7

Advanced Interprocess Communication

7.1 Introduction

So far we've seen how to connect two processes with pipes, but only if the processes are related, and only if the pipes are created before one of the processes, since the only way the file descriptors can be used is if they were inherited. That's much too restrictive for the more realistic situation in which a server is running continuously, and clients that want to talk to it come and go. We want more flexibility in setting up the connections, more control over the information flow (e.g., queues and priorities), and more efficiency if there's a lot of data to be communicated.

Happily, today's UNIX systems have several interprocess communication (IPC) mechanisms besides the pipes we already know about: named pipes, message queues, semaphores and file locks, shared memory, and sockets. This chapter covers all those IPC mechanisms that pass data within a single system; sockets, which go between systems, are in the next chapter.

Most of the IPC mechanisms are flexible enough to accommodate a wide range of application architectures, but it's much easier for a beginner to understand them if we restrict our examples to the simple case of a single server exchanging messages with multiple clients. To make it straightforward to compare the mechanisms, I'll introduce a Simple Messaging Interface (SMI) and then implement it six different ways, using FIFOs (named pipes), System V message queues, POSIX message queues, System V shared memory and semaphores, POSIX shared memory and semaphores, and sockets (in the next chapter). This should give you a solid basis to pick the mechanisms appropriate for your own applica-

tions and then to build on as you investigate the advanced features of the mechanisms you've chosen.

With so many ways to do roughly the same thing, it's natural to wonder which is best. There are at least three dimensions to "best":

- Convenient to use and a good match for your application's needs
- Portable, in case you want to move someday from, say, HP/UX, Solaris, or AIX to Linux
- Efficient

I'll go into the first two criteria in depth as I present each IPC mechanism and summarize things in "critique" sections. Efficiency depends too much on the implementation of the mechanism on the systems of interest to you and how your application uses them, so, if feasible, you might consider implementing your own abstract layer (analogous to SMI). This would allow you to experiment with different mechanisms after your application is running. I'll provide the results of some timing tests at the end of this chapter. Of course, they're definitely not the last word, and the results you get may be different.

7.2 FIFOs, or Named Pipes

A FIFO combines features of a regular file and a pipe. Like a regular file, it has a name, and any process with appropriate permissions may open it for reading or writing. Unlike a pipe, then, unrelated processes may communicate over a FIFO since they need not rely on inheritance alone to access it. Once opened, however, a FIFO acts more like a pipe than a regular file. It follows the pipe behavior described in Section 6.2.2.

Normally, when a FIFO is opened for reading, the `open` waits until it is also opened for writing, usually by another process. Similarly, an `open` for writing blocks until the FIFO is opened for reading. Hence, whichever process, reader or writer, executes the `open` first will wait for the other one. This allows the processes to synchronize themselves before the actual data transmission starts.

If the `O_NONBLOCK` flag is set on `open`, an `open` for reading returns immediately (with an open file descriptor), without waiting for the writer, and an `open` for writing returns –1 with `errno` set to `ENXIO` if no reader has the FIFO open. This asymmetry implies that `O_NONBLOCK` is useful when opening for reading but

awkward when opening for writing since the error has to be processed and then perhaps the `open` has to be retried. The intent is to prevent a process from putting data into a FIFO that will not be read immediately because UNIX has no way of storing data in FIFOs permanently. As with a water pipe, you have no business turning on the water until both ends are soldered in. If any data is still in a FIFO when all readers and writers close their file descriptors, the data is discarded with no error indication. This is like a water pipe too: The water leaks out if you disconnect the pipe at both ends.

The `O_NONBLOCK` flag affects `read` and `write` on FIFOs just as it does on pipes. (Detailed in Section 6.2.2.)

7.2.1 Creating a FIFO

Unlike a regular file, you can't create a FIFO with `open`; you use a separate system call:

mkfifo—make FIFO

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

int mkfifo(
    const char *path,          /* pathname */
    mode_t perms               /* permissions */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

The `perms` argument is used like the third argument to `open`; that is, it is the new file's permissions.

An obvious use for a FIFO is as a replacement for a pipe. Instead of using the `pipe` system call, you make a FIFO and then open it twice, to get read and write file descriptors.[1] From then on you can treat it like a pipe. Used this way, however, FIFOs have absolutely no advantages over pipes and several disadvantages: extra overhead in making the FIFO, two system calls to get file descriptors, and the risk of a name clash just as with a temporary file.

FIFOs weren't added to UNIX to replace *pipes*—they were added to provide a simple way of passing data between server processes and clients. Recall that a limitation of pipes is that the file descriptors used to read and write them can be passed to a process only by inheritance. This won't do if the server is to be kept

1. Opening it for both reading and writing (`O_RDWR`) might work on your system, but it's nonstandard.

running while client processes come and go. Since any process with appropriate permissions can open a FIFO, it's easy to arrange for the server to create a process with a fixed name so clients can open it.

In this chapter we're going to use all of the interprocess communication mechanisms we introduce to pass messages—short pieces of structured data—between processes, rather than streams of data, as we did in Chapter 6. However, bear in mind that you can certainly use a FIFO to pass a stream of data as well.

7.2.2 A Simple FIFO Example

We'll start with two programs, a server and a client. The server runs continuously, waiting for a client to send it a message. In this example, the message is a string to be converted to upper case. The server converts it and sends it back to the client, and then it loops back for the next message, possibly from a different client. The example client sends a few strings to the server, displays the results, and then exits.

We start the server like this:

```
$ smsg_server&
server started
[1] 8725
$
```

Then we run a client like this:

```
$ smsg_client
client 8747 started
client 8747: applesauce --> APPLESAUCE
client 8747: tiger --> TIGER
client 8747: mountain --> MOUNTAIN
Client 8747 done
```

Figure 7.1 shows the arrangement of server, clients, and FIFOs. The server creates a single input FIFO, for use by all clients, named "fifo_server," and each client (two are shown) creates its own FIFO, with the process ID embedded in the name for uniqueness, to receive replies from the server. In the figure, client 8748 has sent a message (represented by a rectangle) containing the data to be converted ("tiger") and its process ID. The server reads that message from its input FIFO, converts the data (to "TIGER"), and then uses the sent process ID (8748) to determine the name of the FIFO to which to write the result message. Thus, there's only one common server FIFO but a separate reply FIFO for each client.

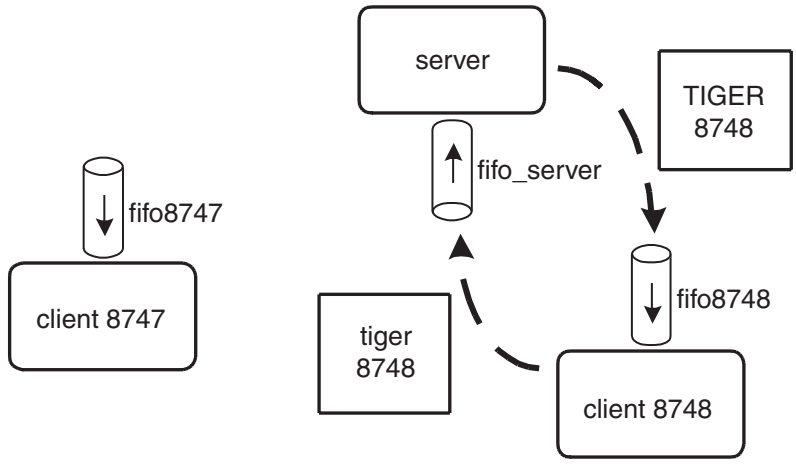

Figure 7.1 Server with two clients.

Client and server have to use the same algorithm for composing a FIFO name from a client process ID, so my example uses a function for that purpose that they share:

```
bool make_fifo_name(pid_t pid, char *name, size_t name_max)
{
    snprintf(name, name_max, "fifo%ld", (long)pid);
    return true;
}
```

A common header file defines the fixed server FIFO name and a structure for messages:

```
#define SERVER_FIFO_NAME "fifo_server"

struct simple_message {
    pid_t sm_clientpid;
    char sm_data[200];
};
```

Here's the code for the server program:

```
int main(void)
{
    int fd_server, fd_client, i;
    ssize_t nread;
    struct simple_message msg;
    char fifo_name[100];
```

```
    printf("server started\n");
    if (mkfifo(SERVER_FIFO_NAME, PERM_FILE) == -1 && errno != EEXIST)
        EC_FAIL
    ec_neg1( fd_server = open(SERVER_FIFO_NAME, O_RDONLY) )
    while (true) {
        ec_neg1( nread = read(fd_server, &msg, sizeof(msg)) )
        if (nread == 0) {
            errno = ENETDOWN;
            EC_FAIL
        }
        for (i = 0; msg.sm_data[i] != '\0'; i++)
            msg.sm_data[i] = toupper(msg.sm_data[i]);
        ec_false( make_fifo_name(msg.sm_clientpid, fifo_name,
          sizeof(fifo_name)) )
        ec_neg1( fd_client = open(fifo_name, O_WRONLY) )
        ec_neg1( write(fd_client, &msg, sizeof(msg)) )
        ec_neg1( close(fd_client) )
    }
    /* never actually get here */
    ec_neg1( close(fd_server) )
    exit(EXIT_SUCCESS);

EC_CLEANUP_BGN
    exit(EXIT_FAILURE);
EC_CLEANUP_END
}
```

Here's what the server does:

- It makes its FIFO, failing only if `mkfifo` returned an error other than the FIFO already existing, as it may be left around from a previous execution of the server.
- It opens its FIFO for reading. This will block until there is a writer, so it's OK to start the server before any client runs.
- Each time through the loop, it reads a message, converts the data, opens the client's FIFO, writes the result message to that FIFO, and then closes it.
- As we didn't provide an explicit way to stop the server, it just stays in the loop indefinitely until it's terminated with the `kill` command (not shown in our examples).

You can probably already imagine what the client program has to do. Here's the code:

```
int main(int argc, char *argv[])
{
    int fd_server, fd_client = -1, i;
    ssize_t nread;
    struct simple_message msg;
    char fifo_name[100];
    char *work[] = {
        "applesauce",
        "tiger",
        "mountain",
        NULL
    };

    printf("client %ld started\n", (long)getpid());
    msg.sm_clientpid = getpid();
    ec_false( make_fifo_name(msg.sm_clientpid, fifo_name,
      sizeof(fifo_name)) )
    if (mkfifo(fifo_name, PERM_FILE) == -1 && errno != EEXIST)
        EC_FAIL
    ec_neg1( fd_server = open(SERVER_FIFO_NAME, O_WRONLY) )
    for (i = 0; work[i] != NULL; i++) {
        strcpy(msg.sm_data, work[i]);
        ec_neg1( write(fd_server, &msg, sizeof(msg)) )
        if (fd_client == -1)
            ec_neg1( fd_client = open(fifo_name, O_RDONLY) )
        ec_neg1( nread = read(fd_client, &msg, sizeof(msg)) )
        if (nread == 0) {
            errno = ENETDOWN;
            EC_FAIL
        }
        printf("client %ld: %s --> %s\n", (long)getpid(),
          work[i], msg.sm_data);
    }
    ec_neg1( close(fd_server) )
    ec_neg1( close(fd_client) )
    ec_neg1( unlink(fifo_name) )
    printf("Client %ld done\n", (long)getpid());
    exit(EXIT_SUCCESS);

EC_CLEANUP_BGN
    exit(EXIT_FAILURE);
EC_CLEANUP_END
}
```

The client does this:

- Makes its FIFO.
- Opens the server FIFO for writing. This will block until the server opens it for reading, so it's OK to start a client before the server is started.

- For each string to be converted, it forms the message, writes it to the server, opens its own client for reading, if necessary, and reads the result.
- After all the strings have been converted, it closes the file descriptors it's opened and unlinks its FIFO since it's unlikely that any other client will use it.

There are two items of bad news to give you: First, client and server both have bugs, of a similar nature, and, second, I lied when I showed the output from the client. In fact, the client's output was this:

```
$ smsg_client
client 8747 started
client 8747: applesauce --> APPLESAUCE
ERROR:  0: main [/aup/c7/smsg_client.c:46] 0
                *** ? (100: "Network is down") ***
$
```

What happened? What's the bug? Well, the location and the `errno` value tell us that the client got an end-of-file from its own FIFO. Why wasn't there data since the client had just sent a message to the server? The problem is one of timing: The server closes its client-FIFO file descriptor after each message and didn't get around to re-opening it before the client went from its `write` to its `read`. So, since there were no file descriptors open for writing, the client got an end-of-file, as explained in Section 6.2.2.

The fix is very easy. If the client maintains a file descriptor open for writing to its own FIFO, even though it will never write, that will force any `read` of its FIFO to block until some data is written there (by the server), which is the behavior we want. So all we have to do to fix the client is to declare another file-descriptor variable like this:

```
int fd_client_w = -1;
```

and then open it along with the reading file descriptor, like this:

```
        if (fd_client == -1)
            ec_neg1( fd_client = open(fifo_name, O_RDONLY) )
        if (fd_client_w == -1)
            ec_neg1( fd_client_w = open(fifo_name, O_WRONLY) )
```

(We should also close it along with `fd_client`.) On most UNIX systems, you can also fix the client with just one file descriptor opened for both reading and writing, but that's nonstandard.

So, rerunning the client with this fix, we now get this output:

```
client 8937 started
client 8937: applesauce --> APPLESAUCE
client 8937: tiger --> TIGER
client 8937: mountain --> MOUNTAIN
Client 8937 done
ERROR:  0: main [/aup/c7/smsg_server.c:33] 0
               *** ? (100: "Network is down") ***
```

Still not right! Now the *server* is complaining that it got an end-of-file from *its* FIFO. The reason is that the client closed the writing end of the server's FIFO when it had no more work, and, as there weren't any other clients running, the server got an end-of-file. We don't want that—we want the server to keep running, ideally blocking in its `read` until there's a message. The fix is similar to the fix for the client—the server maintains a file descriptor open for writing its own FIFO, to avoid an end-of-file. (I won't show the code—you can easily figure it out.)

With the server fixed, not only do both programs run smoothly, but we can even run multiple clients:

```
$ smsg_client & smsg_client & smsg_client & smsg_client
client 9001 started
client 9001: applesauce --> APPLESAUCE
client 9001: tiger --> TIGER
[2] 9001
client 9002 started
client 9002: applesauce --> APPLESAUCE
client 9002: tiger --> TIGER
[3] 9002
[4] 9003
client 9004 started
client 9004: applesauce --> APPLESAUCE
client 9004: tiger --> TIGER
client 9003 started
client 9003: applesauce --> APPLESAUCE
client 9003: tiger --> TIGER
client 9002: mountain --> MOUNTAIN
client 9001: mountain --> MOUNTAIN
client 9004: mountain --> MOUNTAIN
client 9003: mountain --> MOUNTAIN
Client 9001 done
Client 9002 done
Client 9004 done
[2]   Done                    smsg_client
[3]-  Done                    smsg_client
$ Client 9003 done
[4]+  Done                    smsg_client
```

7.2.3 FIFOs Critiqued

The good things about FIFOs are:

- Easy to understand and use because they're really pipes, and you use the basic I/O systems calls (`open`, `read`, `write`, etc.).
- Available in all versions of UNIX.
- Fairly efficient (see Table 7.2 at the end of this chapter).
- Work well for streams of data and for discrete messages. To speed things up, a reader can read multiple messages in one gulp, and a writer can write a whole buffer of messages at once, as long as the maximum for an atomic write isn't exceeded.

The disadvantages are:

- A single FIFO can't have multiple readers because there are no atomicity guarantees when reading (see Section 6.2.2 for more about this problem).
- Data has to be copied from user space in one process to a kernel buffer and then back to another user space, which is expensive. (Message queues and sockets have the same disadvantage.) Thus, FIFOs aren't suitable for the most critical applications.
- If the message size is too big (see Section 6.2.2), a writer may block. If this isn't carefully handled, the application may deadlock.

7.3 An Abstract Simple Messaging Interface (SMI)

It's useful to generalize the messaging sending and receiving from the FIFO example in the previous section into an abstract interface, for two key reasons:

- Critical details like keeping a file descriptor open for writing can be handled by the implementation of the interface, allowing for easier and more reliable application programming.
- Many different implementations can be programmed for different IPC mechanisms. You can experiment with various implementations without changing the application program's source to find the one that works best.

I call the interface I'm going to use SMI, for Simple Messaging Interface.[2]

2. It was designed for this book; it's not part of any standard.

7.3.1 SMI Types and Functions

I'm just going to explain the interface in this section; the specific implementations come later. First, here's the generalized message structure:

struct smi_msg—structure for SMI

```
struct smi_msg {
     long smi_mtype; /* must be first */
     struct client_id {
         long c_id1;
         long c_id2;
     } smi_client;
     char smi_data[1];
};
```

The first two members, `smi_mtype` and `smi_client`, are required by some of the implementations, as we shall see. We've already seen, in the example in Section 7.2.2, that a client has to pass its process ID to the server; in an SMI message, this goes into the `client.cl_id1` member. We'll see `smi_mtype` and `smi_client.c_id2` used later on.

The message data (`smi_data`) can be anything at all, with these restrictions:

- Server and client may be on different machines, so no pointers or other data that isn't meaningful across a network should be passed.
- As different machines have different byte ordering, binary numbers need to be expressed in a network-standard format. This topic is explored in Section 8.1.4.

In this simple interface, server and client have to agree on the fixed message size, and the actual size of the `smi_data` array depends on that size. Once you've mastered the principles in this and the next chapter, you can investigate relaxing that limitation in your own version of the SMI that you use in your own applications.

Before sending or receiving, a server or client has to open a message queue, which should be closed upon termination. Conceptually, sent messages are put on the queue, and received messages are taken from the queue. What a queue actually is is implementation dependent and is hidden inside the SMI implementation, much as the internals of a `FILE` type are hidden by the Standard C I/O functions.

The opening and closing interfaces are:

SMI types—types for SMI

```
typedef void *SMIQ;             /* message queue */
typedef enum {
     SMI_SERVER,                /* server */
     SMI_CLIENT                 /* client */
} SMIENTITY;
#define SERVER_NAME_MAX 50      /* max size of server name */
```

smi_open—open SMI message queue

```
SMIQ *smi_open(
     const char *name,          /* server name */
     SMIENTITY entity,          /* entity being opened */
     size_t msgsize             /* fixed message size */
);
/* Returns pointer to message queue or NULL on error (sets errno) */
```

smi_close—close SMI message queue

```
bool smi_close(
     SMIQ *sqp                  /* queue */
);
/* Returns true on success or false on error (sets errno) */
```

The message-queue name passed to `smi_open` must be known to the server and all clients. If the implementation needs to create it before opening it, this is done behind the scenes inside `smi_open`. There aren't any permissions in this simple interface, which is one of the many reasons why it's called "simple." Whether the message queue is left around after it's closed isn't specified as part of the interface and is also up to the implementation.

The `entity` argument is either `SMI_SERVER` or `SMI_CLIENT`, depending on which kind of program is doing the opening. There can be only one server process but an undefined number of clients.

The `msgsize` argument is the size of the `smi_data` member of the `smi_msg` structure, for all messages. As I mentioned, there's no provision for messages being of different sizes.

For sending and receiving, we could just pass an address of a buffer to simple send and receive functions, something like `write` and `read`, like this (no size argument because it's a fixed number):

```
ec_false( smi_send(sqp, buffer) )
...
ec_false( smi_receive(sqp, buffer) )
```

But if we did that it would require that all implementations copy each message from the sending process to the kernel and again from the kernel to the receiving process. Instead, we'll use a slightly more elaborate interface that uses two calls for sending and two for receiving. The first call of each pair just gets the address of the message, and the second releases access to that address. That way the implementation can keep it in the `SMIQ` structure (implying that it is indeed copied), or keep it in shared memory, or whatever. All the callers know is that they have an address that they can dereference. The release functions are needed because the address may not be good forever—the memory may have to be deallocated, or the shared-memory segment may have to be reused—and because the underlying implementation needs to know when the sender is ready for the message to go.

It's easier to explain receiving before sending:

smi_receive_getaddr—get received SMI message address

```
bool smi_receive_getaddr(
     SMIQ *sqp,              /* queue */
     void **addr             /* message */
);
/* Returns true on success or false on error (sets errno) */
```

smi_receive_release—release received SMI message

```
bool smi_receive_release(
     SMIQ *sqp               /* queue */
);
/* Returns true on success or false on error (sets errno) */
```

The `addr` argument is a pointer to a `void` pointer rather than to a `struct smi_msg` pointer because in practice each application will define its own message structure whose first two members match the first two members of `smi_msg` (`smi_mtype` and `smi_client`).

An application uses these two calls like this:

```
struct my_msg *msg;
...
ec_false( smi_receive_getaddr(sqp, (void **)&msg) )
/* process data in msg->smi_data */
ec_false( smi_receive_release(sqp) )
```

Conceptually, the message is actually received when `smi_receive_getaddr` is called. Note that the application doesn't allocate space for the message; that's done by the implementation.

Sending is very similar, with one added wrinkle—identifying the client:

smi_send_getaddr—get sending SMI message address

```
bool smi_send_getaddr(
     SMIQ *sqp,                 /* queue */
     struct client_id *client,  /* client ID (server only)
     void **addr                /* message */
);
/* Returns true on success or false on error (sets errno) */
```

smi_send_release—release and send SMI message

```
bool smi_send_release (
     SMIQ *sqp                  /* queue */
);
/* Returns true on success or false on error (sets errno) */
```

Conceptually, the actual send occurs when `smi_send_release` is called.

The second argument to `smi_send_getaddr` is used by a server to identify the client it's sending to. It's a pointer to the `struct client_id` that was in a received message. This works because a server receives a message from a client that wants to do business with it before it sends a message to that client. A client that sends doesn't need to identify the server (that was done with `smi_open`), so the second argument is `NULL`.

A server, then, would do something like this:

```
struct my_msg *msg_in, *msg_out;
...
ec_false( smi_receive_getaddr(sqp, (void **)&msg_in) )
/* code to process data in msg_in->smi_data */
ec_false( smi_send_getaddr(sqp, &msg_in->smi_client, (void **)&msg_out) )
ec_false( smi_receive_release(sqp) )
/* code to put data into msg_out->smi_data */
ec_false( smi_send_release(sqp) )
```

Observe that:

- The server used two separate `my_msg` addresses (`msg_in` and `msg_out`) because two different buffers are involved.
- `msg_in` wasn't released until after the call to `smi_send_getaddr` because the `msg_in->smi_client` structure was needed until then. It's OK to interleave the calls in this way; in fact, `smi_receive_release` could have been called even after the call to `smi_send_release`—sending and receiving are totally independent. Alternatively, the `client_id` structure could

have been copied to a temporary variable, the address of which could be used in the call to `smi_send_getaddr`.

