a process that has decremented a semaphore is killed, the semaphore's value is incremented.

5.2.4 Debugging Semaphores

Use the command `ipcs -s` to display information about existing semaphore sets. Use the `ipcrm sem` command to remove a semaphore set from the command line. For example, to remove the semaphore set with identifier 5790517, use this line:

```
% ipcrm sem 5790517
```

5.3 Mapped Memory

Mapped memory permits different processes to communicate via a shared file. Although you can think of mapped memory as using a shared memory segment with a name, you should be aware that there are technical differences. Mapped memory can be used for interprocess communication or as an easy way to access the contents of a file.

Mapped memory forms an association between a file and a process's memory. Linux splits the file into page-sized chunks and then copies them into virtual memory pages so that they can be made available in a process's address space. Thus, the process can read the file's contents with ordinary memory access. It can also modify the file's contents by writing to memory. This permits fast access to files.

You can think of mapped memory as allocating a buffer to hold a file's entire contents, and then reading the file into the buffer and (if the buffer is modified) writing the buffer back out to the file afterward. Linux handles the file reading and writing operations for you.

There are uses for memory-mapped files other than interprocess communication. Some of these are discussed in Section 5.3.5, "Other Uses for `mmap`."

5.3.1 Mapping an Ordinary File

To map an ordinary file to a process's memory, use the `mmap` ("Memory MAPped," pronounced "em-map") call. The first argument is the address at which you would like Linux to map the file into your process's address space; the value NULL allows Linux to choose an available start address. The second argument is the length of the map in bytes. The third argument specifies the protection on the mapped address range. The protection consists of a bitwise "or" of `PROT_READ`, `PROT_WRITE`, and `PROT_EXEC`, corresponding to read, write, and execution permission, respectively. The fourth argument is a flag value that specifies additional options. The fifth argument is a file descriptor opened to the file to be mapped. The last argument is the offset from the beginning of the file from which to start the map. You can map all or part of the file into memory by choosing the starting offset and length appropriately.

The flag value is a bitwise "or" of these constraints:

- `MAP_FIXED`—If you specify this flag, Linux uses the address you request to map the file rather than treating it as a hint. This address must be page-aligned.
- `MAP_PRIVATE`—Writes to the memory range should not be written back to the attached file, but to a private copy of the file. No other process sees these writes. This mode may not be used with `MAP_SHARED`.