A client calls `smi_send_getaddr` and `smi_send_release` like this (there's probably a `msg_in`, but we're not showing it):

```
struct my_msg *msg_out;
...
ec_false( smi_send_getaddr(sqp, NULL, (void **)&msg_out) )
/* code to put data into msg_out->smi_data */
ec_false( smi_send_release(sqp) )
```

The next section has a more complete example.

7.3.2 Example Server and Client Using SMI

To show how the SMI functions are used in an application, here is the server from the example in Section 7.2.2 rewritten to use that interface (the `defines` are in a header included by server and client):

```
#define SERVER_NAME "smsg_server"
#define DATA_SIZE 200

int main(void)
{
    SMIQ *sqp;
    struct smi_msg *msg_in, *msg_out;
    int i;

    printf("server started\n");
    ec_null( sqp = smi_open(SERVER_NAME, SMI_SERVER, DATA_SIZE) )
    while (true) {
        ec_false( smi_receive_getaddr(sqp, (void **)&msg_in) )
        ec_false( smi_send_getaddr(sqp, &msg_in->smi_client,
          (void **)&msg_out) )
        for (i = 0; msg_in->smi_data[i] != '\0'; i++)
            msg_out->smi_data[i] = toupper(msg_in->smi_data[i]);
        msg_out->smi_data[i] = '\0';
        ec_false( smi_receive_release(sqp) )
        ec_false( smi_send_release(sqp) )
    }
    /* never actually get here */
    ec_false( smi_close(sqp) )
    exit(EXIT_SUCCESS);
```

```
EC_CLEANUP_BGN
    exit(EXIT_FAILURE);
EC_CLEANUP_END
}
```

and here's the client:

```
int main(int argc, char *argv[])
{
    SMIQ *sqp;
    struct smi_msg *msg;
    int i;

    char *work[] = {
        "applesauce",
        "tiger",
        "mountain",
        NULL
    };

    printf("client %ld started\n", (long)getpid());
    ec_null( sqp = smi_open(SERVER_NAME, SMI_CLIENT, DATA_SIZE) )
    for (i = 0; work[i] != NULL; i++) {
        ec_false( smi_send_getaddr(sqp, NULL, (void **)&msg) )
        strcpy(msg->smi_data, work[i]);
        ec_false( smi_send_release(sqp) )
        ec_false( smi_receive_getaddr(sqp, (void **)&msg) )
        printf("client %ld: %s --> %s\n", (long)getpid(),
          work[i], msg->smi_data);
        ec_false( smi_receive_release(sqp) )
    }
    ec_false( smi_close(sqp) )
    printf("Client %ld done\n", (long)getpid());
    exit(EXIT_SUCCESS);

EC_CLEANUP_BGN
    exit(EXIT_FAILURE);
EC_CLEANUP_END
}
```

Notice that all the monkey-business with keeping file descriptors opened for writing that we saw in Section 7.2.2 is gone, as are details like FIFO names. Maybe "hidden" would be more accurate than "gone," as we are about to discover.

7.3.3 FIFO Implementation of SMI

All of the complexity of messaging with FIFOs that was removed from the server and client now has to go into the implementation of the SMI functions. In addition, the implementation of the server will try to keep client-FIFO file descriptors

open across messages, to save the overhead of opening and closing them each time, as the example in Section 7.2.2 did.

The FIFO implementation uses function names of the form *smifcn*_fifo, where *smifcn* is the SMI name (e.g., smi_send_getaddr_fifo is the implementation of smi_send_getaddr). A small wrapper file in effect translates the names, with little functions like this:

```
bool smi_send_getaddr(SMIQ *sqp, struct client_id *client, void **addr)
{
    return smi_send_getaddr_fifo(sqp, client, addr);
}
```

Using the wrappers allows two kinds of applications to be linked:

- An implementation-independent application, like the one shown in the previous section, can be written in terms of the generic SMI functions (e.g., smi_send_getaddr) and then linked with a particular implementation and the appropriate wrapper to translate the calls from generic to specific.
- An implementation that wants to use several implementations can use the specific function names (e.g., smi_send_getaddr_fifo, smi_send_getaddr_skt), bypassing the wrappers, which aren't linked in. The main purpose of this is for writing a test program that, for example, runs the same application using different methods to compare their performance. I did exactly that to prepare Table 7.2, which closes this chapter.

I won't show the wrappers in this book, but they're on the Web site [AUP2003].

We'll begin the FIFO implementation with the message queue that it uses internally. Server and client use the same data structure, although each has its own data since they are in separate processes.

```
#define MAX_CLIENTS 20

typedef struct {
    SMIENTITY sq_entity;            /* entity */
    int sq_fd_server;               /* server read and ... */
    int sq_fd_server_w;             /* ... write file descriptors */
    char sq_name[SERVER_NAME_MAX];  /* server name */
    struct {
        int cl_fd;                  /* client file descriptor */
        pid_t cl_pid;               /* client process ID */
    } sq_clients[MAX_CLIENTS];
    struct client_id sq_client;     /* client ID */
    size_t sq_msgsize;              /* msg size */
    struct smi_msg *sq_msg;         /* msg buffer */
} SMIQ_FIFO;
```

There's room for only 20 clients per server. A linked list could be used to remove the limit, but I didn't bother. Both client and server use `sq_fd_server` for the server, and in addition the server uses `sq_fd_server_w` to keep a file descriptor open for writing, as explained in Section 7.2.2. The server uses the `sq_clients` array to hold the client file descriptors to avoid opening and closing them with each incoming message. A client needs just one of them for reading its own FIFO, for which it uses `sq_clients[0]`. It uses `sq_clients[1]` to keep a second file descriptor open for writing. For a client, the rest of the array and `sq_fd_server_w` are wasted space, but I thought it would be easier to use the same queue structure for both, as the wasted space is small.

The client-ID information passed by the server to `smi_send_getaddr_fifo` is kept for use by the following `smi_send_release_fifo` in the `sq_client` member. The size passed to `smi_open_fifo` and a pointer to a buffer for messages are in the `sq_msgsize` and `sq_msg` members. (Note that with FIFOs, as with message queues and sockets, a copy from user space to kernel and back into the receiving process is unavoidable.)

Server and client need to initialize the sq_clients array, with this internal function:

```
static void clients_bgn(SMIQ_FIFO *p)
{
    int i;

    for (i = 0; i < MAX_CLIENTS; i++)
        p->sq_clients[i].cl_fd = -1;
}
```

And, at the end, they call `clients_end`:

```
static void clients_end(SMIQ_FIFO *p)
{
    clients_close_all(p);
}

static void clients_close_all(SMIQ_FIFO *p)
{
    int i;

    for (i = 0; i < MAX_CLIENTS; i++)
        if (p->sq_clients[i].cl_fd != -1) {
            (void)close(p->sq_clients[i].cl_fd);
            p->sq_clients[i].cl_fd = -1;
        }
}
```

We'll see these calls in the `smi_open_fifo` and `smi_close_fifo` functions.

When the server gets a message, it uses `clients_find` to see if it already has a file descriptor open to the FIFO, and to find an available slot if not:

```
static int clients_find(SMIQ_FIFO *p, pid_t pid)
{
    int i, avail = -1;

    for (i = 0; i < MAX_CLIENTS; i++) {
        if (p->sq_clients[i].cl_pid == pid)
            return i;
        if (p->sq_clients[i].cl_fd == -1 && avail == -1)
            avail = i;
    }
    if (avail != -1)
        p->sq_clients[avail].cl_pid = pid;
    return avail;
}
```

This function returns –1 if no more clients can be handled.

The final internal functions are for making the FIFO names. The server's is fixed, while the clients' contain their process IDs, just as in the earlier example in Section 7.2.2.

```
static void make_fifo_name_server(const SMIQ_FIFO *p, char *fifoname,
  size_t fifoname_max)
{
    snprintf(fifoname, fifoname_max, "/tmp/smififo-%s", p->sq_name);
}

static void make_fifo_name_client(pid_t pid, char *fifoname,
  size_t fifoname_max)
{
    snprintf(fifoname, fifoname_max, "/tmp/smififo%ld", (long)pid);
}
```

Now we're ready to look at `smi_open_fifo`:

```
SMIQ *smi_open_fifo(const char *name, SMIENTITY entity, size_t msgsize)
{
    SMIQ_FIFO *p = NULL;
    char fifoname[SERVER_NAME_MAX + 50];

    ec_null( p = calloc(1, sizeof(SMIQ_FIFO)) )
    p->sq_msgsize = msgsize + offsetof(struct smi_msg, smi_data);
    ec_null( p->sq_msg = calloc(1, p->sq_msgsize) )
    p->sq_entity = entity;
```

```
    if (strlen(name) >= SERVER_NAME_MAX) {
        errno = ENAMETOOLONG;
        EC_FAIL
    }
    strcpy(p->sq_name, name);
    make_fifo_name_server(p, fifoname, sizeof(fifoname));
    if (p->sq_entity == SMI_SERVER) {
        clients_bgn(p);
        if (mkfifo(fifoname, PERM_FILE) == -1 && errno != EEXIST)
            EC_FAIL
        ec_neg1( p->sq_fd_server = open(fifoname, O_RDONLY) )
        ec_neg1( p->sq_fd_server_w = open(fifoname, O_WRONLY) )
    }
    else {
        ec_neg1( p->sq_fd_server = open(fifoname, O_WRONLY) )
        make_fifo_name_client(getpid(), fifoname, sizeof(fifoname));
        (void)unlink(fifoname);
        ec_neg1( mkfifo(fifoname, PERM_FILE) )
        ec_neg1( p->sq_clients[0].cl_fd =
          open(fifoname, O_RDONLY | O_NONBLOCK) )
        ec_false( setblock(p->sq_clients[0].cl_fd, true) )
        ec_neg1( p->sq_clients[1].cl_fd = open(fifoname, O_WRONLY) )
    }
    return (SMIQ *)p;

EC_CLEANUP_BGN
    if (p != NULL) {
        free(p->sq_msg);
        free(p);
    }
    return NULL;
EC_CLEANUP_END
}
```

This code should be understandable, especially if you've understood the example in Section 7.2.2. But the parts that actually open the FIFOs are worth reviewing. For the server, the sequence for opening its FIFO is:

```
ec_neg1( p->sq_fd_server = open(fifoname, O_RDONLY) )
ec_neg1( p->sq_fd_server_w = open(fifoname, O_WRONLY) )
```

The first call will block if there is no writer, which is OK because without a writer the server doesn't have anything to do anyway. However, the sequence for a client opening its FIFO is more complicated:

```
ec_neg1( p->sq_clients[0].cl_fd =
  open(fifoname, O_RDONLY | O_NONBLOCK) )
ec_false( setblock(p->sq_clients[0].cl_fd, true) )
ec_neg1( p->sq_clients[1].cl_fd = open(fifoname, O_WRONLY) )
```

Without the `O_NONBLOCK` flag, a client would block in its `open` waiting for the server to open the FIFO for writing. But, the server won't do this until it gets a message, for only then does it even know that this particular client exists. Deadlock! The solution, as shown in the code, is to open the FIFO nonblocking, which means it returns with a valid file descriptor without waiting for a writer. But then we have to clear the `O_NONBLOCK` flag (i.e., set blocking on) so that subsequent `read`s will block, which is the behavior we want. (`setblock`, which uses the `fcntl` system call, came from Section 4.2.2.) In the example in Section 7.2.2, we didn't have to do this because the client didn't open its own FIFO until it had already sent the message to the server. That doesn't work here because the SMI functions are separate, with specific jobs to do. (Sometimes modularization requires extra work.)

Here's the companion `smi_close_fifo` function:

```
bool smi_close_fifo(SMIQ *sqp)
{
    SMIQ_FIFO *p = (SMIQ_FIFO *)sqp;

    clients_end(p);
    (void)close(p->sq_fd_server);
    if (p->sq_entity == SMI_CLIENT) {
        char fifoname[SERVER_NAME_MAX + 50];

        make_fifo_name_client(getpid(), fifoname, sizeof(fifoname));
        (void)unlink(fifoname);
    }
    else
        (void)close(p->sq_fd_server_w);
    free(p->sq_msg);
    free(p);
    return true;
}
```

Note that clients unlink their FIFOs, but the server leaves its around so that in the future a client can be started without the server running.

Here's `smi_send_getaddr_fifo`, which doesn't do much other than save the client ID and return the buffer address:

```
bool smi_send_getaddr_fifo(SMIQ *sqp, struct client_id *client,
  void **addr)
{
    SMIQ_FIFO *p = (SMIQ_FIFO *)sqp;
```

```
    if (p->sq_entity == SMI_SERVER)
        p->sq_client = *client;
    *addr = p->sq_msg;
    return true;
}
```

The real work is in `smi_send_release_fifo`:

```
bool smi_send_release_fifo(SMIQ *sqp)
{
    SMIQ_FIFO *p = (SMIQ_FIFO *)sqp;
    ssize_t nwrite;

    if (p->sq_entity == SMI_SERVER) {
        int nclient = clients_find(p, p->sq_client.c_id1);
        if (nclient == -1 || p->sq_clients[nclient].cl_fd == -1) {
            errno = EADDRNOTAVAIL;
            EC_FAIL
        }
        ec_neg1( nwrite = write(p->sq_clients[nclient].cl_fd, p->sq_msg,
          p->sq_msgsize) )
    }
    else {
        p->sq_msg->smi_client.c_id1 = (long)getpid();
        ec_neg1( nwrite = write(p->sq_fd_server, p->sq_msg,
          p->sq_msgsize) )
    }
    return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}
```

There's nothing unusual here that didn't appear in the earlier example in Section 7.2.2.

`smi_receive_getaddr_fifo` does all the work of receiving the message:

```
bool smi_receive_getaddr_fifo(SMIQ *sqp, void **addr)
{
    SMIQ_FIFO *p = (SMIQ_FIFO *)sqp;
    ssize_t nread;

    if (p->sq_entity == SMI_SERVER) {
        int nclient;
        char fifoname[SERVER_NAME_MAX + 50];

        while (true) {
            ec_neg1( nread = read(p->sq_fd_server, p->sq_msg,
              p->sq_msgsize) )
```

```
            if (nread == 0) {
                errno = ENETDOWN;
                EC_FAIL
            }
            if (nread < offsetof(struct smi_msg, smi_data)) {
                errno = E2BIG;
                EC_FAIL
            }
            if ((nclient = clients_find(p,
              (pid_t)p->sq_msg->smi_client.c_id1)) == -1) {
                continue; /* client not notified */
            }
            if (p->sq_clients[nclient].cl_fd == -1) {
                make_fifo_name_client((pid_t)p->sq_msg->smi_client.c_id1,
                  fifoname, sizeof(fifoname));
                ec_neg1( p->sq_clients[nclient].cl_fd =
                  open(fifoname, O_WRONLY) )
            }
            break;
        }
    }
    else
        ec_neg1( nread = read(p->sq_clients[0].cl_fd, p->sq_msg,
          p->sq_msgsize) )
    *addr = p->sq_msg;
    return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}
```

The one sticky part here is:

```
if ((nclient = clients_find(p, (pid_t)mp->smi_client)) == -1) {
    continue; /* client not notified */
}
```

If `clients_find` returns −1, it means that the server got a message from a client but can't get a slot in the `sq_clients` array so it can't open the response FIFO. Since it can't respond, it can't even send an error message. The example code just ignores the error, leaving the client blocked. Some alternatives would be:

- Send a signal to the client, and modify `smi_open_fifo` to arrange to catch it.
- Use an integer variable to hold an emergency file descriptor to open the FIFO so an error message can be sent.
- At least change the client so it times out rather than staying blocked forever.

There's one function left, but it has nothing at all to do:

```
bool smi_receive_release_fifo(SMIQ *sqp)
{
    return true;
}
```

That's the whole implementation. Even with the limitations I've discussed, it's still a highly portable, reasonably reliable implementation of the SMI functions that you can start using right away in your own application, at least for prototyping. Later, you can use one of the other implementations since the interface is identical.

7.4 System V IPC (Interprocess Communication)

As explained in Section 1.1.7, there are two sets of system calls for messages, semaphores, and shared memory. The older set is usually called System V IPC, and the newer set is called POSIX IPC. I'll discuss both in this chapter, first System V and POSIX messages, then both types of semaphores, and finally shared memory. This section explains some general concepts that apply to all of the System V IPC mechanisms; later, in Section 7.6, general POSIX IPC concepts will be explained.

7.4.1 System V IPC Objects

There are three kinds of System V IPC Objects: message queues, semaphore sets, and shared memory segments. They are not files—not even special files—but objects with their own naming scheme, own lifetime rules, and own permission system. Here are the principal characteristics of System V IPC objects:

- They exist only within a single machine. They can't be used for communication across a network.
- Their lifetime is the same as the kernel—they are destroyed on a reboot.
- You access an object with an integer *identifier* that's fixed for the lifetime of the object. Any process that knows the identifier can use it directly to access the object—there's no need to open the object first. This is different from a file descriptor, which is a property of the process and goes away when the process does. (System V IPC also has *keys,* which I'll explain in the next section.)

- There are no i-nodes or pathnames, so none of the traditional file and directory manipulation system calls, such as `unlink`, `stat`, `read`, or `write`, can be used.

All three kinds of objects have system calls of the form *X*`get` and *X*`ctl`, where *X* is `msg`, `sem`, or `shm`. Thus, the six calls are `msgget`, `semget`, `shmget`, `msgctl`, `semctl`, and `shmctl`. I'll continue to use the terms *X*`get` and *X*`ctl` to refer to "get" and "ctl" calls when what I'm discussing applies to all three objects.

There are only five more calls: `msgsnd` and `msgrcv` for sending and receiving messages, `semop` for manipulating semaphore sets, and `shmat` and `smdt` for attaching and detaching shared memory segments.

7.4.2 Identifiers, Keys, and the ftok System Call

An identifier is assigned by the kernel when a System V IPC object is created, so in general it has a different value from reboot to reboot. To make it easier for different processes to get the identifier from an object they need to share, there are also permanent *keys* that never change in value. A process can specify a key when it creates an object with *X*`get` or just use *X*`get` to get the identifier from the key if the object already exists. However, the key doesn't really identify the specific object, in the sense that a pathname identifies a file, because the object lasts only until the next reboot. Next time, the same key is used to generate a new object with a possibly different identifier.

Keys are of type `key_t`, which is not required to even be an arithmetic type, although it is used that way by various system calls, so it's a safe assumption. If you wanted to, you could simply define key values for programs to use, like this:

```
#define MSGQ_KEY   1234
#define SEM_KEY    1235
#define SHM_KEY1   1236
#define SHM_KEY2   1237
```

Then any process that calls `msgget` with the key `MSGQ_KEY` as an argument will get an identifier to the same message queue.

The problem with keys is that some world-wide administration scheme is needed to assign them so there are conflicts between applications. This is clearly unworkable, so another layer of abstraction allows a key to be generated from a pathname with the `ftok` system call.

ftok—generate System V IPC key

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

key_t ftok(
    const char *path,          /* pathname of existing file */
    int id                     /* desired key number */
);
/* Returns key on success or -1 on error (sets errno) */
```

Actually, `ftok` can generate lots of keys from the same path; the desired key is specified by the `id` argument, which could be a character, as only its lowest 8 bits are used. It can't be zero, either. For example, if you need four keys for your application (one message queue, one semaphore set, and two shared memory segments), you can use a single path name (e.g., /tmp/myappkeys) and call `ftok` four times with the second argument taking on the values `q`, `s`, `m`, and `n` (or whatever four values you like). The pathname must already exist—`ftok` will not create it.

Recall that I said that System V IPC objects are not files and do not have i-nodes. The file whose name is passed to `ftok` exists only to provide a global name for key generation. The contents of the file don't matter. It's not guaranteed that if the file is unlinked and re-created the same keys will be generated even though the name stays the same, so you should create the file when the application is installed or run for the first time and leave it around until the application is uninstalled.

Two different paths are guaranteed to generate two different keys only if the paths are in the same file system (Section 3.2.4). If you're worried about different applications accidentally using the same key, you can use the special key `IPC_PRIVATE` in an *X*`get` call (explained further in Section 7.5.1), which guarantees you a unique IPC object without using `ftok` or specific keys at all. You have to somehow publish the resulting identifier so all the processes that need it can get it; one way to do that is to write it to a file whose name is known to those processes. But, for most purposes, including all the examples in this book, we're going to assume that `ftok` conflicts won't occur.

To summarize:

- You access a System V IPC object with an identifier. How you get the identifier doesn't matter—it can be an argument to an `exec`, passed with a message, read from a file, returned from a system call like `msgget`, or whatever.

- If you want (and you usually do), you can refer to an object with a key which, unlike an identifier, remains constant even as the object is destroyed and re-created.
- Because it's hard to manage keys, they can be generated from pathnames with `ftok`, and in practice, this is usually what you'll do. Nonunique keys are a remote possibility; creating `IPC_PRIVATE` objects is a potential solution.

You may be asking yourself, "Why have all this identifier/key/`ftok` stuff when UNIX has always had a perfectly good mechanism involving path name and file descriptors already?" The answer will be clear a bit later when we look at some example code that uses System V IPC messages. We'll see that a server can take an identifier for a client's message queue and immediately use it to send a message back, with no overhead to find and open an i-node, which is what `open` does.

Still, one wonders if a cleaner way could have been designed, one that's efficient but still more tightly integrated into the rest of the file system. We'll see such a way when we look at POSIX IPC messages, although it has its own problems with the way it uses pathnames.

Aside from the complexity and irregularity, another disadvantage of the System V IPC approach is that since identifiers aren't file descriptors, you can't use `select` or `poll` (Section 4.2) to block, for example, until a message is ready. I've already shown one solution to this problem in Section 5.18.

In any case, the System V IPC system calls are now standardized and there's no chance that any improvements will ever be made.

7.4.3 System V IPC Ownership and Permissions

If only System V IPC objects were files, they could use the permission system that files use, and I wouldn't have to explain anything here. But, as they are not files, they have their own system. Fortunately, it's a lot like the one that files use.

Permissions are specified using the usual 9 bits: read, write, and execute for owner, group, and others. However, "execute" doesn't mean anything, so those bits aren't used. The permissions for an object are set when it's created with one of the *X*`get` system calls. Later, you can change the permissions with an *X*`ctl` system call.

Files have an owner user-ID and owner group-ID. System V IPC objects have those two and, in addition, store the user-ID and group-ID of the creator. You can use *X*`ctl` to change the owner IDs but not the creator IDs.

When you manipulate an object, the algorithm for checking the permissions uses the effective user-ID and effective group-ID just as for files, and the "other" permission bits are used only when there's no match for user- or group-IDs. However, the effective user- or group-IDs can match either the creator or owner IDs.

Having separate creator and owner IDs allows an administrative user, say, "dbmsadm," to be the creator of a message queue and also the effective user ID of a database server process that has access to database files. A lesser user, "dbmsuser," say, can access the message queue but not the files.

When you're getting or setting permissions with one of the *X*`ctl` system calls, you use this structure:

struct ipc_perm—structure for System V IPC permissions

```
struct ipc_perm {
     uid_t uid;             /* owner user-ID */
     gid_t gid;             /* owner group-ID */
     uid_t cuid;            /* creator user-ID */
     gid_t cgid;            /* creator group-ID */
     mode_t mode;           /* permission bits */
};
```

7.4.4 System V IPC Utilities

Since System V IPC objects are not files and don't have i-nodes, you can't use system calls like `unlink` or `stat` on them, and, therefore, you can't remove them with `rm` and you won't see them listed with the `ls` command. Instead, there are two commands specifically for manipulating System V IPC objects: `ipcrm`, which removes an object, and `ipcs`, which reports its status. Check [SUS2002] or your system's documentation to see all the various options for these commands; I'll just sketch the basics here.

You call `ipcrm` with one or more argument pairs of the form

-X Y

where *X* is a lower or upper case `Q` for a message queue, `S` for a semaphore set, or `M` for a shared memory segment. If lower case, *Y* is an identifier; if upper case, *Y* is a key.[3] For example, the command

```
ipcrm -q 50
```

removes the message queue whose identifier is 50. When you remove a file, all you're really doing is removing an entry from a directory; the i-node stays around until the last file descriptor open to it is closed, so running processes aren't unduly affected. No such behavior is specified for System V IPC objects, however—the object may go away immediately, causing havoc with running processes that are using it. So, generally, `ipcrm` is used only as a part of off-hours system administration or when you know that the object isn't in use.

The System V IPC equivalent to `ls` is `ipcs`. Used without arguments it displays something like this:

```
IPC status from <running system> as of Wed Mar 12 15:04:06 MST 2003
T         ID      KEY        MODE        OWNER        GROUP
Message Queues:
q         50   0x1007b8c  --rw-rw-rw-     marc sysadmin
Shared Memory:
m          0   0x500004b7 --rw-r--r--     root        root
m        102   0          --rw-rw-rw-     marc sysadmin
m        103   0          --rw-rw-rw-     marc sysadmin
m        104   0          --rw-rw-rw-     marc sysadmin
m        105   0          --rw-rw-rw-     marc sysadmin
Semaphores:
s     131072   0x1007b8d  --ra-ra-ra-     marc sysadmin
s     131073   0          --ra-ra-ra-     marc sysadmin
```

Note that some of the keys are zero. These are private objects, created by *X*`get` with a key argument of `IPC_PRIVATE`, rather than some key. This is done when the identifier will be passed to other processes directly (e.g., as data in a message) and there's no need for the other process to execute its own *X*`get`. We'll see this in the System V message queue implementation of the SMI functions (Section 7.5.3). (`IPC_PRIVATE` can also be used when you want to ensure that the object is unique, as I explained in Section 7.4.2.)

3. That's the standard, anyway. Linux (or maybe it's GNU) does it differently; see your system documentation for details.

7.5 System V Message Queues

Now we're ready for the details of the System V message-queue system calls.

7.5.1 System V Message-Queue System Calls

We already sketched what `msgget` does; here are the particulars:

msgget—get message-queue identifier

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

int msgget(
    key_t key,              /* key */
    int flags               /* creation flags */
);
/* Returns identifier or -1 on error (sets errno) */
```

As I said in Section 7.4, where we discussed System V IPC objects in general, `msgget` gets the identifier for the existing message queue whose key is given by its first argument. If the `flags` argument has the `IPC_CREAT` flag set, it creates the queue if it doesn't already exist. In this case the permissions are taken from the low-order 9 bits of the `flags` argument. The creator and owner user- and group-IDs are taken from the effective IDs of the process that issues the `msgget`.

For the permission bits, you can use the same `S_` flags that you use with `open` (Section 2.3). But make sure you use `IPC_CREAT`; don't use `O_CREAT` by mistake![4] There's also an `IPC_EXCL` flag you can use to make `msgget` fail if the queue already exists.

A `key` argument of `IPC_PRIVATE` allows you to create a message queue that's not associated with a specific key. Each time you call `msgget` with `IPC_PRIVATE` (the `IPC_CREAT` flag isn't needed) you get a different queue. This is ideal for a client process that wants its own message queue for replies from a server; it passes the identifier to the server, which can then use it to reply. There's no need for client and server to share a key, which would be awkward, since usually a server doesn't know in advance what clients it may have.

4. Why didn't the people who designed the System V IPC calls use the same symbols as `open` instead of making up their own symbols? Because back then (mid-1970s), `open` didn't use symbols. So the question should be, why didn't `open` use the same symbols as System V IPC? Because those symbols all started with the prefix `IPC_`, and, besides, the mainstream UNIX community within Bell Labs didn't like the IPC calls.

You can control an existing queue with `msgctl`:

msgctl—control message queue

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

int msgctl(
     int msqid,             /* identifier */
     int cmd,               /* command */
     struct msqid_ds *data  /* data for command */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

struct msqid_ds—structure for msgctl

```
struct msqid_ds {
  struct ipc_perm msg_perm; /* permission structure */
  msgqnum_t msg_qnum;       /* number of messages currently on queue */
  msglen_t msg_qbytes;      /* maximum number of bytes allowed on queue */
  pid_t msg_lspid;          /* process ID of last msgsnd */
  pid_t msg_lrpid;          /* process ID of last msgrcv */
  time_t msg_stime;         /* time of last msgsnd */
  time_t msg_rtime;         /* time of last msgrcv */
  time_t msg_ctime;         /* time of last msgctl change */
};
```

There are three values for the `cmd` argument:

`IPC_RMID` Remove the queue associated with `msqid`. The effective user-ID must be superuser or equal to the creator or owner user-IDs of the queue. The `data` argument isn't used and can be `NULL`.

`IPC_STAT` Fill the structure pointed to by `data` with information about the queue.

`IPC_SET` Set four properties of the queue from these members of the structure pointed to by data:

```
msg_perm.uid
msg_perm.gid
msg_perm.mode
msg_qbytes
```

The same permissions as for `IPC_RMID` are required, except that only the superuser can raise the value of `msg_qbytes`.

You put messages on and take messages from a queue with `msgsnd` and `msgrcv`.

msgsnd—send message

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

int msgsnd(
     int msqid,              /* identifier */
     const void *msgp,       /* message */
     size_t msgsize,         /* size of message */
     int flags               /* flags */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

msgrcv—receive message

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

ssize_t msgrcv(
     int msqid,              /* identifier */
     void *msgp,             /* message */
     size_t mtextsize,       /* size of mtext buffer */
     long msgtype,           /* message type requested */
     int flags               /* flags */
);
/* Returns number of bytes placed in mtext or -1 on error (sets errno) */
```

struct msg—typical structure for msgsnd and msgrcv

```
struct msg {
     long mtype;             /* message type */
     char mtext[MTEXTSIZE]; /* message text */
};
```

The structure you use for the message can be anything you like, as long as its first member is a `long` that's used for the type of message. This is so you can group messages that you send into types and then specify what types of messages you want to receive when you call `msgrcv`:

- If the `msgtype` argument to `msgrcv` is 0, you get the first message in the queue, regardless of type.
- If the argument is >0, you get the first message of that type.
- If the argument is <0, you get the first message of the lowest type that's less than or equal to the absolute value of `msgtype`. That is, if there are messages of types 5, 6, and 17 on the queue, and you specify –6, you will get the first message of type 5.

If you don't care about types, use 1 when you send (the type must be nonzero) and 0 for the `msgtype` argument to `msgrcv`. When types are used, it's most often to establish a priority system. You could also use a single queue for multiple serv-

ers and clients where each has its own type number, but that's an awkward way to organize things.

The `msgsize` argument to `msgsnd` is the size of just the message that's in the `mtext` member of the structure, not the whole structure, which includes the `mtype` member. Similarly, `msgrcv` returns the number of bytes of message in the `mtext` member, not the size of the whole structure. It's an error if the received message exceeds the `mtextsize` argument, unless you specify `MSG_NOERROR` in flags argument, in which case an oversize message is truncated, with no notice at all. This is usually a bad idea.

`msgsnd` ordinarily blocks if a limit on the number of messages in a queue or the total size of the messages on a queue would be exceeded. Or, you can specify `IPC_NOWAIT` (the System V IPC version of `O_NONBLOCK`) in the flags argument to instead make it return –1 with `errno` set to `EAGAIN`. (More on limits in the next section.)

`msgrcv` ordinarily blocks if a message of the requested type isn't present. Or, as with `msgsnd`, you can set the `IPC_NOWAIT` flag to make it nonblocking. Because message queues don't use file descriptors, you can't use `select` or `poll`, so try to avoid designs that require you to wait on more than one queue. You may be able to use a single queue instead with different message types to keep the contents straight. If you have to wait on multiple queues, consider using the techniques in Section 5.18.

7.5.2 System V Message-Queue Limits

There are a bunch of limits that serious applications are likely to bump up against, including:

- Limit on the total size of all messages in a queue. You can access this limit with the `msgctl` system call, through the `msg_qbytes` member of the `msqid_ds` structure.
- Limit on the number of messages in the queue.
- Limit on the size of a message.
- Limit on the number of queues.

Reaching the first two limits isn't necessarily a serious error. You can arrange for `msgsnd` to block or to return with an indication that a limit has been reached, as discussed in the previous section.

The other limits do cause an error, and there's nothing you can do about it other than reconfigure the kernel. You can't easily even determine what the limits are (`sysconf` won't get them), although it's fairly easy to run experiments to get an idea. For example, I hit a limit of 40 messages on Solaris, but that was the maximum number of messages for all queues. On FreeBSD, it was 20, and that seemed to be the limit per queue. There was no limit on Linux. For the maximum size of a message, I came up with 2048 bytes on Solaris and FreeBSD, and 8192 on Linux.[5]

Reconfiguring the kernel may be OK on your development system, but it's impractical if you're shipping your application for a customer to run on its own computers. Worse, the customer may have more than one application using System V IPC message queues, and the configuration instructions might conflict.

So here's some practical advice:

- Keep messages short—say, 1024 bytes or under. Use shared memory if you have to communicate more data. Shared memory also avoids two expensive copies (from one process to the kernel, and back to the other process).
- Don't push the other limits. Keep the number of queues small and don't assume that you can place any particular number of messages on a queue.
- During your application's installation, run some tests to ensure that the limits are adequate. Be prepared to advise the customer how to reconfigure the system if necessary.

7.5.3 System V Message-Queue Implementation of SMI

The message-queue system calls are actually very easy to use, as I'll demonstrate by implementing the SMI interface. You may want to review Section 7.3.1, where the interface was introduced, before proceeding.

The internal message queue is pretty simple compared to the one for FIFOs in Section 7.3.3:

```
typedef struct {
    SMIENTITY sq_entity;     /* entity */
    int sq_qid_server;       /* server identifier */
    int sq_qid_client;       /* client identifier (not used by server) */
```

5. The version of Darwin I used, 6.6, doesn't have System V messages.

```
    char sq_name[SERVER_NAME_MAX];  /* server name */
    struct client_id sq_client;     /* client ID */
    size_t sq_msgsize;              /* msg size */
    struct smi_msg *sq_msg;         /* msg buffer */
} SMIQ_MSG;
```

Note that that server doesn't have to retain information on each client to reduce the overhead when a message is to be returned because the identifier that will be passed with each message can be used directly in a call to `msgsnd`. In fact, the server doesn't even use the `sq_qid_client` member.

Opening an SMI message queue is also simple:

```
SMIQ *smi_open_msg(const char *name, SMIENTITY entity, size_t msgsize)
{
    SMIQ_MSG *p = NULL;
    char msgname[SERVER_NAME_MAX + 100];
    key_t key;

    ec_null( p = calloc(1, sizeof(SMIQ_MSG)) )
    p->sq_msgsize = msgsize + offsetof(struct smi_msg, smi_data);
    ec_null( p->sq_msg = calloc(1, p->sq_msgsize) )
    p->sq_qid_server = p->sq_qid_client = -1;
    p->sq_entity = entity;
    if (strlen(name) >= SERVER_NAME_MAX) {
        errno = ENAMETOOLONG;
        EC_FAIL
    }
    strcpy(p->sq_name, name);
    mkmsg_name_server(p, msgname, sizeof(msgname));
    (void)close(open(msgname, O_WRONLY | O_CREAT, 0));
    ec_neg1( key = ftok(msgname, 1) )
    if (p->sq_entity == SMI_SERVER) {
        if ((p->sq_qid_server = msgget(key, PERM_FILE)) != -1)
            (void)msgctl(p->sq_qid_server, IPC_RMID, NULL);
        ec_neg1( p->sq_qid_server = msgget(key, IPC_CREAT | PERM_FILE) )
    }
    else {
        ec_neg1( p->sq_qid_server = msgget(key, 0) )
        ec_neg1( p->sq_qid_client = msgget(IPC_PRIVATE, PERM_FILE) );
    }
    return (SMIQ *)p;

EC_CLEANUP_BGN
    if (p != NULL)
        (void)smi_close_msg((SMIQ *)p);
    return NULL;
EC_CLEANUP_END
}
```

```
static void mkmsg_name_server(const SMIQ_MSG *p, char *msgname,
  size_t msgname_max)
{
    snprintf(msgname, msgname_max, "/tmp/smimsg-%s", p->sq_name);
}
```

We use a file in the /tmp directory for each unique server name; the function `mkmsg_name_server` constructs the pathname. If that file doesn't already exist, the line just before the call to `ftok` creates it. We want the server to start with a fresh queue so it removes the message queue if it already exists before creating it. (In your own application you probably can't be so cavalier—you'll be concerned with what to do about running clients and whether you really want a fresh queue or whether you want to do more with existing messages than simply tossing them out.) The client gets access to the server queue, assuming the server started already, and creates a private queue for itself.

Here's `smi_close_msg`:

```
bool smi_close_msg(SMIQ *sqp)
{
    SMIQ_MSG *p = (SMIQ_MSG *)sqp;

    if (p->sq_entity == SMI_SERVER) {
        char msgname[FILENAME_MAX];

        (void)msgctl(p->sq_qid_server, IPC_RMID, NULL);
        mkmsg_name_server(p, msgname, sizeof(msgname));
        (void)unlink(msgname);
    }
    else
        (void)msgctl(p->sq_qid_client, IPC_RMID, NULL);
    free(p->sq_msg);
    free(p);
    return true;
}
```

Next come `smi_send_getaddr_msg` and `smi_send_release_msg`:

```
bool smi_send_getaddr_msg(SMIQ *sqp, struct client_id *client,
  void **addr)
{
    SMIQ_MSG *p = (SMIQ_MSG *)sqp;

    if (p->sq_entity == SMI_SERVER)
        p->sq_client = *client;
    *addr = p->sq_msg;
    return true;
}
```

```
bool smi_send_release_msg(SMIQ *sqp)
{
    SMIQ_MSG *p = (SMIQ_MSG *)sqp;
    int qid_receiver;

    p->sq_msg->smi_mtype = 1;
    if (p->sq_entity == SMI_SERVER)
        qid_receiver = p->sq_client.c_id1;
    else {
        qid_receiver = p->sq_qid_server;
        p->sq_msg->smi_client.c_id1 = p->sq_qid_client;
    }
    ec_neg1( msgsnd(qid_receiver, p->sq_msg,
      p->sq_msgsize - sizeof(p->sq_msg->smi_mtype), 0) )
    return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}
```

In `smi_send_release_msg`, if the client is sending, it already has the server's identifier, and it puts the identifier to its private queue in the message. If the server is sending, it takes the identifier to send to from the `SMIQ_MSG` structure where it was saved by the previous call to `smi_send_getaddr_msg`. You can also see now why we put a message type at the start of the `smi_msg` structure: It allows us to use that structure directly in calls to `msgsnd` and `msgrcv`. Note also that the size passed to `msgsnd` is just the data part, not the whole structure.

Finally, here are `smi_receive_getaddr_msg` and `smi_receive_release_msg`, which are even simpler than the sending pair:

```
bool smi_receive_getaddr_msg(SMIQ *sqp, void **addr)
{
    SMIQ_MSG *p = (SMIQ_MSG *)sqp;
    int qid_receiver;
    ssize_t nrcv;

    if (p->sq_entity == SMI_SERVER)
        qid_receiver = p->sq_qid_server;
    else
        qid_receiver = p->sq_qid_client;
    ec_neg1( nrcv = msgrcv(qid_receiver, p->sq_msg,
      p->sq_msgsize - sizeof(p->sq_msg->smi_mtype), 0, 0) )
    *addr = p->sq_msg;
    return true;
```

```
EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}

bool smi_receive_release_msg(SMIQ *sqp)
{
    return true;
}
```

Server and client both have the identifiers of their queues at hand in the `SMIQ_MSG` structure.

7.5.4 System V Message Queues Critiqued

This SMI implementation illustrates the best parts of System V message queues: A process can send a message to a queue given only the identifier, without the overhead of additional system calls to get access to it, and there's no worry about keeping messages atomic. We don't see it here, but even multiple receivers from the same queue are no problem (with FIFOs they're not safe).

One negative is that the various limits are inadequately specified (minimums would be nice) and awkward to query at run time. The biggest negative, however, true of all IPC mechanisms other than sockets, is that messages can be sent only within a single machine. For today's applications, that's often too restrictive.

The first edition of this book included this advice:

> [System V messages queues are] complex, incompletely documented, and nonportable. *Avoid them if at all possible!*

Reconsidering from a 2004 perspective, they no longer seem very complex, now that we have sockets, threads, and lots of other features that are even more complex. They are still incompletely documented, but, again, that doesn't make them so unusual. They are definitely portable—they're in the SUS and seem to be present on all the principal systems (Darwin soon, we hope). Thus, there's no reason to avoid them, as long as you know their limitations.

7.6 POSIX IPC

This section explains some general things about POSIX IPC that apply to messages, semaphores, and shared memory, which are covered in Sections 7.7, 7.10, and 7.14.

7.6.1 POSIX IPC History

POSIX IPC was introduced with POSIX 1003.1b-1993 (POSIX1993 for short) as part of the real-time extensions. The POSIX message-queue system calls described here constitute the message-passing option of that standard and are still optional as of POSIX2001. By contrast, System V IPC was never part of POSIX but is now mandatory on Open Group UNIX-certified systems.

Historically System V IPC was nonstandard yet almost universally implemented, whereas POSIX IPC has been a standard (but optional) for 10 years yet is still not available on many UNIX systems, notably Linux, FreeBSD, and Darwin. In theory POSIX IPC was more portable than System V IPC; in practice it was the reverse.

What's more, even when it is implemented, POSIX IPC isn't necessarily efficient enough for critical real-time applications. The standard requires no particular level of performance, and some OS vendors haven't committed the resources to any objectives beyond being conformant.

Therefore, as we'll see, the POSIX IPC system calls are in many ways more functional and more cleanly designed than the ones of System V IPC, but you may not be able to use them in your application.

7.6.2 POSIX IPC Names

Recall from Section 7.4.2 that System V IPC uses keys to specify its objects, with the added-on convenience of a scheme to generate keys from pathnames with the `ftok` system call. POSIX IPC uses a simpler scheme, with names (strings) instead of keys.

Names have to follow the same rules as for pathnames, and the standard requires that if the name begins with a slash then all references to that name refer to the same object. There's no such requirement if the name does not begin with a slash, but there are problems with this approach:

- There's no requirement that something like a file with the specified name actually appear in the file system. POSIX IPC objects, like System V IPC objects, are *not* files. This is for efficiency—file systems represent a lot of heavy machinery that a fast implementation of POSIX IPC is free to bypass,

using, for example, an in-memory hash table to find objects instead of using the file system's directory tree.
- On most UNIX systems, ordinary users can't write in the root directory, which creates a problem if the implementation actually creates a file corresponding to the name. A potential solution to this problem is for an administrator to first set up at least one subdirectory for general use (e.g., /ipc), but see the next point.
- The interpretation of slashes other than the first is implementation dependent. Some implementations might treat them as ordinary characters, some might treat them as directory delimiters, and some (such as Solaris) might prohibit them altogether. So, for portability, you should have no slashes other than the leading one. This rules out the solution in the previous point.

So, what to do? Probably write a small initialization program that just creates all the objects your application will need (using names with a leading slash only) and set it up so its effective user ID is the superuser or whatever user can write into the root directory. For development, either use the initialization program or run on a system that doesn't create actual files. The examples in this book were developed on Solaris, which doesn't create files, so the pathnames that appear to be in the root directory aren't really there and everything works just fine.

An alternative approach is to `#ifdef` your code for various systems, using whatever naming scheme that system supports. But generally this is a bad idea because whatever small set of systems you decide to include will undoubtedly be insufficient someday, and then somebody will have to modify the code to include the new system. Then the modified code has to be worked back into the base code, with all the support and maintenance headaches that invariably result.

Yet another idea is to put the pattern for the POSIX IPC names in a configuration file that's read at run-time. Then choosing the appropriate scheme becomes an installation option, rather than a compile-time option. This might be OK, but it still adds to the many ways that a complex application can be installed incorrectly, which is something you don't need. Maybe the System V IPC scheme using keys and `ftok` wasn't so silly after all.

7.6.3 POSIX IPC Feature-Test Macros

As explained at length in Section 1.5.4, the SUS and earlier POSIX standards provided macros that you can check at compile time to test whether optional features

are present, absent, or whether you have to code a run-time check. Unfortunately, the systems that implement the macros exactly as they're supposed to usually have the optional features (which means you could have skipped the test), and the systems that don't have the optional features don't implement the macros correctly either. Therefore, in practice, testing the macros doesn't actually lead to portability.

But, if you want to give it a try, read carefully what I said in Chapter 1 about how they work in general and then consult [SUS2002] or the relevant POSIX standard to see what specific macros to test.

I generally won't use macro tests in my examples. They compile and run on the systems that support the optional features, and they don't compile on the others.

7.6.4 POSIX IPC Utilities

For System V, we had the handy `ipcs` and `ipcrm` utilities (Section 7.4.4). There aren't any for POSIX IPC, so this section is short. Too bad if you want to check what message queues, semaphores, or shared-memory segments might be left around by errant applications.

7.7 POSIX Message Queues

This section explains POSIX message queues and shows how they can be used to implement the SMI.

7.7.1 POSIX Message-Queue System Calls

It turns out that the name issue is the only real sticky part of using POSIX message queues. The rest is easy enough. First, `mq_open` opens a message queue.

`mq_open` behaves much like `open`, although the object it creates can't portably be treated as a file, and it returns a message-queue descriptor, not a file descriptor. Regrettably, you can't use the message-queue descriptor with `select` or `poll`, which is the same disadvantage that we saw with System V message queues. However, you can be notified with a signal when a message is available; see `mq_notify`, below.

mq_open—open message-queue

```
#include <mqueue.h>

mqd_t mq_open(
     const char *name,      /* POSIX IPC name */
     int flags              /* flags (excluding O_CREAT) */
);
/* Returns message-queue descriptor or -1 on error (sets errno) */

mqd_t mq_open(
     const char *name,      /* POSIX IPC name */
     int flags,             /* flags (including O_CREAT) */
     mode_t perms,          /* permissions */
     struct mq_attr *attr   /* attributes (or NULL) */
);
/* Returns message-queue descriptor or -1 on error (sets errno) */
```

struct mq_attr—structure for mq_open, mq_getattr, and mq_setattr

```
struct mq_attr {
     long mq_flags;         /* flags */
     long mq_maxmsg;        /* max number of messages */
     long mq_msgsize;       /* max message size */
     long mq_curmsgs;       /* number of messages currently queued */
};
```

The macros for the `flags` argument are the same as for `open` (e.g., `O_CREAT`), not special macros as used with `msgget` (e.g., `IPC_CREAT`):

- You must specify one of `O_RDONLY`, `O_WRONLY`, or `O_RDWR`, depending on whether you want to receive messages only, send messages only, or both.
- You specify `O_CREAT` if you want to create the queue if it doesn't already exist. In this case you supply four arguments, as shown in the second form in the synopsis box.
- You specify `O_EXCL` if you want the function to fail when the queue already exists.
- You specify `O_NONBLOCK` if you want the message-queue descriptor to be nonblocking, which means that `mq_send` and `mq_receive` return –1 with `errno` set to `EAGAIN` if the queue is full or empty, respectively. The default behavior is blocking.

When `O_CREAT` is set and the queue is actually created, the interpretation of the `perms` argument is similar to the permissions on files and System V message queues, with the usual `S_` flags (Section 2.3). Read and write permission mean the ability to receive and send messages, respectively, and execute doesn't mean anything. As with files, the queue has a user- and group-ID, and they're set from the effective IDs of the process that created it. Subsequently, the interaction of the

owner, group, and other bits with the effective user- and group-IDs is that same as with files.

Also, if `O_CREAT` is specified and a queue is created and the `attr` argument is non-`NULL`, two attributes, `mq_maxmsg` and `mq_msgsize` are set from the supplied `mq_attr` structure. These are the maximum number of messages that the queue can hold and the maximum size for an individual message. The standard doesn't specify a limit for either attribute, but if there's insufficient space `mq_open` will return –1 with `errno` set to `ENOSPC`. Most implementations will implement the queue with a linked list, so `mq_open` won't run out of space regardless of the numbers because a new queue is empty. If the `attr` argument is `NULL`, the two attributes are set to implementation-defined values.

If your eyes glazed over reading the previous few paragraphs and you go ahead assuming `mq_open` is just like `open`, you'll probably be just fine as long as you don't assume it returns a file descriptor.

You close an open message queue with `mq_close`:

mq_close—close message queue

```
#include <mqueue.h>

int mq_close(
    mqd_t mqd              /* message-queue descriptor */
);
/* Returns zero on success or -1 on error (sets errno) */
```

As with System V message queues, POSIX message queues persist until they're removed or the kernel is rebooted. You remove a POSIX message queue with a call that's like `unlink`:

mq_unlink—remove message queue

```
#include <mqueue.h>

int mq_unlink(
    const char *name       /* POSIX IPC name */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

Like `unlink`, `mq_unlink` makes the name disappear right away, but actual destruction of the queue is postponed until all open message-queue descriptors are closed. (Recall that with System V message queues, removing a queue with `msgctl` was immediate and potentially disruptive.) So, if you want to, once all

processes that need the queue have opened it, you can `mq_unlink` it, just as you sometimes do with temporary files.

OK, ready to send a message? You call `mq_send`, as you might have guessed:

mq_send—send message

```
#include <mqueue.h>

int mq_send(
     mqd_t mqd,             /* message-queue descriptor */
     const char *msg,       /* message */
     size_t msgsize,        /* size of message */
     unsigned priority      /* priority */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

`mq_send` places the message of size `msgsize` onto the queue, positioning it in front of all messages with a lower `priority`, but after messages with the same `priority` (in other words, just as you would expect). Priorities go from zero to 31, at least; you can retrieve the actual maximum value with `sysconf` (Section 1.5.5).

As mentioned above, `mq_send` blocks if the queue is full, unless the `O_NONBLOCK` flags is set.

The call to receive a message is, of course, `mq_receive`:

mq_receive—receive message

```
#include <mqueue.h>

ssize_t mq_receive(
     mqd_t mqd,             /* message-queue descriptor */
     char *msg,             /* message buffer */
     size_t msgsize,        /* size of message buffer */
     unsigned *priorityp    /* returned priority or NULL */
);
/* Returns size of message or -1 on error (sets errno) */
```

`mq_receive` gets the oldest of the highest priority messages, which, because of the way `mq_send` is defined, is the same as saying the first message on the queue.

The `msgsize` argument is a little tricky: It's the size of the buffer pointed to by the `msg` argument, as you would expect, but it must be at least as great as the `mq_msgsize` attribute (see `mq_open`, above), or `mq_receive` will fail, *even if the front-most message is small enough to fit*. This is actually an excellent design choice: Catch the problem of a too-small buffer right away, rather than requiring

just the right test data to expose the bug. The actual size of the received message is the return value.

If the `priorityp` argument is non-`NULL`, it gets the priority of the received message. The received message is removed from the queue; there's no way to just peek at it. If the queue is empty, `mq_receive` blocks, unless `O_NONBLOCK` is set.

There are variants of `mq_send` and `mq_receive` that allow a blocked call to timeout:

mq_timedsend—send message with timeout

```
#include <mqueue.h>
#include <time.h>

int mq_timedsend(
     mqd_t mqd,                     /* message-queue descriptor */
     const char *msg,               /* message */
     size_t msgsize,                /* size of message */
     unsigned priority              /* priority */
     const struct timespec *tmout   /* timeout */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

mq_timedreceive—receive message with timeout

```
#include <mqueue.h>
#include <time.h>

ssize_t mq_timedreceive(
     mqd_t mqd,                     /* message-queue descriptor */
     char *msg,                     /* message buffer */
     size_t msgsize,                /* size of message buffer */
     unsigned *priorityp            /* returned priority or NULL */
     const struct timespec *tmout   /* timeout */
);
/* Returns size of message or -1 on error (sets errno) */
```

The timeout applies only if the call would block and `O_NONBLOCK` is clear. When that amount of time elapses, the function returns with –1 and `errno` set to `ETIMEDOUT`. These two functions are part of the Timeouts option (`_POSIX_TIMEOUTS`) and are new with SUS3.

As I said, it's awkward to handle more than one message queue or a message queue in combination with something else that can block, such as terminal or a network connection since you can't test a message-queue descriptor with `select` or `poll`. However, you can arrange to be notified with a signal when a message arrives, and then you can call `mq_receive` without worrying about blocking. (See Section 5.18 for another solution.) You set this up with `mq_notify`.

mq_notify—register or unregister for message notification

```
#include <mqueue.h>

int mq_notify(
    mqd_t mqd,                      /* message-queue descriptor */
    const struct sigevent *ep       /* notification */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

When a process or thread is registered, it receives a signal when a message arrives on an empty queue designated by `mqd`, unless one or more processes or threads are already blocking in an `mq_receive`, in which case one of the `mq_receives` returns instead.

Only one process or thread may be registered; attempting to register a second process or thread results in an error. A process or thread is unregistered when a signal is sent or when `mq_notify` is called with a `NULL` second argument.

There's no notification of a message arrival on a nonempty queue, so `mq_notify` all by itself doesn't ensure that a process will get a signal every time a message arrives. For example, it may get a signal when a message arrives on an empty queue, but then a second message may arrive before the first message is received, which would not cause notification, since the queue was not empty. So, an application needs to do something like this:

- After calling `mq_notify`, repeatedly call `mq_receive` with `O_NONBLOCK` set until the queue is empty.
- When a signal arrives, immediately call `mq_notify` again and then do the previous step.

What signal is sent and other properties of the notification are determined by the contents of the `sigevent` structure passed to `mq_notify`, as explained in Section 9.5.6.

Finally, you can get and set message-queue attributes:

mq_getattr—get message-queue attributes

```
#include <mqueue.h>

int mq_getattr(
    mqd_t mqd,              /* message-queue descriptor */
    struct mq_attr *attr    /* attributes */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

mq_setattr—set message queue attributes

```
#include <mqueue.h>

int mq_setattr(
    mqd_t mqd,                      /* message-queue descriptor */
    const struct mq_attr *attr,     /* new attributes */
    struct mq_attr *oldattr         /* old attributes if not NULL */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

`mq_getattr` fills the structure (shown along with `mq_open`) according to the current state of the queue. The most useful attribute is probably member `mq_curmsgs`, which contains the current number of messages on the queue.

`mq_setattr` sets or clears the `O_NONBLOCK` flag in the `mq_flags` member; by the standard, it has no effect on anything else. (Implementations might have other nonportable flags that can be set.) Thus, for the `O_NONBLOCK` flag, `mq_setattr` is used on message-queue descriptors as `fcntl` is used on file descriptors. If `oldattr` is non-`NULL`, it's used to get the old attributes returned.

7.7.2 POSIX Message-Queue Implementation of SMI

Now for our third implementation of the SMI functions. First, here is the internal message queue, which is almost the same as the one for FIFOs, because it has similar issues to deal with:

```
#define MAX_CLIENTS 20

typedef struct {
    SMIENTITY sq_entity;            /* entity */
    mqd_t sq_mqd_server;            /* server message-queue descriptor */
    char sq_name[SERVER_NAME_MAX];  /* server name */
    struct client_info {            /* Client uses only sq_clients[0] */
        mqd_t cl_mqd;               /* client message-queue descriptor */
        pid_t cl_pid;               /* client process ID */
    } sq_clients[MAX_CLIENTS];
    struct client_id sq_client;     /* client ID */
    size_t sq_msgsize;              /* msg size */
    struct smi_msg *sq_msg;         /* msg buffer */
} SMIQ_MQ;
```

As with FIFOs, to avoid the overhead of opening a client's message queue for each message, the server is going to keep track of each client it knows about and just look up the stored message-queue descriptor for an already-open queue. In many real servers, this isn't really such a burden because it would almost always

want to keep track of its clients anyway. The client will only use the first element of the array to store its own message-queue descriptor, but we don't care about wasted space in this example.

There are two functions to translate SMI server names and client process IDs to POSIX IPC names:

```
static void make_mq_name_server(const SMIQ_MQ *p, char *mqname,
  size_t mqname_max)
{
    snprintf(mqname, mqname_max, "/smimq-s%s", p->sq_name);
}

static void make_mq_name_client(pid_t pid, char *mqname,
  size_t mqname_max)
{
    snprintf(mqname, mqname_max, "/smimq-c%d", pid);
}
```

These names look like they're in the root directory, but on Solaris they're not, as explained in Section 7.6.2.

Next comes a function used by the server to get a client's message-queue descriptor from its process ID, which is what a client sends in a message to the server:

```
static mqd_t get_client_mqd(SMIQ_MQ *p, pid_t pid)
{
    int i, avail = -1;
    char mqname[SERVER_NAME_MAX + 100];

    for (i = 0; i < MAX_CLIENTS; i++) {
        if (p->sq_clients[i].cl_pid == pid)
            return p->sq_clients[i].cl_mqd;
        if (avail == -1 && p->sq_clients[i].cl_pid == 0)
            avail = i;
    }
    errno = ECONNREFUSED;
    ec_neg1( avail )
    p->sq_clients[avail].cl_pid = pid;
    make_mq_name_client(pid, mqname, sizeof(mqname));
    ec_neg1( p->sq_clients[avail].cl_mqd = mq_open(mqname, O_WRONLY) )
    return p->sq_clients[avail].cl_mqd;

EC_CLEANUP_BGN
    return (mqd_t)-1;
EC_CLEANUP_END
}
```

The function first looks in the array to see if it's already seen the client, in which case it already has the message-queue descriptor. If not, it opens the queue for writing and saves the descriptor for next time.

Now we're ready for the first SMI function, `smi_open_mq`:

```
static pid_t my_pid;

SMIQ *smi_open_mq(const char *name, SMIENTITY entity, size_t msgsize)
{
    SMIQ_MQ *p = NULL;
    char mqname[SERVER_NAME_MAX + 100];
    struct mq_attr attr = {0};

    my_pid = getpid();
    ec_null( p = calloc(1, sizeof(SMIQ_MQ)) )
    p->sq_msgsize = msgsize + offsetof(struct smi_msg, smi_data);
    ec_null( p->sq_msg = calloc(1, p->sq_msgsize) )
    p->sq_entity = entity;
    p->sq_mqd_server = p->sq_clients[0].cl_mqd = (mqd_t)-1;
    if (strlen(name) >= SERVER_NAME_MAX) {
        errno = ENAMETOOLONG;
        EC_FAIL
    }
    strcpy(p->sq_name, name);
    make_mq_name_server(p, mqname, sizeof(mqname));
    attr.mq_maxmsg = 100;
    attr.mq_msgsize = p->sq_msgsize;
    if (p->sq_entity == SMI_SERVER) {
        (void)mq_unlink(mqname);
        ec_cmp( errno, ENOSYS )
        ec_neg1( p->sq_mqd_server = mq_open(mqname, O_RDONLY | O_CREAT,
          PERM_FILE, &attr) )
    }
    else {
        ec_neg1( p->sq_mqd_server = mq_open(mqname, O_WRONLY) )
        make_mq_name_client(my_pid, mqname, sizeof(mqname));
        ec_neg1( p->sq_clients[0].cl_mqd = mq_open(mqname,
          O_RDONLY | O_CREAT, PERM_FILE, &attr) )
    }
    return (SMIQ *)p;

EC_CLEANUP_BGN
    if (p != NULL)
        (void)smi_close_mq((SMIQ *)p);
    return NULL;
EC_CLEANUP_END
}
```

After initializing the internal SMI queue, we initialize an `mq_attr` structure with limits of 100 messages on the queue and a message size that was passed as the third argument. Then, if we're the server, we discard any existing server queue and create a fresh one. We don't bother doing this for a client, although we probably should, because the client's process ID is embedded in the queue name, and it's unlikely to be reused.

The close function looks like this, doing what you'd expect:

```
bool smi_close_mq(SMIQ *sqp)
{
    SMIQ_MQ *p = (SMIQ_MQ *)sqp;
    char msgname[SERVER_NAME_MAX + 100];

    if (p->sq_entity == SMI_SERVER) {
        make_mq_name_server(p, msgname, sizeof(msgname));
        (void)mq_close(p->sq_mqd_server);
        (void)mq_unlink(msgname);
    }
    else {
        make_mq_name_client(my_pid, msgname, sizeof(msgname));
        (void)mq_close(p->sq_mqd_server);
        (void)mq_unlink(msgname);
    }
    free(p->sq_msg);
    free(p);
    return true;
}
```

Next comes the `smi_send_getaddr_mq` and `smi_send_release_mq`:

```
bool smi_send_getaddr_mq(SMIQ *sqp, struct client_id *client,
  void **addr)
{
    SMIQ_MQ *p = (SMIQ_MQ *)sqp;

    if (p->sq_entity == SMI_SERVER)
        p->sq_client = *client;
    *addr = p->sq_msg;
    return true;
}

bool smi_send_release_mq(SMIQ *sqp)
{
    SMIQ_MQ *p = (SMIQ_MQ *)sqp;
    mqd_t mqd_receiver;
```

```
    if (p->sq_entity == SMI_SERVER)
        ec_neg1( mqd_receiver = get_client_mqd(p, p->sq_client.c_id1) )
    else {
        mqd_receiver = p->sq_mqd_server;
        p->sq_msg->smi_client.c_id1 = my_pid;
    }
    ec_neg1( mq_send(mqd_receiver, (const char *)p->sq_msg,
      p->sq_msgsize, 0) )
    return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}
```

As before with FIFOs and System V message queues, `smi_send_getaddr_mq` just saves the client ID (if it's called from the server) and returns the buffer address. The real work is in `smi_send_release_mq`. The server calls the look-up function `get_client_mqd` to get the message-queue descriptor. Its argument was saved in the `SMIQ_MQ` structure by `smi_send_getaddr_mq`.

The client already has the server's descriptor, and it puts its process ID into the message. (The message-queue descriptor can't be passed between processes, like a System V message-queue identifier can.)

The receive pair are simpler since a receiver already has the descriptor to its message queue:

```
bool smi_receive_getaddr_mq(SMIQ *sqp, void **addr)
{
    SMIQ_MQ *p = (SMIQ_MQ *)sqp;
    mqd_t mqd_receiver;
    ssize_t nrcv;

    if (p->sq_entity == SMI_SERVER)
        mqd_receiver = p->sq_mqd_server;
    else
        mqd_receiver = p->sq_clients[0].cl_mqd;
    ec_neg1( nrcv = mq_receive(mqd_receiver, (char *)p->sq_msg,
      p->sq_msgsize, NULL) )
    *addr = p->sq_msg;
    return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}
```

```
bool smi_receive_release_mq(SMIQ *sqp)
{
    return true;
}
```

7.7.3 POSIX Message Queues Critiqued

POSIX message-queue system calls have a somewhat cleaner interface than the System V calls and are more integrated into the rest of UNIX, but they still use their own descriptor rather than a file descriptor so `select` and `poll` can't be used. To compensate, there is a notification feature, but it's very tricky to use, as we'll see in Chapter 9, because it's based on signals, which are always tricky to use.

The main problem with POSIX message queues (all POSIX IPC, actually) is that it's not widely available, but, of course, this isn't a criticism of the design.

Table 7.1 compares POSIX and System V message queues in more detail:

Table 7.1 POSIX vs. System V Message Queues

Criterion	POSIX	System V
Standardized?	yes	yes
Mandatory in UNIX-certified systems?	no	yes
Available in all major UNIX systems?	no	yes
Message limits?	very few; settable	many; only one settable; hard to manage
Thread safe (according to SUS3)?	yes	yes
Message priorities?	yes	yes
Receive other than highest-priority message?	no	yes
Notification?	yes	no

Table 7.1 POSIX vs. System V Message Queues (cont.)

Criterion	POSIX	System V
Uses file descriptors?	no	no
Efficiency?	depends on implementation	depends on implementation
Two user-kernel copies of each message?*	yes	yes
Portability issues with queue names?	yes	no
* This means that data is copied from user space to kernel space, or the other way around.		

With this mixed bag of pros and cons, why did the POSIX real-time group invent a new message system when System V messages had been around for over 10 years? And, since they did, why aren't the results better?

Here are my answers:

- While the POSIX group certainly had a lot of quibbles with the existing System V approach, the main reason they needed a new standard was that many major systems already had a System V implementation that was unsuitable for real-time applications, and it would be hard to have two simultaneous implementations behind the same interface. They needed a way to give implementors a fresh start.
- But, there was no practical way within the POSIX process to require any particular level of performance, so on many, if not most, systems, the efficiency of the two (when there even are two) is about the same.
- The nonportability of POSIX IPC names is caused by underspecification, which was required in order to allow a broad set of implementations, including very fast in-memory queues with no file-system lookup of any kind.
- The POSIX group is not responsible for the lack of implementations. What's probably most responsible is that by the mid-1990s many, if not most, application developers were focusing on network applications, for which they used sockets (Chapter 8), and there wasn't much interest in an alternative way to handle non-networked messages. So well-funded outfits like Sun implemented the POSIX option packages, and shoestring outfits like the BSD and Linux communities found more pressing things to do.

7.8 About Semaphores

This section explains general properties of semaphores and shows how they can be implemented with files and messages. The System V and POSIX semaphore system calls are covered in Sections 7.9 and 7.10.

7.8.1 Basic Semaphore Usage

We already encountered mutexes in Section 5.17.3. More generally, a semaphore is a counter that prevents two or more processes or threads from accessing a shared resource simultaneously. It's only advisory, however: if a process or thread doesn't check a semaphore before accessing a shared resource, chaos might result.

In UNIX, the term "semaphore" usually applies to an object manipulated by the semaphore system calls to synchronize processes. The term "mutex" usually means a much lighter-weight object that's used to synchronize threads, which is what we discussed in Section 5.17.3. I'll talk only about semaphores in this section.

A *binary* semaphore has only two states: locked and unlocked. A *general* semaphore has an infinite (or, at least, very large) number of states. It's a counter that decreases by one when it is acquired ("locked") and increases by one when it is released ("unlocked"). If it's zero, a process trying to acquire it must wait for another process to increase its value—it can't ever get negative. (If a semaphore can take on only two values, 0 and 1, it is a binary semaphore.) General semaphores are usually accessed with two operations that we can refer to abstractly as *semwait* and *sempost* (sometimes called P and V[6]). We can try to write these in C:

```
void semwait(int *sem)
{
    while (*sem <= 0)
        ; /* do nothing */
     (*sem)--;
}

void sempost(int *sem)
{
     (*sem)++;
}
```

6. P and V are abbreviations used by E. W. Dijkstra for the Dutch phrase *proberen te verlagen* ("try to decrease") and the word *verhogen* ("increase"). Some books say that P stands for *prolagen*, but that isn't a word, not even in Dutch. It's a contraction of the full phrase.

The semaphore must be initialized by calling *sempost;* otherwise it starts out with nothing to acquire. For example, if the semaphore counts the number of free buffers, and there are initially five buffers, we would start out by calling *sempost* five times. Alternatively, we can just set the semaphore variable to five.

Having gone to the trouble of programming *semwait* and *sempost,* I now have to say that they won't work, for three reasons:

- The semaphore variable pointed to by `sem` isn't normally shared among processes, which have distinct data segments. (Although it could be in shared memory.)
- The functions do not execute atomically—the kernel can interrupt a process at any time. The following scenario could occur: Process 1 completes the `while` loop in *semwait* and is interrupted before it can decrement the semaphore; process 2 enters *semwait,* finds the semaphore equal to 1, completes its `while` loop, and decrements the semaphore to 0; process 1 resumes and decrements the semaphore to –1 (an illegal value).
- *semwait* does what's called a *busy-wait* if `sem` is zero. This is a dumb way to use a CPU.

Therefore, *semwait* and *sempost* can't just be programmed in user space. Semaphores have to be supplied by the kernel, which can share data between processes, can execute atomic operations, and can give the CPU to a ready process when a process blocks.

7.8.2 Implementing Semaphores with Files and Messages

Back in Section 2.4.3 I showed how to use `open` to make a crude binary semaphore. That method is OK for a process that accesses a shared resource only a few times, such as a mail program that's writing into a mailbox file, but the overhead is far too great for heavy-duty use. We want a semaphore that takes less time to check and set.

A message queue can also be used as a semaphore: A send operation adds a message to the queue and is equivalent to *sempost;* a receive removes a message from the queue and is equivalent to *semwait*. A receive blocks when the queue is empty, which is equivalent to the semaphore being zero.

However, any UNIX system that supports messages (System V or POSIX varieties) also supports semaphores in their own right, and much more efficiently.

7.9 System V Semaphores

Neither System V nor POSIX semaphores use anything as simple as *semwait* and *sempost*. Their system calls are more complicated, although POSIX semaphores are not much more complicated. I'll critique the two together in Section 7.10.3.

I'll start with System V. These system calls are too complex for me to describe completely. I'll make sure, however, that I explain enough to allow us to implement *semwait* and *sempost;* for the rest, see your system's documentation or [SUS2002].

The information in Section 7.4 applies to these system calls so we don't need to say anything specific about keys, identifiers, ownership, etc.

What's interesting about the System V semaphore facility is that the system calls operate not just on a single semaphore, but on a whole array at once. With one atomic operation, you can do a *semwait* on several and a *sempost* on several others. It's questionable whether you will ever want to do this in practice, but it's there if you need it. In most of our examples we'll operate on only one semaphore at a time, although we will later on use a set of two, just for convenience.

One comment before we go on: System V semaphores are too complicated! In any program using semaphores it's essential to be able to demonstrate that access to shared resources is exclusive, that deadlock does not occur, and that starvation (never getting access) does not occur. Testing alone can't usually suffice because things are so timing-dependent; analysis must be used. This is difficult enough with plain *semwait* and *sempost*. With the System V system calls, used in their full glory, analysis is probably impossible.

7.9.1 System V Semaphore System Calls

As with System V message queues, you start with the *X*`get` call:

semget—get semaphore-set identifier

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

int semget(
    key_t key,                  /* key */
    int nsems,                  /* size of set */
    int flags                   /* flags */
);
/* Returns identifier or -1 on error (sets errno) */
```

As you would expect, `semget` translates a key to an ID representing a set of semaphores. If the `IPC_CREAT` bit of `flags` is on, the set is created if it doesn't already exist. There are `nsems` semaphores in the set, numbered starting with zero.

The semaphores aren't ready to use right away—they have to be initialized with a call to `semctl`. Why `semget` doesn't initialize them to zero is a mystery, but the SUS doesn't require it to do so,[7] and many implementations don't. Here's `semctl`:

semctl—control semaphore set

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

int semctl(
     int semid,                 /* identifier */
     int semnum,                /* semaphore number */
     int cmd,                   /* command */
     union semun arg            /* argument for command */
);
/* Returns value or 0 on success; -1 on error (sets errno) */
```

union semun—union for semctl

```
union semun {
    int val;                    /* integer */
    struct semid_ds *buf;       /* pointer to structure */
    unsigned short *array;      /* array */
};
```

struct semid_ds—structure for semctl

```
struct semid_ds {
     struct ipc_perm sem_perm; /* permission structure */
     unsigned short sem_nsems; /* size of set */
     time_t sem_otime;         /* time of last semop */
     time_t sem_ctime;         /* time of last semctl */
};
```

Oddly, `union semun` isn't defined in the header sem.h—you have to define it yourself.

7. Actually, the SUS says, "The data structure associated with each semaphore in the set shall not be initialized." The use of the phrase "shall not" means that initializing it is not even an implementation option. Is this intentional or just sloppy writing?

There are seven commands unique to semaphores, in addition to the common System V IPC commands IPC_RMID, IPC_STAT, and IPC_SET:

GETNCNT Get the number of processes blocked waiting for semaphore semnum to increase.

GETZCNT Get the number of processes blocked waiting for semaphore semnum to become zero.

GETPID Get the ID of the process to last perform a semop.

GETVAL Get the value of semaphore semnum.

SETVAL Set the value of semaphore semnum. Uses arg.val.

GETALL Get the values of all the semaphores in the set. Uses arg.array.

SETALL Set the values of all the semaphores in the set. Uses arg.array.

Amazingly, if you have a set of 100 semaphores and you want to set them all to zero with a single semctl, you have to build an array of 100 zeros to use as the fourth argument! Doesn't bother us, because we always initialize them one at a time, with SETVAL.

The IPC_RMID command acts just like it does with msgctl (Section 7.5.1).

IPC_STAT fills the semid_ds structure passed via the fourth argument, using the buf member of the union. IPC_SET can be used only to set the sem_perm.uid, sem_perm.gid, and sem_perm.mode members.

Since, as I said, semget doesn't initialize the semaphores, you use semctl to do it, as in this sequence:

```
ec_neg1(semid = semget(key, 1, PERM_FILE | IPC_CREAT) )
arg.val = 0;
ec_neg1( semctl(semid, 0, SETVAL, arg) )
```

Unfortunately, as it takes two system calls to get the semaphore created and initialized, a second process or thread that does a semget on the same semaphore may start processing with an uninitialized semaphore before the semctl is executed. The solution is to take advantage of the fact that, while the value of a new semaphore isn't initialized, its "last semop time," which is stored in the

sem_otime member of the semid_ds structure, is initialized to zero by semget. This suggests[8] the following scheme:

- The process or thread that creates the semaphore also calls semctl to initialize it and then does a semop on it so sem_otime will take on a nonzero value.
- Any other process or thread that gets the semaphore ID waits for the time to become nonzero.

This scheme works only for brand-new semaphores. If you restart an application with a semaphore left around from a previous run, that semaphore will already have a value and a nonzero value for sem_otime, which will probably mess things up. So, make sure you clean up between runs of the application by removing the semaphore, either with a call to semctl or with the ipcrm command (Section 7.4.4).

Unfortunately, as of this writing, waiting for sem_otime doesn't work on FreeBSD or Darwin because the time is never updated.[9] Therefore, System V semaphores can't be used reliably at all unless you're sure because of the nature of the application that the initialization will occur before the second attempt to semget the semaphore. (Even more unfortunately, those systems don't completely support POSIX semaphores either.)

The system call to operate on a semaphore set, semop, is in the next section.

7.9.2 Simple Semaphore Interface

We can hide the complexities of semaphores with a simple semaphore-open call that we'll implement for System V now, and for POSIX semaphores a bit later on. Here's the synopsis for SimpleSemOpen and its companion, SimpleSemClose:

SimpleSemOpen—open simple semaphore

```
#include "SimpleSem.h"

struct SimpleSem *SimpleSemOpen(
    const char *name       /* name (follows System V or POSIX rules) */
);
/* Returns pointer to structure or NULL on error (sets errno) */
```

8. I learned of this technique from [Ste1999], p. 284. I had it wrong in the first edition of *Advanced UNIX Programming*.
9. I've been told it's fixed in FreeBSD 5.1.

SimpleSemClose—close simple semaphore

```
#include "SimpleSem.h"

bool SimpleSemClose(
    struct SimpleSem *sem  /* semaphore */
);
/* Returns true on success or false on error (sets errno) */
```

struct SimpleSem—structure for simple-semaphore functions

```
struct SimpleSem {
    union {
        int sm_semid;       /* System V semaphore-set ID */
        void *sm_sem;       /* POSIX sem_t pointer (needs a cast) */
    } sm;
};
```

`SimpleSemOpen` has no options at all: It opens a single semaphore by name (getting the key itself, in the case of a System V implementation), creating it if necessary with permissions `PERM_FILE` (from Section 2.3). It returns a pointer to a `SimpleSem` structure, which contains whatever the other functions need to identify the semaphore set (of one), which for System V is an integer semaphore-set ID. (We'll get to what POSIX semaphores require later.) But the user of the simple-semaphore package doesn't need to worry about what's inside a `SimpleSem` structure, as a pointer to it is just passed to the other functions, like `SimpleSemClose`, which closes the semaphore and frees any memory used by the SimpleSem structure.

Here's a System V implementation of `SimpleSemOpen`, where you can see the scheme outlined above for ensuring that the semaphore is properly initialized:

```
struct SimpleSem *SimpleSemOpen(const char *name)
{
    struct SimpleSem *sem = NULL;
    key_t key;
    union semun {
        int val;
        struct semid_ds *buf;
        unsigned short *array;
    } arg;
    struct sembuf sop;

    (void)close(open(name, O_WRONLY | O_CREAT, 0));
    ec_neg1( key = ftok(name, 1) )
    ec_null( sem = malloc(sizeof(struct SimpleSem)) )
```

```
    if ((sem->sm.sm_semid = semget(key, 1,
      PERM_FILE | IPC_CREAT | IPC_EXCL)) != -1) {
        arg.val = 0;
        ec_neg1( semctl(sem->sm.sm_semid, 0, SETVAL, arg) )
        sop.sem_num = 0;
        sop.sem_op = 0;
        sop.sem_flg = 0;
        ec_neg1( semop(sem->sm.sm_semid, &sop, 1) )
    }
    else {
        if (errno == EEXIST) {
            while (true)
                if ((sem->sm.sm_semid = semget(key, 1, PERM_FILE)) == -1) {
                    if (errno == ENOENT) {
                        sleep(1);
                        continue;
                    }
                    else
                        EC_FAIL
                }
                else
                    break;
            while (true) {
                struct semid_ds buf;

                arg.buf = &buf;
                ec_neg1( semctl(sem->sm.sm_semid, 0, IPC_STAT, arg) )
                if (buf.sem_otime == 0) {
                    sleep(1);
                    continue;
                }
                else
                    break;
            }
        }
        else
            EC_FAIL
    }
    return sem;

EC_CLEANUP_BGN
    free(sem);
    return NULL;
EC_CLEANUP_END
}
```

The sequence

```
    (void)close(open(name, O_WRONLY | O_CREAT, 0));
    ec_neg1( key = ftok(name, 1) )
```

is the same as what we did in Section 7.5.3. It creates the name if it doesn't already exist, and any problems with the name are reported as problems with `ftok`, not with `open` or `close`. (If you don't like that, you can check the `open` for an error other than `ENOENT` separately, instead of embedding it in an immediate call to `close`.)

Then we do a `semget` with the `IPC_CREAT` and `IPC_EXCL` flags so that it will fail if the semaphore already exists. All processes and threads for which it failed for that reason need to wait for it to be initialized. The process or thread that got a successful return from `semget` uses `semctl` to initialize it, and then it does a `semop` (which I'll explain shortly) that doesn't do anything to the value, but as a side-effect sets the `sem_otime`.

Processes and threads that need to wait will loop until a `semget` succeeds, sleeping each time, and then looping again until the time becomes nonzero, also sleeping each time.

For System V semaphores, `SimpleSemClose` doesn't do much:

```
bool SimpleSemClose(struct SimpleSem *sem)
{
    free(sem);
    return true;
}
```

To remove a simple semaphore, you call `SimpleSemRemove`:

SimpleSemRemove—remove simple semaphore

```
#include "SimpleSem.h"

bool SimpleSemRemove(
    struct SimpleSem *sem      /* semaphore */
);
/* Returns true on success or false on error (sets errno) */
```

It's OK to call `SimpleSemRemove` on a semaphore that doesn't exist—that's not considered an error. In fact, that's exactly what you should do when an application starts up and you want to start with a fresh semaphore.

Here's the code for `SimpleSemRemove`:

```
bool SimpleSemRemove(const char *name)
{
    key_t key;
    int semid;
```

```
    if ((key = ftok(name, 1)) == -1) {
        if (errno != ENOENT)
            EC_FAIL
    }
    else {
        if ((semid = semget(key, 1, PERM_FILE)) == -1) {
            if (errno != ENOENT)
                EC_FAIL
        }
        else
            ec_neg1( semctl(semid, 0, IPC_RMID) )
    }
    return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}
```

Now, back to System V. We're ready to introduce the scary `semop` system call, which does both *semwait* and *sempost* operations:

semop—operate on semaphore set

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

int semop(
     int semid,                 /* identifier */
     struct sembuf *sops,       /* operations */
     size_t nsops               /* number of operations */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

struct sembuf—structure for semop

```
struct sembuf {
    unsigned short sem_num;    /* semaphore number */
    short sem_op;              /* Semaphore operation */
    short sem_flg;             /* Operation flags */
};
```

As I said, `semop` doesn't just operate on one semaphore but on as many as you like, even the whole set. You need to build an array of operations (`struct sembufs`) before you make the call, although if there's only one to be operated on, a single `struct sembuf` will do—you just pass its address and use 1 for `nsops`.

Each `sem_op` can be positive, negative, or zero:

> 0 The value of `sem_op` is added to the semaphore value (*sempost*).

< 0 The absolute value of `sem_op` is subtracted from the value, unless that would make the value go negative, in which case the call blocks until the entire value can be subtracted (*semwait*).

0 The call blocks until the value becomes zero.

All of the operations passed to `semop` are performed atomically, and the function doesn't return until everything is done. (Unless it's interrupted by thread cancellation, a signal, or the removal of the semaphore set.)

You can prevent blocking by setting the `IPC_NOWAIT` flag in the `sem_flg` member for an operation. If any operation in the array would block and has the flag set, `semop` returns immediately. Because `semop` always acts atomically, no other operations are done, even if they appeared earlier in the array and even if they wouldn't have blocked.

There's one more feature: For every semaphore that a process increments or decrements, an adjustment is kept, along with the actual value. When the `IPC_UNDO` flag is set for an incrementing operation, the adjustment is decremented, and vice versa for a decrementing operation. When a process exits, its adjustment is added to the semaphore's value, thus undoing whatever the process did to the semaphore. For example, suppose a process decrements a semaphore (i.e., *semwait*) to lock a buffer, and increments it (*sempost*) when it's done. With the `IPC_UNDO` flag, it can ensure that the buffer is unlocked if it should exit abnormally.

For our simple semaphores, we only want to operate on one at a time, we only increment or decrement by 1, and we don't use `IPC_NOWAIT` or `IPC_UNDO`. So `SimpleSemWait` and `SimpleSemPost` are, well, simple:

SimpleSemWait—decrement simple semaphore

```
#include "SimpleSem.h"

bool SimpleSemWait(
    struct SimpleSem *sem      /* semaphore */
);
/* Returns true on success or false on error (sets errno) */
```

SimpleSemPost—increment simple semaphore

```
#include "SimpleSem.h"

bool SimpleSemPost(
    struct SimpleSem *sem      /* semaphore */
);
/* Returns true on success or false on error (sets errno) */
```

Here's the code:

```
bool SimpleSemWait(struct SimpleSem *sem)
{
    struct sembuf sop;

    sop.sem_num = 0;
    sop.sem_op = -1;
    sop.sem_flg = 0;
    ec_neg1( semop(sem->sm.sm_semid, &sop, 1) )
    return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}

bool SimpleSemPost(struct SimpleSem *sem)
{
    struct sembuf sop;

    sop.sem_num = 0;
    sop.sem_op = 1;
    sop.sem_flg = 0;
    ec_neg1( semop(sem->sm.sm_semid, &sop, 1) )
    return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}
```

I should mention one additional use of System V semaphores before we move on: to pass an integer between processes. Suppose, for example, a client wants to pass its process ID to a server. It gets access to a semaphore and sets its value to its process ID. Then the server can look at the value to get the number. Since a semaphore set can have an array of semaphores, you can pass an array of integers this way.

7.10 POSIX Semaphores

These semaphore system calls are part of the POSIX standard, though optional, and you have to check the `_POSIX_SEMAPHORES` feature-test macro to tell whether they're present (see Section 1.5.4 and Section 7.6.3). As of now, they're not in FreeBSD, Darwin, or Linux.

7.10.1 Named POSIX Semaphores

POSIX semaphores are much simpler and much easier to use than System V semaphores. In fact, five of the system calls line right up with the SimpleSem interface from the previous section:

sem_open—open named semaphore

```
#include <semaphore.h>

sem_t *sem_open(
    const char *name,      /* POSIX IPC name */
    int flags              /* flags (excluding O_CREAT) */
);
/* Returns pointer to semaphore or SEM_FAILED on error (sets errno) */

sem_t *sem_open(
    const char *name,      /* POSIX IPC name */
    int flags,             /* flags (including O_CREAT) */
    mode_t perms,          /* permissions */
    unsigned value         /* initial value */
);
/* Returns pointer to semaphore or SEM_FAILED on error (sets errno) */
```

sem_close—close named semaphore

```
#include <semaphore.h>

int sem_close(
    sem_t *sem             /* semaphore */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

sem_unlink—remove named semaphore

```
#include <semaphore.h>

int sem_unlink(
    const char *name       /* POSIX IPC name */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

sem_wait—decrement semaphore

```
#include <semaphore.h>

int sem_wait(
    sem_t *sem             /* semaphore */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

sem_post—increment semaphore

```
#include <semaphore.h>

int sem_post(
     sem_t *sem/* semaphore */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

POSIX semaphores are counting semaphores, like the System V variants, but in steps of one only. Each `sem_t` object represents only one semaphore, not a set of them, as with System V.

The calls for opening, closing, and unlinking follow the pattern for POSIX IPC calls that we saw in Section 7.7.1 for POSIX message-queue system calls. In particular, the name passed to `sem_open` has to follow the portability rules that were discussed in Section 7.6.2. As you would expect, `sem_post` adds one to the semaphore's value, and `sem_wait` subtracts one, blocking if the value is already zero.

You don't use `O_RDONLY`, `O_WRONLY`, or `O_RDWR` with `sem_open`, as they're meaningless—the semaphore is useless unless `sem_post` and `sem_wait` can both be used.

All the initialization rigmarole[10] that we needed for System V is gone. We just pass the value we want (often zero) to `sem_open`.

Be careful about the return value from `sem_open` when it fails: It's `SEM_FAILED`, not `NULL`, which is the usual failure return from functions that otherwise return a pointer.

The implementation of the SimpleSem calls for POSIX semaphores is trivial:

```
struct SimpleSem *SimpleSemOpen(const char *name)
{
    struct SimpleSem *sem = NULL;

    ec_null( sem = malloc(sizeof(struct SimpleSem)) )
    if ((sem->sm.sm_sem = sem_open(name, O_CREAT, PERM_FILE, 0)) ==
      SEM_FAILED)
        EC_FAIL
    return sem;
```

10. If you're not familiar with this term, one dictionary I consulted defined it as "a complicated, petty set of procedures," which is just perfect.

```
EC_CLEANUP_BGN
    free(sem);
    return NULL;
EC_CLEANUP_END
}

bool SimpleSemClose(struct SimpleSem *sem)
{
    ec_neg1( sem_close(sem->sm.sm_sem) )
    free(sem);
    return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}

bool SimpleSemRemove(const char *name)
{
    if (Sem_unlink(name) == -1 && errno != ENOENT)
        EC_FAIL
    return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}

bool SimpleSemPost(struct SimpleSem *sem)
{
    ec_neg1( sem_post(sem->sm.sm_sem) )
    return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}

bool SimpleSemWait(struct SimpleSem *sem)
{
    ec_neg1( sem_wait(sem->sm.sm_sem) )
    return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}
```

POSIX semaphores have some additional features that we didn't need for the SimpleSem interface. First of all, as with System V, you can query the value of a semaphore without modifying it or waiting:

sem_getvalue—get value of semaphore

```
#include <semaphore.h>

int sem_getvalue(
    sem_t *restrict sem,       /* semaphore */
    int *valuep                /* returned value */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

If the value of the semaphore is greater than zero during the call to `sem_getvalue`, that value is returned. It could be something different, however, by the time `sem_getvalue` returns, so the actual number isn't that useful. If the value is zero, the value returned is the negative of the number of processes waiting on the semaphore. If there aren't any, zero is returned.

There are two variations on `sem_wait`: One, `sem_trywait`, is nonblocking:

sem_trywait—decrement semaphore if possible

```
#include <semaphore.h>

int sem_trywait(
    sem_t *sem                 /* semaphore */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

`sem_trywait` returns –1 with `errno` set to `EAGAIN` if the semaphore is already zero. (Other system calls use a flag like `O_NONBLOCK` or `IPC_NOWAIT` for nonblocking; here it's a separate system call.)

The other `sem_timedwait` variant times out after an interval if the semaphore doesn't become positive:

sem_timedwait—decrement semaphore

```
#include <semaphore.h>
#include <time.h>

int sem_timedwait(
    sem_t *restrict sem,          /* semaphore */
    const struct timespec *time   /* absolute time */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

The time passed to `sem_timedwait` is an absolute time (e.g., 1:23:17 PM), not a time interval (e.g., 27 secs.). What clock it's compared to and with what resolution depends on whether the POSIX Timers option is supported. If so, it's the real-time clock. If not, the ordinary clock (as used by the `time` system call) is used. (See Section 1.7 for a discussion of `struct timespec` and other aspects of UNIX time.)

`sem_timedwait` is part of the Timeouts option (`_POSIX_TIMEOUTS`) and is new with SUS3.

7.10.2 Unnamed POSIX Semaphores

Quick review: `sem_open`, `sem_close`, and `sem_unlink` are used with named semaphores that exist external to a process somewhere and are accessed by a POSIX IPC name. `sem_open` gets you a pointer to a `sem_t` object, which you never deal with directly. Whatever memory it uses is freed when you call `sem_close`. These semaphores inherently work between threads and processes.

To make semaphores faster, you can also declare a `sem_t` object directly:

```
sem_t sem;
```

or even allocate one dynamically, like this:

```
sem_t *semp = malloc(sizeof(sem_t));
```

But if you allocate the `sem_t` object yourself, you have to call `sem_init` to initialize it:

sem_init—initialize unnamed semaphore

```
#include <semaphore.h>

int sem_init(
     sem_t *sem,            /* semaphore */
     int pshared,           /* shared between processes? */
     unsigned value         /* initial value */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

and then you call `sem_destroy` to destroy it:

sem_destroy—destroy unnamed semaphore

```
#include <semaphore.h>

int sem_destroy(
     sem_t *sem         /* semaphore */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

`sem_destroy` doesn't deallocate the `sem_t` object, as it knows nothing about how the memory got there (static declaration, `malloc`, etc.). You have to free the memory (if that's appropriate) yourself. If the semaphore was allocated globally or on the stack, of course, you don't have to do anything; its life will end at the appropriate time.

`sem_init` and `sem_destroy` are substitutes for `sem_open` and `sem_close`; you use one pair or the other, depending on whether the semaphore is named or not. The other calls (e.g., `sem_post`, `sem_timedwait`) don't care how you got the `sem_t` pointer.

OK, unnamed semaphores may be fast, but if they're in the memory of a single process, what good are they? They're real good:

- They work fine for synchronizing between threads when you want a counting semaphore, not just a mutex that has only two states (binary semaphore).
- They work fine between processes if they're in shared memory, which we'll get to shortly. In this case the `pshared` argument to `sem_init` has to be nonzero.

The first use is important for threads, and some UNIX implementations, notably Linux and FreeBSD (but not Darwin), support it even though some versions support POSIX semaphores completely. That is, `sem_init` (`pshared` zero only) and `sem_destroy` are supported but not `sem_open` and `sem_close`. In these limited implementations `sem_post`, `sem_wait`, `sem_trywait`, and `sem_getvalue` are also supported but not `sem_timedwait`.

One more rule about unnamed (in-memory) semaphores: Only the actual memory passed to `sem_init` can be used, not a copy of it. So the following is wrong:

```
void fcn(sem_t s)
{
    sem_post(&s);
    ...
}

void fcn2(void)
{
    sem_t sem;

    sem_init(&sem);
    fcn(sem);
    ...
}
```

Calling `fcn` copies the semaphore, which violates the rule. Rather, all parts of the program that manipulate the semaphore need to work with the address of the originally initialized storage. I'm going to provide an example of an unnamed semaphore located in shared memory in Section 7.14.2.

7.10.3 System V and POSIX Semaphores Critiqued

For interprocess communication, the System V semaphore system calls are ridiculously difficult to use and overwrought with features, but they're universally available and not so bad once the hard stuff (initialization, mainly) is hidden behind a reasonable interface like SimpleSem. POSIX semaphores are much easier to use but not universally available. If portability is important to you, you need to use the System V system calls, and there's nothing to be gained by using the POSIX calls anywhere if you can't use them everywhere. If portability isn't important, and the POSIX calls are supported on the UNIX systems of interest, then clearly those are the ones to use.[11]

For intraprocess communication—that is, between threads—the System V semaphores are much too cumbersome and slow, and unnamed, nonprocess-shared POSIX semaphores are usually available wherever POSIX Threads are, so those are the ones to use if you need something beyond mutexes.

7.10.4 Process-Shared Mutexes and Read-Write Locks

I introduced mutexes in Section 5.17.3 for use in synchronizing threads, but they were in-memory and within a process—essentially, in-memory, nonprocess-shared POSIX semaphores used in a binary fashion. You can also create an in-memory mutex and initialize it with `pthread_mutex_init` with the `PTHREAD_PROCESS_SHARED` attribute set, in which case it's shared between threads that may be in different processes. However, this feature isn't always implemented even if POSIX Threads are (it's a separate suboption), so it may not be available.

11. Be careful about assuming that portability isn't important. Almost every application nowadays is a candidate for someday being ported to Linux. And potential portability, even if it never happens, may be helpful in negotiations with vendors.

There are also POSIX Thread read-write locks, which I didn't describe in Chapter 5 at all. These are like mutexes, but they distinguish between reading and writing (share and exclusive), to allow more throughput. As with mutexes, there's a `PTHREAD_PROCESS_SHARED` attribute you can set. (As we'll see in the next section, read-write locks are to mutexes as `fcntl` file-locking is to `lockf` file-locking [Sections. 7.11.3 and 7.11.4.)]

7.11 File Locking

This section explains how file locking works in UNIX and how it sometimes doesn't work the way you need it to.

7.11.1 A Bad Example

We like to do things wrong at first—it provides some motivation for getting it right.

To motivate our discussion of file locking, let's build an example application that has one process building a file while another is reading it. The file is a linked list of records, each of which holds an integer of data and the offset of the next record. The linked list is sorted, but the physical position of the records is in their order of creation. The structure for each record is:

```
struct rec {
    int r_data;
    off_t r_next;
};
```

If we insert records with data of 1000, 999, and 998, in that order, the file looks like Figure 7.2. The first record is just a header that tells where the list starts.

The example's `main` program starts a child process (`process1`) to build the list while the parent (`process2`) traverses it repeatedly to check if it's properly formed:

```
int main(void)
{
    pid_t pid;
```

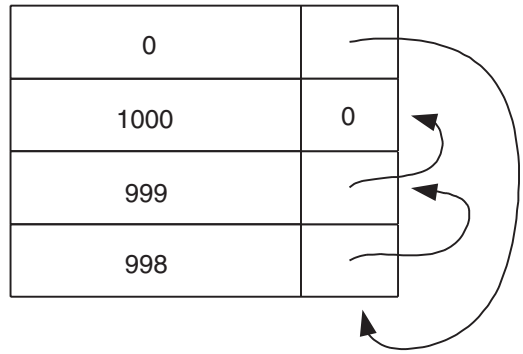

Figure 7.2 Linked list of records.

```
    ec_neg1( pid = fork() )
    if (pid == 0)
        process1();
    else {
        process2();
        ec_neg1( waitpid(pid, NULL, 0) )
    }
    exit(EXIT_SUCCESS);

EC_CLEANUP_BGN
    exit(EXIT_FAILURE);
EC_CLEANUP_END
}
```

For data, `process1` just counts backwards from 1000 after writing the header record:

```
#define DBNAME "termdb"

static void process1(void)
{
    int dbfd, data;
    struct rec r;

    ec_neg1( dbfd = open(DBNAME, O_CREAT | O_TRUNC | O_RDWR, PERM_FILE) )
    memset(&r, 0, sizeof(r));
    ec_false( writerec(dbfd, &r, 0) )
    for (data = 100; data >= 0; data--)
        ec_false( store(dbfd, data) )
    ec_neg1( close(dbfd) )
    exit(EXIT_SUCCESS);

EC_CLEANUP_BGN
    exit(EXIT_FAILURE);
```

```
EC_CLEANUP_END
}
```

To make the I/O easy, there are two functions: `writerec` writes a record at a specified offset, and `readrec` reads a record from an offset. They both consider partial writes or reads or an end-of-file to be errors:

```
bool readrec(int dbfd, struct rec *r, off_t off)
{
    ssize_t nread;

    if ((nread = pread(dbfd, r, sizeof(struct rec), off)) ==
      sizeof(struct rec))
        return true;
    if (nread != -1)
        errno = EIO;
    EC_FAIL
    return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}

bool writerec(int dbfd, struct rec *r, off_t off)
{
    ssize_t nwrote;

    if ((nwrote = pwrite(dbfd, r, sizeof(struct rec), off)) ==
      sizeof(struct rec))
        return true;
    if (nwrote != -1)
        errno = EIO;
    EC_FAIL
    return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}
```

We could have used `pwrite` and `pread` directly almost as easily, but the error checking is a bit convoluted so it makes sense to encapsulate them.

The function `store` (called from `process1`) is responsible for keeping the linked list in order. Note that it doesn't try to minimize I/O:

```
bool store(int dbfd, int data)
{
```

```
    struct rec r, rnew;
    off_t end, prev;

    ec_neg1( end = lseek(dbfd, 0, SEEK_END) )
    prev = 0;
    ec_false( readrec(dbfd, &r, prev) )
    while (r.r_next != 0) {
        ec_false( readrec(dbfd, &r, r.r_next) )
        if (r.r_data > data)
            break;
        prev = r.r_next;
    }
    ec_false( readrec(dbfd, &r, prev) )
    rnew.r_next = r.r_next;
    r.r_next = end;
    ec_false( writerec(dbfd, &r, prev) )
    rnew.r_data = data;
    usleep(1); /* give up CPU */
    ec_false( writerec(dbfd, &rnew, end) )
    return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}
```

Spend a bit of time with `store` and it should become clear. Note that it handles the cases of the new record going at the beginning or at the end properly. The call to `usleep` (sleeps for 1 microsecond) near the end is just to give up the CPU to let other processes run a bit. This happens naturally in complicated programs, but I had to force it in this simple one because I want the other process, which I'm about to show, to run concurrently.

Here's `process2` which checks the integrity of the file:

```
static void process2(void)
{
    int try, dbfd;
    struct rec r1, r2;

    for (try = 0; try < 10; try++)
        if ((dbfd = open(DBNAME, O_RDWR)) == -1) {
            if (errno == ENOENT) {
                continue;
            }
            else
                EC_FAIL
        }
    ec_neg1( dbfd )
```

```
    for (try = 0; try < 100; try++) {
        ec_false( readrec(dbfd, &r1, 0) )
        while (r1.r_next != 0) {
            ec_false( readrec(dbfd, &r2, r1.r_next) )
            if (r1.r_data > r2.r_data) {
                printf("Found sorting error (try %d)\n", try);
                break;
            }
            r1 = r2;
        }
    }
    ec_neg1( close(dbfd) )
    return;

EC_CLEANUP_BGN
    exit(EXIT_FAILURE);
EC_CLEANUP_END
}
```

I try a few times to open the database because it may take `process1` a while to get scheduled and to complete its call to `open`. Then I repeatedly (100 times) go through the list looking for errors, such as a missing record or out-of-order data.

Sure enough, here's what I got when I ran the program:

```
ERROR:  0: process2 [/aup/c7/f1.c:107] readrec(dbfd, &r2, r1.r_next)
        1: readrec [/aup/c7/f1.c:17] 0
                *** EIO (5: "I/O error") ***
```

This particular error resulted because a new record that `store` was about to write to the end of the database hadn't gotten written when `process2` looked for it.

Now we have an application that doesn't work, which is no great accomplishment, but ours doesn't work specifically because two processes are accessing the same file and one of them is finding inconsistent data. They need some coordination.

So, are you motivated to see what this section is about?

7.11.2 Using a Semaphore as a File Lock

The obvious way to fix the example from the previous section is to use a semaphore to prevent `process2` from seeing an inconsistent file. Using the SimpleSem interface from Section 7.9.2, we can define a global like this:

```
static struct SimpleSem *sem;
```

Each process independently opens the semaphore with a line like this:

```
ec_null( sem = SimpleSemOpen("sem") )
```

In addition, `process1` initially increments the semaphore to indicate that the database is consistent:

```
ec_false( SimpleSemPost(sem) )
```

The critical lines in `store` are the ones that update a record in place and store a new record at the end:

```
ec_false( SimpleSemWait(sem) )
ec_false( readrec(dbfd, &r, prev) )
rnew.r_next = r.r_next;
r.r_next = end;
ec_false( writerec(dbfd, &r, prev) )
rnew.r_data = data;
usleep(1); /* give up CPU */
ec_false( writerec(dbfd, &rnew, end) )
ec_false( SimpleSemPost(sem) )
```

And the critical lines in `process2` are the ones that traverse the list:

```
for (try = 0; try < 100; try++) {
    ec_false( SimpleSemWait(sem) )
    ec_false( readrec(dbfd, &r1, 0) )
    while (r1.r_next != 0) {
        ec_false( readrec(dbfd, &r2, r1.r_next) )
        if (r1.r_data > r2.r_data) {
            printf("Found sorting error (try %d)\n", try);
            break;
        }
        r1 = r2;
    }
    ec_false( SimpleSemPost(sem) )
}
```

With these changes, the application works perfectly. (Except on FreeBSD and Darwin, where System V semaphores are incorrectly implemented, as explained in Section 7.9.1.)

The chief problem with using a semaphore as a file lock is that for an arbitrary file it's not clear what the semaphore name should be. This isn't a problem for an application with a few fixed file names, but dealing with files on an ad hoc basis would be much too awkward. We would have to invent some facility for mapping file names to semaphore names. To make things worse, sometimes only part of a file needs to be locked, and that would mean different semaphores for different

parts. Finally, managing semaphores—getting them open and closed, and removing them when they get left around—is a pain.

7.11.3 `lockf` System Call

The hassle of using a semaphore to lock a file isn't really a problem because UNIX has a special system call to lock a section of a file:

lockf—lock section of file

```
#include <unistd.h>

int lockf(
     int fd,            /* file descriptor */
     int op,            /* operation */
     off_t len          /* length of section */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

The file descriptor must be opened for writing (`O_WRONLY` or `O_RDWR`).

The section to be locked or unlocked is from the current file offset (set by a `read`, a `write`, or an `lseek`) for `len` bytes either forward or, if `len` is negative, backward. In the case of backward, the byte at the current offset is not part of the section. In the case of forward, bytes can be locked that don't exist because the file hasn't gotten that big yet. If `len` is zero, the section extends from the current offset to the end of the file, even as the file grows. So, to lock the whole file, just make sure the offset is zero and use a length of zero.

If a section to be locked overlaps a section already locked, the two sections are merged. If part of a locked section is unlocked, the locked section is made smaller, and is possibly split into two discontinuous sections.

All locks that a process has on a file are released when *any* file descriptor that that process has open on the file is closed, even if the file descriptor that's closed was obtained independently (i.e., different `open`) from the one that was passed to `lockf`.[12] It follows that any locks are released when a process terminates since

12. From a FreeBSD man page: "This interface follows the completely stupid semantics of System V and IEEE Std 1003.1-1988 ('POSIX.1') that require that all locks associated with a file for a given process are removed when any file descriptor for that file is closed by that process. This semantic means that applications must be aware of any files that a subroutine library may access. For example if an application for updating the password file locks the password file database while making the update, and then calls getpwnam(3) to retrieve a record, the lock will be lost because getpwnam(3) opens, reads, and closes the password database." BSD-based systems have a better call, `flock`, but it's nonstandard.

that causes all file descriptors to be closed. Locks are *not* inherited when a process calls `fork`.

Here are the operations for the `op` argument:

`F_LOCK` Lock the section; block if any part of it is already locked by another process.

`F_TLOCK` Like `F_LOCK`, but return –1 with `errno` set to `EAGAIN` (or `EACCES`) if `F_LOCK` would have blocked.

`F_TEST` Don't lock, but return an error as `F_TLOCK` would if `F_LOCK` would have blocked.

`F_ULOCK` Unlock the section.

You don't have to have a semaphore name or open anything special to use `lockf` on a file, since it takes the same file descriptor you use to access the file. So, we can easily slip it into our example in place of the semaphore calls from the previous section. I'll show just the critical part of `store`:

```
ec_neg1( lseek(dbfd, 0, SEEK_SET) )
ec_neg1( lockf(dbfd, F_LOCK, 0) )
ec_false( readrec(dbfd, &r, prev) )
rnew.r_next = r.r_next;
r.r_next = end;
ec_false( writerec(dbfd, &r, prev) )
rnew.r_data = data;
usleep(1); /* give up CPU */
ec_false( writerec(dbfd, &rnew, end) )
ec_neg1( lseek(dbfd, 0, SEEK_SET) )
ec_neg1( lockf(dbfd, F_ULOCK, 0) )
```

and the critical part of `process2`:

```
for (try = 0; try < 100; try++) {
    ec_neg1( lseek(dbfd, 0, SEEK_SET) )
    ec_neg1( lockf(dbfd, F_LOCK, 0) )
    ec_false( readrec(dbfd, &r1, 0) )
    while (r1.r_next != 0) {
        ec_false( readrec(dbfd, &r2, r1.r_next) )
        if (r1.r_data > r2.r_data) {
            printf("Found sorting error (try %d)\n", try);
            break;
        }
        r1 = r2;
    }
    ec_neg1( lseek(dbfd, 0, SEEK_SET) )
    ec_neg1( lockf(dbfd, F_ULOCK, 0) )
}
```

7.11.4 `fcntl` System Call for File Locking

You can also lock a file with the `fcntl` system call, which we first encountered in Section 3.8.3. Its locking functionality is a superset of `lockf`'s. Here's a recap of the `fcntl` synopses:

fcntl—control open file

```
#include <unistd.h>
#include <fcntl.h>

int fcntl(
    int fd,             /* file descriptor */
    int op,             /* operation */
    ...                 /* optional argument depending on op */
);
/* Returns result depending on op or -1 on error (sets errno) */
```

There are three `fcntl` operations for locking that operate on a structure, a pointer to which is passed as the third argument:

struct flock—structure for fcntl file locking

```
struct flock {
    short l_type;       /* lock type: F_RDLCK, F_WRLCK, F_UNLCK */
    short l_whence;     /* interpretation of l_start */
    off_t l_start;      /* start of section */
    off_t l_len;        /* length of section */
    pid_t l_pid;        /* process holding lock; used with F_GETLK */
};
```

Three members of the structure, `l_whence`, `l_start`, and `l_len`, establish the section to be operated on. The first two are like the arguments to `lseek` (`l_whence` can be `SEEK_SET`, `SEEK_CUR`, or `SEEK_END`), and `l_start` is absolute, relative to the current file offset, or relative to the end of the file. The length of the section is given by `l_len`.

With `lockf` there is only one type of lock, but with `fcntl` you can have both read locks and write locks, or, as they're commonly called, *share* locks and *exclu-*

sive locks. A share lock on a section prevents an exclusive lock on that section; an exclusive lock prevents a share or exclusive lock. In practice, you set a share lock when you will only be reading data, and an exclusive lock when you will be writing it.

You specify the lock type in the `l_type` member; the third choice shown in the synopsis is for unlocking a section.

Now we're ready to explain the `fcntl` operations for locking, which are pretty simple:

`F_SETLK` Perform the operation specified in the structure, if possible. If a lock can't be set immediately, return –1 with `errno` set to `EAGAIN` or `EACCES`; that is, do not block.

`F_SETLKW` Just like `F_SETLK`, but block if the lock can't be set immediately.

`F_GETLK` Return information about the first lock that would cause the lock specified in the structure to block, if any. All members of the passed-in structure are overwritten with the results, including the process ID of the process holding the lock. If the lock passed in would not block, the structure is passed back unchanged except that the first member is changed to `F_UNLCK`.

Thus, `fcntl`-locking can do everything `lockf` can. In addition, it distinguishes between share and exclusive locks, and it can retrieve information about existing locks. Usually, implementations implement `lockf` as a library function on top of `fcntl`, but that's not required. Also, [SUS2002] says that you shouldn't assume that the locks manipulated by the two functions are the same. That is, if you lock with `lockf`, don't unlock with `fcntl`. In fact, don't expect `fcntl` to even know about the lock.

As with `lockf` locks, locks set with `fcntl` are released when a file descriptor open to the file is closed or when the process terminates, and they are not inherited when a process calls `fork`.

7.11.5 Advisory and Mandatory Locks

The locks set with `lockf` and `fcntl` normally affect only those function calls, not other I/O operations. That is, if you set a lock on a file with `lockf` and then another process writes on the file without calling `lockf`, that write will proceed. This is called *advisory* locking, and obviously it works only if all the processes cooperate by calling `lockf` or `fcntl` appropriately.

Mandatory locking means that once a lock is set, it really does prohibit a conflicting I/O operation. The POSIX and SUS standards don't specify mandatory locking at all, but they don't prohibit it either.

On those systems that support mandatory locking, you don't do anything differently in the calls to `lockf` or `fcntl`. Instead you mark the file itself with a set of permissions that otherwise make no sense: set-group-ID-on-execution bit on, and group-execute bit off. Here's an example program that uses a lock that's advisory or mandatory, depending on the argument:

```
int main(int argc, char *argv[])
{
    int fd;
    mode_t perms = PERM_FILE;

    if (fork() == 0) {
        sleep(1); /* wait for parent */
        ec_neg1( fd = open("tmpfile", O_WRONLY | O_NONBLOCK) )
        ec_neg1( write(fd, "x", 1) )
        printf("child wrote OK\n");
    }
    else {
        (void)unlink("tmpfile");
        if (argc == 2)
            perms |= S_ISGID; /* mandatory locking */
        ec_neg1( fd = open("tmpfile", O_CREAT | O_RDWR, perms) )
        ec_neg1( lockf(fd, F_LOCK, 0) )
        printf("parent has lock\n");
        ec_neg1( wait(NULL) )
    }
    exit(EXIT_SUCCESS);

EC_CLEANUP_BGN
    exit(EXIT_FAILURE);
EC_CLEANUP_END
}
```

Here's what I got when I ran this on Solaris, which has mandatory locking:

```
$ lockftest
parent has lock
child wrote OK
$ lockftest x
parent has lock
ERROR:  0: main [/aup/c7/lockftest.c:11] write(fd, "x", 1)
               *** EAGAIN (11: "Resource temporarily unavailable") ***
$
```

Without the `O_NONBLOCK` flag, the write in the child process would have blocked waiting for the lock to be released.

7.11.6 High-Performance Database Locking

File locking as provided for by the `fcntl` and `lockf` system calls, even with mandatory locking, isn't suitable for high-performance databases because there's too much overhead in manipulating the locks, and it's too hard to implement sophisticated deadlock detection and correction algorithms. That's usually OK, however, because big database systems run in their own processes anyway, so they act as a gatekeeper to the database files. Locks can easily be kept in the address space of the database process or in memory that's shared among multiple database processes. Done this way, there's no need at all to use system calls to manage the locks.

7.12 About Shared Memory

Recall from Section 5.17 that threads within a process share all static data—global data and static data that's internal to a function. Processes, on the other hand, have entirely separate memory, even if a `fork` was executed without an `exec`, in which case the child gets a copy of the parent's address space.

With shared memory, you can arrange for separate processes to have some memory in common. As with threads, they usually need to use mutexes or semaphores to coordinate their access to the memory.

There are both System V and POSIX versions of shared memory, just as with messages and semaphores. For both mechanisms, each process "opens" a shared-memory segment and gets a pointer to the memory, which it is then free to use with ordinary C or C++ operators, without using a system call. Normally, each process's pointer has a different value, meaningful only within that process, but

the underlying memory is the same. As we did before, we'll start with System V shared memory and then move on to POSIX.

7.13 System V Shared Memory

By now you should be familiar with how the System V IPC calls work. As Section 7.4 explained, you get a key, with `ftok` if you like, and use an *X*`get` call (`shmget`) to get an identifier. You control it with an *X*`ctl` call (`shmctl`). In the case of shared memory, you attach it to your process with `shmat`, which returns a pointer, and detach it with `shmdt` when you're done with it.

7.13.1 System V Shared-Memory System Calls

Here are the synopses for the System V shared-memory system calls:

shmget—get shared memory segment

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

int shmget(
     key_t key,                 /* key */
     size_t size,               /* size of segment */
     int flags                  /* creation flags */
);
/* Returns shared-memory identifier or -1 on error (sets errno) */
```

shmctl—control shared memory segment

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

int shmctl(
     int shmid,                 /* identifier */
     int cmd,                   /* command */
     struct shmid_ds *data      /* data for command */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

struct shmid_ds—structure for shmctl

```
struct msqid_ds {
     struct ipc_perm shm_perm;  /* permission structure */
     size_t shm_segsz;          /* size of segment in bytes */
     pid_t shm_lpid;            /* process ID of last shared memory op */
     pid_t shm_cpid;            /* process ID of creator */
     shmatt_t shm_nattch;       /* number of current attaches */
     time_t shm_atime;          /* time of last shmat */
     time_t shm_dtime;          /* time of last shmdt */
     time_t shm_ctime;          /* time of last change by shmctl */
};
```

shmat—attach shared memory segment

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

void *shmat(
     int shmid,             /* identifier */
     const void *shmaddr,   /* desired address or NULL */
     int flags              /* attachment flags */
);
/* Returns pointer or -1 on error (sets errno) */
```

shmdt—detach shared memory segment

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

int shmdt(
     const void *shmaddr    /* pointer to segment */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

As you'd expect, `shmget` accesses a shared memory segment, creating it if necessary and if the `IPC_CREAT` or `IPC_PRIVATE` flags are present. The `size` argument is meaningful only if the segment is created. A newly created segment is initialized to zeros.

Then, to use the segment, you call `shmat`, which gives you a pointer to it. Oddly, `shmat` returns –1 on an error, not `NULL`. This works because it will never return a pointer that can be mistaken for –1; in fact, returned pointers are even numbers on essentially all UNIX systems.

When you're done with the segment, you call `shmdt`, passing in the pointer you got from `shmat`, not the identifier. The segment stays around until it's explicitly removed (via `shmctl` or the `ipcrm` command) or until the machine is rebooted, so you're free to reattach it. You might get a different pointer, though.

Normally, you don't care what address `shmat` gives you, so you set the second argument to `NULL`. But, if you want, you can try to force it to give you the address you specify with `shmaddr`. If it can't, because that address is already in use or otherwise invalid, it fails with `errno` set to `EINVAL`. There's also a flag, `SHM_RND`, for rounding the address to a proper boundary, but I won't describe the details. Another flag, `SHM_RDONLY`, if set, attaches the segment read-only.

With `shmctl` you use the same commands as with the other *X*`ctl` system calls, `IPC_STAT`, `IPC_SET`, and `IPC_RMID`. `IPC_SET` sets `shm_perm.uid`, `shm_perm.gid`, and the low-order 9 bits of `shm_perm.mode`.

Here's a simple program that shows how a segment (a small one!) can be shared between two processes:

```
static int *getaddr(void)
{
    key_t key;
    int shmid, *p;

    (void)close(open("shmseg", O_WRONLY | O_CREAT, 0));
    ec_neg1( key = ftok("shmseg", 1) )
    ec_neg1( shmid = shmget(key, sizeof(int), IPC_CREAT | PERM_FILE) )
    ec_neg1( p = shmat(shmid, NULL, 0) )
    return p;

EC_CLEANUP_BGN
    return NULL;
EC_CLEANUP_END
}

int main(void)
{
    pid_t pid;

    if ((pid = fork()) == 0) {
        int *p, prev = 0;

        ec_null( p = getaddr() )
        while (*p != 99)
            if (prev != *p) {
                printf("child saw %d\n", *p);
                prev = *p;
            }
        printf("child is done\n");
    }
    else {
        int *p;

        ec_null( p = getaddr() )
        for (*p = 1; *p < 4; (*p)++)
            sleep(1);
        *p = 99;
    }
    exit(EXIT_SUCCESS);

EC_CLEANUP_BGN
    exit(EXIT_FAILURE);
EC_CLEANUP_END
}
```

Each process calls `getaddr`; one of them creates the segment, and the other just accesses it. Here's the output we got:

```
$ shmex
child saw 1
child saw 2
child saw 3
$ child saw 99
child is done
```

A little strange, no? Why did the child report that it saw 99 when that's what terminates its `while` loop? Well, in the line

```
while (*p != 99)
```

`*p` was equal to 3 at one point, but by the time it got to the line

```
printf("child saw %d\n", *p);
```

`*p` was already 99. (The `$` prompt is where it is because the parent didn't wait for the child before exiting. That's not a defect but a pretty common occurrence in UNIX.)

Things get even stranger if you run `shmex` a second time:

```
$ shmex
child is done
$
```

It broke! No, actually all that's wrong is that the segment was left there—unattached—after `shmex` terminated the first time, and it still held the value 99. In the second execution, the child saw 99 before the parent even got to its `for` loop. Obviously, we could fix this problem by removing the segment when `shmex` terminates. But you get the point: Sharing memory is tricky!

7.13.2 Shared Memory and Semaphores

The example in the previous section is wrong for another, more subtle reason: As we saw with threads (Section 5.17.3), we can't assume that references to `*p` are atomic. Generally, you can't share memory between processes without some control in the form of a semaphore. Therefore, let's add that in (`getaddr` is unchanged):

```
int main(void)
{
    pid_t pid;

    ec_false( SimpleSemRemove("shmexsem") )
    if ((pid = fork()) == 0) {
        struct SimpleSem *sem;
        int *p, prev = 0, n;

        ec_null( sem = SimpleSemOpen("shmexsem") )
        ec_null( p = getaddr() )
        while (true) {
            ec_false( SimpleSemWait(sem) )
            n = *p;
            ec_false( SimpleSemPost(sem) )
            if (n == 99)
                break;
            if (prev != n) {
                printf("child saw %d\n", n);
                prev = n;
            }
        }
        printf("child is done\n");
        ec_false( SimpleSemClose(sem) )
    }
    else {
        struct SimpleSem *sem;
        int *p, i;

        ec_null( sem = SimpleSemOpen("shmexsem") )
        ec_null( p = getaddr() )
        *p = 0;
        ec_false( SimpleSemPost(sem) )
        for (i = 1; i < 4; i++) {
            ec_false( SimpleSemWait(sem) )
            *p = i;
            ec_false( SimpleSemPost(sem) )
            sleep(1);
        }
        ec_false( SimpleSemWait(sem) )
        *p = 99;
        ec_false( SimpleSemPost(sem) )
        ec_false( SimpleSemClose(sem) )
    }
    exit(EXIT_SUCCESS);

EC_CLEANUP_BGN
    exit(EXIT_FAILURE);
EC_CLEANUP_END
}
```

Here's the output now—it's much more sensible:

```
$ shmex2
child saw 1
child saw 2
child saw 3
$ child is done
shmex2
child saw 1
child saw 2
child saw 3
$ child is done
```

Look at all the changes we needed to put in the protection:

- The child assigned `*p` to local memory with the semaphore locked and then was free to use the local memory with the semaphore unlocked.
- Similarly, the parent used a local variable in the `for` loop, locking the semaphore only to access the shared memory.
- Initially, the semaphore is locked (zero value), so the parent is free to initialize the shared memory to zero. Then it calls `SimpleSemPost` to get things moving. It's OK if the child accesses the shared memory at that point. This version then can be run repeatedly since it initializes the segment each time it's run.
- We remove the semaphore at the start of each run so that it will start with zero.

Although we've fixed the atomicity problems, the child process is still inefficient. To see why, look again at its loop:

```
while (true) {
    ec_false( SimpleSemWait(sem) )
    n = *p;
    ec_false( SimpleSemPost(sem) )
    if (n == 99)
        break;
    if (prev != n) {
        printf("child saw %d\n", n);
        prev = n;
    }
}
```

It keeps racing around and around, locking and unlocking the semaphore, but only does something (prints) when the value changes. Most of its efforts are wasted, and all the locking keeps the semaphore unavailable for no reason. Wouldn't it be better for the parent to just tell the child when the value changed? (This is reminiscent of the motivation for condition variables in Section 5.17.4.)

We can fix the program with two semaphores instead of one: A semaphore *W* that must be locked to write to the shared memory, and a semaphore *R* for reading it. The parent waits on *W* before writing and then posts *R* when it's done. The child waits on *R* before reading and then posts *W* when it's done. To get things started, *W* is initialized to 1 (posted), and *R* is left at zero. Here's the revised code (`getaddr` is still unchanged):

```
int main(void)
{
    pid_t pid;

    ec_false( SimpleSemRemove("shmexsem") )
    if ((pid = fork()) == 0) {
        struct SimpleSem *semR, *semW;
        int *p, n;

        ec_null( semR = SimpleSemOpen("shmexsemR") )
        ec_null( semW = SimpleSemOpen("shmexsemW") )
        ec_null( p = getaddr() )
        while (true) {
            ec_false( SimpleSemWait(semR) )
            n = *p;
            ec_false( SimpleSemPost(semW) )
            if (n == 99)
                break;
            printf("child saw %d\n", n);
        }
        printf("child is done\n");
        ec_false( SimpleSemClose(semR) )
        ec_false( SimpleSemClose(semW) )
    }
    else {
        struct SimpleSem *semR, *semW;
        int *p, i;

        ec_null( semR = SimpleSemOpen("shmexsemR") )
        ec_null( semW = SimpleSemOpen("shmexsemW") )
        ec_null( p = getaddr() )
        *p = 0;
        ec_false( SimpleSemPost(semW) )
        for (i = 1; i < 4; i++) {
            ec_false( SimpleSemWait(semW) )
            *p = i;
            ec_false( SimpleSemPost(semR) )
            sleep(1);
        }
        ec_false( SimpleSemWait(semW) )
        *p = 99;
```

```
            ec_false( SimpleSemPost(semR) )
            ec_false( SimpleSemClose(semR) )
            ec_false( SimpleSemClose(semW) )
        }
        exit(EXIT_SUCCESS);

EC_CLEANUP_BGN
        exit(EXIT_FAILURE);
EC_CLEANUP_END
}
```

Note that the child no longer uses `prev` to tell when the shared memory changed because now it never gets `semR` unless there's a change. This is way more efficient!

7.13.3 System V Shared-Memory Implementation of SMI

Now let's extend the ideas from the previous section and use shared memory and semaphores to implement the SMI functions that we already implemented with FIFOs (Section 7.3.3) and messages (Secs. 7.5.3 and 7.7.2).

The server and each client will use a separate shared-memory segment for incoming messages, along with two semaphores, as I explained in the previous example. So, if there are two clients, there will be three shared-memory segments and six semaphores. With this approach, there isn't a queue of messages—each segment holds one message at a time, and a new one can't be written until the recipient has finished with it and posts the writing semaphore (*W*). This is not the best design because each client can only work as fast as the server can service it, but it will suffice to show how shared memory is used, which is our objective.

Figure 7.3 shows the server and two clients. Shared-memory segment *mem-server* is shared by all three; segment *mem-1* is shared by client1 and the server; segment *mem-2* is shared by client2 and the server. Each of the three processes can access messages located in shared memory without them ever moving.

In the presentation of the SMI implementation for System V shared memory, we're going to assume that you're familiar with the FIFO and message-queue implementations from earlier sections in this chapter, so we won't explain details that have already been explained.

Recall that the SMI functions were specifically designed to allow messages to be processed in-place; the send and receive operations are divided into "getaddr" and "release" functions. The benefit of that wasn't so apparent in the earlier implementations, but it will be in this one.

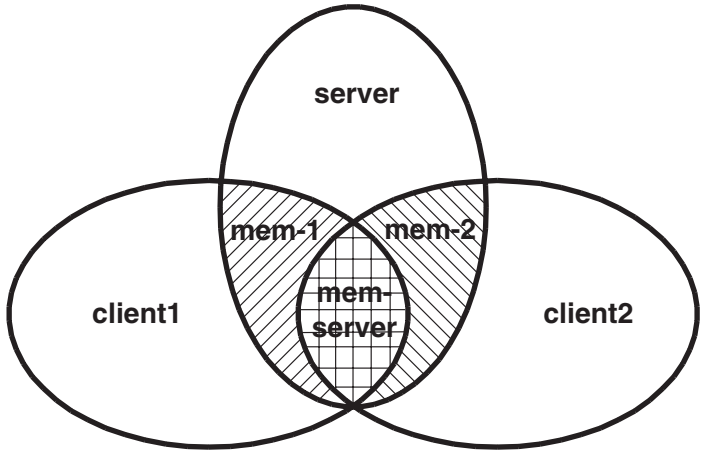

Figure 7.3 Server sharing memory with two clients.

We'll take advantage of one of the features of System V semaphores and use one semaphore set of two semaphores for each shared-memory segment. Semaphore 0 is the read semaphore, and 1 is the write semaphore. Here are some macros for those numbers and for the *semwait* and *sempost* operations on them:

```
#define SEMI_READ    0
#define SEMI_WRITE   1
#define SEMI_POST    1
#define SEMI_WAIT    -1
```

Given a semaphore, the function `op_semi` operates on it. For example, to post the write semaphore for a segment, you execute:

```
ec_neg1( op_semi(semid_receiver, SEMI_WRITE, SEMI_POST) )
```

Here's the code for `op_semi`:

```
static int op_semi(int semid, int sem_num, int sem_op)
{
    struct sembuf sbuf;
    int r;

    sbuf.sem_num = sem_num;
    sbuf.sem_op = sem_op;
    sbuf.sem_flg = 0;
    ec_neg1( r = semop(semid, &sbuf, 1) )
    return r;
```

```
EC_CLEANUP_BGN
    return -1;
EC_CLEANUP_END
}
```

There's also a handy function to initialize a semaphore:

```
static int init_semi(int semid)
{
    union semun arg;
    int r;

    arg.val = 0;
    semctl(semid, SEMI_WRITE, SETVAL, arg);
    semctl(semid, SEMI_READ, SETVAL, arg);
    /* Following call will set otime, allowing clients to proceed. */
    ec_neg1( r = op_semi(semid, SEMI_WRITE, SEMI_POST) )
    return r;

EC_CLEANUP_BGN
    return -1;
EC_CLEANUP_END
}
```

This is the internal data structure behind the `SMIQ` type that the SMI uses:

```
typedef struct {
    SMIENTITY sq_entity;          /* entity */
    int sq_semid_server;          /* server sem */
    int sq_semid_client;          /* client sem (client only) */
    int sq_shmid_server;          /* server shm ID */
    int sq_shmid_client;          /* client shm ID (client only) */
    struct smi_msg *msg_server;   /* ptr to server shm */
    struct smi_msg *msg_client;   /* ptr to client shm (client only) */
    char sq_name[SERVER_NAME_MAX];/* server name */
    struct client_id sq_client;   /* client identification (server only) */
} SMIQ_SHM;
```

A client uses almost the whole structure, but the server just uses some of the members. A client stores:

- Its entity (`SMI_CLIENT`) and the server name
- The semaphore-set IDs (2 semaphores each) for itself and the server
- The shared-memory segment IDs for itself and the server
- Pointers (from `shmat`) to its and the server's segments

The server stores:

- Its entity (SMI_SERVER) and name.
- Its semaphore-set ID (2 semaphores).
- Its shared-memory segment ID.
- A pointer to its segment.
- The client_id passed to smi_send_getaddr, so it can use it in the following smi_send_release. This is how it knows which client to send to. For this use, and in messages, the c_id1 member of the client_id structure is the shared-memory identifier, and the c_id2 member is the semaphore-set identifier. (See the code for smi_send_getaddr_shm, below, to see where these members are set.)

The server doesn't store any client's semaphore-set or shared-memory information because there are a lot of clients. This information can be readily passed in an incoming message from a client, similar to the way a message-queue ID was passed in the message in the System V message-queue SMI implementation (Section 7.5.3). We'll see the details soon.

Given this description of how the structure is used, the code for smi_open_shm should be understandable:

```
SMIQ *smi_open_shm(const char *name, SMIENTITY entity, size_t msgsize)
{
    SMIQ_SHM *p = NULL;
    char shmname[FILENAME_MAX];
    int i;
    key_t key;

    ec_null( p = calloc(1, sizeof(SMIQ_SHM)) )
    p->sq_entity = entity;
    if (strlen(name) >= SERVER_NAME_MAX) {
        errno = ENAMETOOLONG;
        EC_FAIL
    }
    strcpy(p->sq_name, name);
    mkshm_name_server(p, shmname, sizeof(shmname));
    (void)close(open(shmname, O_WRONLY | O_CREAT, 0));
    ec_neg1( key = ftok(shmname, 1) )
    if (p->sq_entity == SMI_SERVER) {
        if ((p->sq_semid_server = semget(key, 2, PERM_FILE)) != -1)
            (void)shmctl(p->sq_semid_server, IPC_RMID, NULL);
        ec_neg1( p->sq_semid_server = semget(key, 2,
          PERM_FILE | IPC_CREAT) )
        p->sq_semid_client = -1;
        if ((p->sq_shmid_server = shmget(key, 0, PERM_FILE)) != -1)
            (void)shmctl(p->sq_shmid_server, IPC_RMID, NULL);
```

```
        ec_neg1( p->sq_shmid_server = shmget(key, msgsize,
          PERM_FILE | IPC_CREAT) )
        p->sq_shmid_client = -1;
        ec_neg1( init_semi(p->sq_semid_server) )
    }
    else {
        ec_neg1( p->sq_semid_server = semget(key, 2, PERM_FILE) )
        ec_neg1( p->sq_semid_client = semget(IPC_PRIVATE, 2,
          PERM_FILE | IPC_CREAT) )
        ec_neg1( p->sq_shmid_server = shmget(key, msgsize, PERM_FILE) )
        ec_neg1( p->sq_shmid_client = shmget(IPC_PRIVATE, msgsize,
          PERM_FILE | IPC_CREAT) )
        ec_neg1( p->msg_client = shmat(p->sq_shmid_client, NULL, 0) )
        ec_neg1( init_semi(p->sq_semid_client) )
        for (i = 0; !smi_client_nowait && i < 10; i++) {
            union semun arg;
            struct semid_ds ds;

            arg.buf = &ds;
            ec_neg1( semctl(p->sq_semid_server, SEMI_WRITE, IPC_STAT,
              arg) )
            if (ds.sem_otime > 0)
                break;
            sleep(1);
        }
    }
    ec_neg1( p->msg_server = shmat(p->sq_shmid_server, NULL, 0) )
    return (SMIQ *)p;

EC_CLEANUP_BGN
    free(p);
    return NULL;
EC_CLEANUP_END
}
```

The first part, through the call to `ftok`, is almost identical to the code in the version of this function for System V message queues in Section 7.5.3. Then, for the server, it removes an old semaphore set, if there is one, and creates a new one, and the same for a shared-memory segment. The client creates a private semaphore set and shared-memory segment. Both attach the server's shared-memory segment.

Note that `init_semi` (shown earlier), which posts the writing semaphore (to allow sends to proceed), also sets the operation-time, which is how another process can tell that the semaphore has been initialized, as explained in Section 7.9.1. We showed a waiting algorithm in the implement of `SimpleSem` in that section; the algorithm here is a little more elaborate:

- We try at most 10 times because FreeBSD and Darwin systems don't set the time, and we don't want to get stuck.
- There's an under-the-covers global variable `smi_client_nowait` that can be used to prevent any waiting. It's used during timing tests when we know that the server started well in front of any clients. We used it in the program that developed the data for the comparison table at the end of this chapter.

`smi_close_shm` is pretty simple:

```
bool smi_close_shm(SMIQ *sqp)
{
    SMIQ_SHM *p = (SMIQ_SHM *)sqp;

    if (p->sq_entity == SMI_SERVER) {
        char shmname[FILENAME_MAX];

        (void)getaddr(-1);
        ec_neg1( semctl(p->sq_semid_server, 0, IPC_RMID) )
        (void)shmdt(p->msg_server);
        (void)shmctl(p->sq_shmid_server, IPC_RMID, NULL);
        mkshm_name_server(p, shmname, sizeof(shmname));
        (void)unlink(shmname);
    }
    else {
        ec_neg1( semctl(p->sq_semid_client, 0, IPC_RMID) )
        (void)shmdt(p->msg_server);
        (void)shmdt(p->msg_client);
        (void)shmctl(p->sq_shmid_client, IPC_RMID, NULL);
    }
    free(p);
    return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}
```

Now we're ready for `smi_send_getaddr_shm`:

```
bool smi_send_getaddr_shm(SMIQ *sqp, struct client_id *client,
  void **addr)
{
    SMIQ_SHM *p = (SMIQ_SHM *)sqp;
    int semid_receiver;

    if (p->sq_entity == SMI_SERVER) {
        semid_receiver = client->c_id2;
        p->sq_client = *client;
    }
```

```
    else
        semid_receiver = p->sq_semid_server;
    ec_neg1( op_semi(semid_receiver, SEMI_WRITE, SEMI_WAIT) )
    if (p->sq_entity == SMI_SERVER)
        ec_null( *addr = getaddr(client->c_id1) )
    else {
        *addr = p->msg_server;
        ((struct smi_msg *)*addr)->smi_client.c_id1 = p->sq_shmid_client;
        ((struct smi_msg *)*addr)->smi_client.c_id2 = p->sq_semid_client;
    }
    return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}
```

For sends from the server, the `client` argument must point to client identification; for sends from the client, it's `NULL`. Let's first track the code for the server and then for a client.

For the server, `semid_receiver` is set from the `c_id2` member, and the whole `client_id` structure is saved for later use by `smi_send_release`, as I mentioned earlier. Then we wait on the `SEMI_WRITE` semaphore. It's been posted in `init_semi`, so at the start we proceed immediately. The message itself is in the shared-memory segment given by `client->c_id1`, and we call `getaddr` to get its address, which is what's returned through the `addr` argument. We'll see `getaddr` shortly; for now, think of it as just doing a `shmat`.

For a client, it's actually simpler, as the client already has all it needs to access its and the server's shared-memory segments and semaphores. It sets `semid_receiver` directly from the `SMIQ_SHM` structure, waits on the `SEMI_WRITE` semaphore, sets the returned address, and stores identifiers in the message for its shared-memory segment and semaphore set. That's how the receiver (the server) will know who sent the message and how to respond.

Once `smi_send_getaddr_shm` returns, its caller is free to use the returned address of the message, and that memory is locked from further writing by any other client. (Somewhat inefficient, as I noted earlier—a pool of incoming server messages would be better, but much more complicated to implement.)

When the caller is done, it calls `smi_send_release`:

```
bool smi_send_release_shm(SMIQ *sqp)
{
    SMIQ_SHM *p = (SMIQ_SHM *)sqp;
    int semid_receiver;

    if (p->sq_entity == SMI_SERVER)
        semid_receiver = p->sq_client.c_id2;
    else
        semid_receiver = p->sq_semid_server;
    ec_neg1( op_semi(semid_receiver, SEMI_READ, SEMI_POST) )
    return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}
```

All this function needs to do is post the `SEMI_READ` semaphore, but to do that it needs the semaphore-set identifier. For the server, it's in the `client_id` that `smi_send_getaddr_shm` stored in the `SMIQ_SHM` structure; for the client we had it in that structure from the start.

At this point you should be able to follow `smi_receive_getaddr_shm` pretty easily:

```
bool smi_receive_getaddr_shm(SMIQ *sqp, void **addr)
{
    SMIQ_SHM *p = (SMIQ_SHM *)sqp;
    int semid_receiver;

    if (p->sq_entity == SMI_SERVER)
        semid_receiver = p->sq_semid_server;
    else
        semid_receiver = p->sq_semid_client;
    ec_neg1( op_semi(semid_receiver, SEMI_READ, SEMI_WAIT) )
    if (p->sq_entity == SMI_SERVER)
        *addr = p->msg_server;
    else
        *addr = p->msg_client;
    return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}
```

The corresponding function `smi_receive_release_shm` has to post the

SEMI_WRITE semaphore:

```
bool smi_receive_release_shm(SMIQ *sqp)
{
    SMIQ_SHM *p = (SMIQ_SHM *)sqp;
    int semid_receiver;

    if (p->sq_entity == SMI_SERVER)
        semid_receiver = p->sq_semid_server;
    else
        semid_receiver = p->sq_semid_client;
    ec_neg1( op_semi(semid_receiver, SEMI_WRITE, SEMI_POST) )
    return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}
```

That's it. Lots of horsing around with semaphores, but very little with the shared memory. If the messages are large (100,000 bytes, say), this is a huge win. Message queues (either System V or POSIX) probably couldn't even handle messages of that size, and, even if they could, the copying from user space to the kernel and back on each message would really slow things down. The speed of the shared-memory SMI implementation, by contrast, is independent of the message size.

7.13.4 System V Shared Memory Critiqued

Not much to dislike about System V shared memory. The system calls are reasonably straightforward, efficient, and usually very well implemented. The hard part is synchronization, which is also the hard part about using threads, and for exactly the same reason: Once things are shared, the program runs fast, but possibly incorrectly, and the work to make it right is exceedingly difficult to test. You have to prove it's correct and then implement it flawlessly.[13]

7.14 POSIX Shared Memory

This section explains the POSIX shared-memory system calls and shows an SMI implementation using POSIX shared memory and POSIX semaphores.

13. About 30 years ago, a colleague of mine said that if a program didn't have to be correct, he could make it arbitrarily fast—he would just change it to print zero and throw away all the other code. So, if you can't guarantee that your use of shared memory or threads is correct, don't use those features—something slower but correct would be more efficient.

7.14.1 POSIX Shared-Memory System Calls

POSIX shared memory involves opening, and perhaps creating, a shared-memory segment with a POSIX IPC name, with all of the entanglements described in Section 7.6.2. The call, `shm_open`, returns a file descriptor, as though a file had been opened. Indeed, POSIX shared-memory segments act like in-memory files. Once opened, you set the size with `ftruncate`, which we saw way back in Section 2.17, as though you're setting the size of a file, which you are. Then you map the segment to your address space with `mmap`, which can map files in general, not just those that are shared-memory segments. In other words, only `shm_open` and `shm_unlink` are specialized for POSIX shared memory. The other calls work on any regular files.

Here are the shared-memory-specific calls:

shm_open—open shared-memory object

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

int shm_open(
    const char *name,      /* POSIX IPC name */
    int flags,             /* flags */
    mode_t perms           /* permissions */
);
/* Returns file descriptor or -1 on error (sets errno) */
```

shm_unlink—remove shared-memory object

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

int shm_unlink(
    const char *name       /* POSIX IPC name */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

The flags for `shm_open` are the ones we've already seen for other file-oriented system calls: `O_CREAT`, `O_EXCL`, `O_TRUNC`, `O_RDONLY`, and `O_RDWR`. You can't use `O_WRONLY`. If the object is created, the third argument establishes its permissions.

As with other POSIX and System V IPC objects, the contents of a POSIX shared-memory object persist at least until the system is rebooted.

You close the file descriptor when you're done with it using `close`, just as for any other file descriptor.

[SUS2002] doesn't say that you can actually do I/O on the shared-memory object, using, say, `read` and `write`, but it's possible for an implementation to allow that. This would be a way of using an in-memory file like any other file. The whole point of having an in-memory file, however, is so you can use ordinary C or C++ operations on it, not go through the expense of an I/O system call, so, even if this feature were supported, it wouldn't be very useful.

Usually, after creating a shared-memory object you set its size with `ftruncate`, as its initial size is zero bytes. Here's a recap of `ftruncate` from Section 2.17:

ftruncate—truncate or stretch a file by file descriptor

```
#include <unistd.h>

int ftruncate(
    int fd,             /* file descriptor */
    off_t length        /* new length */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

Next, you map the object to your address space (analogously to using `shmat` on System V shared-memory objects):

mmap—map pages of memory

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

void *mmap(
    void *addr,         /* desired address or NULL */
    size_t len,         /* length of segment */
    int prot,           /* protection (see below) */
    int flags,          /* flags */
    int fd,             /* file descriptor */
    off_t off           /* offset in file or shared-memory object */
);
/* Returns pointer to segment or MAP_FAILED on error (sets errno) */
```

The first argument, `addr`, is an address near where the segment should be mapped, or exactly where it should be mapped if `MAP_FIXED` is set in the `flags` argument. Otherwise, it's `NULL`. This means you'll take any address, which is the most common case. We'll do it that way in our examples.

The part of the object to be mapped starts at `off` within the object and extends for `len` bytes. It's not necessary to map the whole object (whose size was set with

`ftruncate`) at once, although that's what we'll do in our examples, so `off` will be zero and `len` will be the same as the argument to `ftruncate`.

The `prot` argument is either `PROT_NONE`, meaning that the memory can't be accessed at all, or one or more of the following flags ORed together:

`PROT_READ` Data can be read.

`PROT_WRITE` Data can be written.

`PROT_EXEC` Data can be executed (may be unsupported).

For our purposes, we're going to use `PROT_READ | PROT_WRITE`. Also, the only flag for the flags argument we'll use is `MAP_SHARED`, which means that all changes to the segment are immediately visible. The alternative is `MAP_PRIVATE`, which means changes are private to the process that did the `mmap`. `MAP_PRIVATE` with shared memory doesn't make much sense.

`mmap` is another one of those pointer-returning functions that has a special symbol, `MAP_FAILED`, in this case, for the error return. Make sure you test against it and not `NULL`.

So, all of the above means that to map all of a shared-memory object for reading and writing we'll do this:

```
ec_neg1( ftruncate(fd, len) )
mem = mmap(NULL, len, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
ec_cmp(mem, MAP_FAILED)
```

When you're finished with a mapped segment, you unmap it:

munmap—unmap pages of memory

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

int munmap(
    void *addr,        /* pointer to segment */
    size_t len         /* length of segment */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

You don't have to `unmap` all of what was mapped, but we will in our examples. Thus, `addr` will be what `mmap` returned and `len` is the same as what was passed into `mmap`.

7.14.2 POSIX Shared-Memory Implementation of SMI

The POSIX shared-memory implementation of the SMI functions is very similar to the System V shared-memory implementation (from Section 7.13.3, which you should review before proceeding), except that instead of distinct semaphore objects, we're going to take advantage of a POSIX semaphore feature and use in-memory semaphores. Since they have to be shared between server and client, we'll put them in the shared-memory segment.

To be more precise, each shared memory segment (one for the server and one for each client) has this layout:

```
struct shared_mem {
    sem_t sm_sem_w;
    sem_t sm_sem_r;
    struct smi_msg sm_msg; /* variable size -- must be last */
};
```

The data part of the `smi_msg` extends to the end of the segment, the size of which is calculated from what's passed to `smi_open_pshm` using this macro

```
#define MEM_SIZE(s)\
  (sizeof(struct shared_mem) - sizeof(struct smi_msg) + (s))
```

where `s` is the third argument to `smi_open_pshm`.

```
#define SEMI_READ       0
```

Given a pointer to a `struct shared_mem`, in shared memory, here's a handy function that performs a *semwait*, *sempost*, or destroy operation:

```
#define SEMI_WRITE      1
#define SEMI_DESTROY    2
#define SEMI_POST       1
#define SEMI_WAIT       -1

static int op_semi(struct shared_mem *m, int sem_num, int sem_op)
{
    sem_t *sem_p = NULL;

    if (sem_num == SEMI_WRITE)
        sem_p = &m->sm_sem_w;
    else
        sem_p = &m->sm_sem_r;
    switch (sem_op) {
    case SEMI_WAIT:
        ec_neg1( sem_wait(sem_p) )
        break;
```

```
    case SEMI_POST:
        ec_neg1( sem_post(sem_p) )
        break;
    case SEMI_DESTROY:
        ec_neg1( sem_destroy(sem_p) )
    }
    return 0;

EC_CLEANUP_BGN
    return -1;
EC_CLEANUP_END
}
```

So, if `m` is such a pointer, we can make calls like these:

```
ec_neg1( op_semi(m, SEMI_READ, SEMI_POST) )
ec_neg1( op_semi(m, SEMI_WRITE, SEMI_WAIT) )
```

The POSIX calls don't allow a client to just pass an identifier for a shared-memory segment along with a message to the server. Instead, we have to use an approach similar to what we used with FIFOs back in Section 7.3.3: A client passes its process ID, and the server uses that to form the POSIX name for the object and then opens it and maps it. This is expensive, so the server does this only once per client, looking up an already-mapped segment in a table. All of that and more is stored in the `SMIQ_PSHM` structure, which takes on what by now should be a familiar shape:

```
#define MAX_CLIENTS 50

typedef struct {
    SMIENTITY sq_entity;            /* entity */
    char sq_name[SERVER_NAME_MAX];  /* server name */
    int sq_srv_fd;                  /* server shm file descriptor */
    struct shared_mem *sq_srv_mem;  /* server mapped shm segment */
    struct client {
        pid_t cl_pid;               /* client process ID */
        int cl_fd;                  /* client shm file descriptor */
        struct shared_mem *cl_mem;  /* client mapped shm segment */
    } sq_clients[MAX_CLIENTS];      /* client uses only [0] */
    struct client_id sq_client;     /* client id (server only) */
    size_t sq_msgsize;              /* message size */
} SMIQ_PSHM;
```

The server can keep track of 50 clients. Missing entirely from our implementation is a way for a client to tell the server that it's finished so the server can re-use its slot.

Now we're ready to see how the `SMIQ_PSHM` structure is populated by `smi_open_pshm`.

```
SMIQ *smi_open_pshm(const char *name, SMIENTITY entity, size_t msgsize)
{
    SMIQ_PSHM *p = NULL;
    char shmname[SERVER_NAME_MAX + 50];

    ec_null( p = calloc(1, sizeof(SMIQ_PSHM)) )
    p->sq_entity = entity;
    p->sq_msgsize = msgsize;
    if (strlen(name) >= SERVER_NAME_MAX) {
        errno = ENAMETOOLONG;
        EC_FAIL
    }
    strcpy(p->sq_name, name);
    mkshm_name_server(p, shmname, sizeof(shmname));
    if (p->sq_entity == SMI_SERVER) {
        if ((p->sq_srv_fd = shm_open(shmname, O_RDWR, PERM_FILE)) != -1) {
            (void)shm_unlink(shmname);
            (void)close(p->sq_srv_fd);
        }
        ec_neg1( p->sq_srv_fd = shm_open(shmname, O_RDWR | O_CREAT,
          PERM_FILE) )
        ec_neg1( ftruncate(p->sq_srv_fd, MEM_SIZE(msgsize)) )
        p->sq_srv_mem = mmap(NULL, MEM_SIZE(msgsize),
          PROT_READ | PROT_WRITE, MAP_SHARED, p->sq_srv_fd, 0);
        ec_cmp(p->sq_srv_mem, MAP_FAILED)
        ec_neg1( sem_init(&p->sq_srv_mem->sm_sem_w, true, 1) )
        ec_neg1( sem_init(&p->sq_srv_mem->sm_sem_r, true, 0) )
    }
    else {
        ec_neg1( p->sq_srv_fd = shm_open(shmname, O_RDWR, PERM_FILE) )
        p->sq_srv_mem = mmap(NULL, MEM_SIZE(msgsize),
          PROT_READ | PROT_WRITE, MAP_SHARED, p->sq_srv_fd, 0);
        ec_cmp(p->sq_srv_mem, MAP_FAILED)
        mkshm_name_client(getpid(), shmname, sizeof(shmname));
        if ((p->sq_clients[0].cl_fd = shm_open(shmname, O_RDWR, PERM_FILE))
          != -1) {
            (void)shm_unlink(shmname);
            (void)close(p->sq_clients[0].cl_fd);
        }
        ec_neg1( p->sq_clients[0].cl_fd = shm_open(shmname,
          O_RDWR | O_CREAT, PERM_FILE) )
        ec_neg1( ftruncate(p->sq_clients[0].cl_fd, MEM_SIZE(msgsize)) )
        p->sq_clients[0].cl_mem = mmap(NULL, MEM_SIZE(msgsize),
          PROT_READ | PROT_WRITE, MAP_SHARED, p->sq_clients[0].cl_fd, 0);
        ec_cmp(p->sq_clients[0].cl_mem, MAP_FAILED)
        ec_neg1( sem_init(&p->sq_clients[0].cl_mem->sm_sem_w, true,
          1) )
        ec_neg1( sem_init(&p->sq_clients[0].cl_mem->sm_sem_r, true,
          0) )
    }
    return (SMIQ *)p;
```

```
EC_CLEANUP_BGN
    if (p != NULL)
        (void)smi_close_pshm((SMIQ *)p);
    return NULL;
EC_CLEANUP_END
}

static void mkshm_name_server(const SMIQ_PSHM *p, char *shmname,
  size_t shmname_max)
{
    snprintf(shmname, shmname_max, "/smipshm-%s", p->sq_name);
}

static void mkshm_name_client(pid_t pid, char *shmname,
  size_t shmname_max)
{
    snprintf(shmname, shmname_max, "/smipshm-%d", pid);
}
```

The first few lines, up through the call to `mkshm_name_server`, are just like what we've seen before in previous implementations. Then the server creates a fresh shared-memory object (removing the old one), sets its size, and maps it. Once that's done, the semaphores are in memory, and they're initialized.

A client also maps in the server's segment, but it doesn't set its size nor does it initialize the server's semaphores because the server already did that. A client does, however, create, size, and map its own segment, which it uses the first element of the array for, and it initializes its own semaphores.

As I said, the server won't map in a client's segment until it gets a message from that client, since it doesn't know in advance who the clients may be. Here's a function for the server to use (we'll see where just below) to get a mapped-in address to a client's segment, given only the client's process ID, which, as we've seen before, is sent in each message from client to server:

```
static struct client *get_client(SMIQ_PSHM *p, pid_t pid)
{
    int i, avail = -1;
    char shmname[SERVER_NAME_MAX + 50];

    for (i = 0; i < MAX_CLIENTS; i++) {
        if (p->sq_clients[i].cl_pid == pid)
            return &p->sq_clients[i];
        if (p->sq_clients[i].cl_pid == 0 && avail == -1)
            avail = i;
    }
```

```
    if (avail == -1) {
        errno = EADDRNOTAVAIL;
        EC_FAIL
    }
    p->sq_clients[avail].cl_pid = pid;
    mkshm_name_client(pid, shmname, sizeof(shmname));
    ec_neg1( p->sq_clients[avail].cl_fd = shm_open(shmname, O_RDWR,
      PERM_FILE) )
    p->sq_clients[avail].cl_mem = mmap(NULL, MEM_SIZE(p->sq_msgsize),
      PROT_READ | PROT_WRITE, MAP_SHARED, p->sq_clients[avail].cl_fd,
      0);
    ec_cmp(p->sq_clients[avail].cl_mem, MAP_FAILED)
    return &p->sq_clients[avail];

EC_CLEANUP_BGN
    return NULL;
EC_CLEANUP_END
}
```

To see `get_client` in use, let's look next at `smi_send_getaddr_pshm`:

```
bool smi_send_getaddr_pshm(SMIQ *sqp, struct client_id *client,
  void **addr)
{
    SMIQ_PSHM *p = (SMIQ_PSHM *)sqp;
    struct client *cp;
    struct shared_mem *sm;

    if (p->sq_entity == SMI_SERVER) {
        p->sq_client = *client;
        ec_null( cp = get_client(p, client->c_id1) )
        sm = cp->cl_mem;
    }
    else
        sm = p->sq_srv_mem;
    ec_neg1( op_semi(sm, SEMI_WRITE, SEMI_WAIT) )
    if (p->sq_entity == SMI_CLIENT)
        sm->sm_msg.smi_client.c_id1 = getpid();
    *addr = &sm->sm_msg;
    return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}
```

Recall that the argument `client` is only used when the server sends a message; it got that structure from the message that it previously received. The `c_id1` mem-

ber is the process ID, and that's what's passed to `get_client`. For a client, the address of the server's segment is right in the `SMIQ_PSHM` structure. With the segment address, we wait on the writing semaphore, and, when it's available, the segment is ours to use. A client then stores its process ID in the message, and the address is returned.

The counterpart is `smi_send_release_pshm`:

```
bool smi_send_release_pshm(SMIQ *sqp)
{
    SMIQ_PSHM *p = (SMIQ_PSHM *)sqp;
    struct client *cp;
    struct shared_mem *sm;

    if (p->sq_entity == SMI_SERVER) {
        ec_null( cp = get_client(p, p->sq_client.c_id1) )
        sm = cp->cl_mem;
    }
    else
        sm = p->sq_srv_mem;
    ec_neg1( op_semi(sm, SEMI_READ, SEMI_POST) )
    return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}
```

This function when called by the server calls `get_client` to get the address of the client's segment using the process ID (`c_id1` member) that `smi_send_getaddr_pshm` saved. The client's segment was already mapped by `smi_send_getaddr_pshm`—we just need to look it up. For a client, as before, the server's segment is right in the `SMIQ_PSHM` structure. Once we have the segment, the only work we need to do is post the reading semaphore, which is in the segment. That will allow a `smi_receive_getaddr_pshm` on the segment to proceed, as we're about to see:

```
bool smi_receive_getaddr_pshm(SMIQ *sqp, void **addr)
{
    SMIQ_PSHM *p = (SMIQ_PSHM *)sqp;
    struct shared_mem *sm;

    if (p->sq_entity == SMI_SERVER)
        sm = p->sq_srv_mem;
```

```
    else
        sm = p->sq_clients[0].cl_mem;
    ec_neg1( op_semi(sm, SEMI_READ, SEMI_WAIT) )
    *addr = &sm->sm_msg;
    return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}
```

Since the server and a client each receive from their own segments, the address is available and can be passed right to `op_semi`. There's no need for a call to `get_client`. Once the segment is available for reading, its address is returned.

Finally, `smi_receive_release_pshm` posts the writing semaphore for the received segment once the receiver has finished reading it:

```
bool smi_receive_release_pshm(SMIQ *sqp)
{
    SMIQ_PSHM *p = (SMIQ_PSHM *)sqp;
    struct shared_mem *sm;

    if (p->sq_entity == SMI_SERVER)
        sm = p->sq_srv_mem;
    else
        sm = p->sq_clients[0].cl_mem;
    ec_neg1( op_semi(sm, SEMI_WRITE, SEMI_POST) )
    return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}
```

If you haven't been comparing this implementation with the one for System V shared memory from Section 7.13.3, you should do so now. You'll see that they're very similar but with these differences:

- With the System V calls, a client can pass in each message shared-memory-segment and semaphore-set identifiers that the server can use to access the objects quickly. Therefore, the server doesn't need a table to keep track of clients.

- With the POSIX calls, we used in-memory semaphores that were inside the shared memory, rather than separate semaphore objects. In theory, these should be faster, although whether they are or not depends on the implementation.

7.14.3 POSIX Shared Memory Critiqued

For shared memory, the POSIX and System V interfaces are about equally convenient. Whether one is more efficient than the other depends on the implementation, but it's likely that any implementation that has both would use the same internal mechanisms in the kernel for both.

The key advantage of POSIX shared memory is that the mapping call, `mmap`, works for any regular file, not just an in-memory file opened with `shm_open`. You have a lot of control over which part of the segment is mapped at any one time.

The key disadvantage of POSIX shared memory is that it's not always available. It's not in some versions of Linux, FreeBSD, or Darwin, for example.

7.15 Performance Comparisons

With an implementation of SMI for six different IPC methods (including sockets in Chapter 8), it's easy to construct a test program that compares the times for sending messages of different sizes. For all my tests, I sent 5000 messages between four clients and a server. Then I normalized the numbers for four systems (Solaris, FreeBSD, Darwin, and Linux) by dividing by the FIFO time for 100-byte messages on that system. Otherwise, the results would be misleading since the computer hardware for the four systems varies widely in performance. Table 7.2 shows the results. The two large-sized messages could be sent using only shared memory and POSIX message queues. Also, as of this writing, FreeBSD, Darwin, and Linux don't support POSIX message queues or POSIX shared memory.

Table 7.2 Performance of Different Message-Passing Methods

Method	100-byte msg	2000-byte msg	20,000-byte msg	100,000-byte msg
FIFO	S: 1.00 B: 1.00 D: 1.00 L: 1.00	S: 1.22 B: 1.46 D: 1.29 L: 1.51	too big	too big
System V message queue	S: 0.90 B: 0.62 L: 0.31	S: 1.82 B: 3.76 L: 0.64	too big	too big
POSIX message queue	S: 2.02	S: 2.40	S: 7.03	S: 33.39
System V shared memory	S: 1.47 B: 0.94 D: 1.04 L: 0.55	S: 1.41 B: 0.91 D: 1.07 L: 0.53	S: 1.55 B: 0.90 D: 1.02 L: 0.47	S: 1.24 B: 0.90 D: 1.06 L: 0.51
POSIX shared memory	S: 1.27	S: 1.25	S: 1.41	S: 1.37
Sockets	S: 1.84 B: 0.81 D: 1.04 L: 0.75	S: 2.15 B: 1.00 D: 1.27 L: 0.95	S: 10.15 B: 7.99 D: 5.52 L: 6.06	S: 44.13 B: 35.83 D: 25.86 L: 31.91

S: Solaris; B: FreeBSD; D: Darwin; L: Linux. All times normalized for each system so FIFO time for 100-byte messages is 1.00. Sockets used the `AF_UNIX` domain (see Chapter 8).

Some comments on the results:

- They're not definitive because this was only one possible test, SMI is only one possible interface, and our implementations weren't optimal—they were mainly designed to serve as textbook examples.
- Even with the normalization, a system with a faster time for a given method doesn't necessarily implement that method better. It might implement FIFOs (the normalization divisor) slower.

- For small messages (100 bytes or so), System V message queues performed really well on all the systems, except for Darwin 6.6, which doesn't support them.
- Sockets perform almost as well as the two message-queue methods (even better in a few cases), and handle messages of any size. In addition, they're the only method that goes between machines, including over the Internet, although the timing tests used them only within a single machine.
- On Solaris, POSIX message queues performed less well than System V message queues but had the advantage of handling very large messages.
- As we would expect, the performance of the two shared-memory methods is independent of the message size.

Therefore, if we had to make a sweeping generalization, we'd say that, for maximum performance, use System V message queues for small messages, and use shared memory for big messages. One idea we didn't try combines the two: Use a System V message to tell the server when it has work to do, but use a shared-memory segment to pass the data.

If optimal performance isn't essential and you'd like your life to be simple, use sockets for everything. They're fairly efficient, they handle messages of any size, they're universally supported, and they go between machines.

These recommendations are only for message-oriented IPC. For other uses, one IPC method may be better than another for your purposes.

Exercises

The first six Exercises ask you to implement System V IPC functions with POSIX IPC functions or the other way around. In most cases you won't be able to implement every feature and behavior, so part of the Exercise is to carefully document exactly where your implementation falls short.

7.1. Implement `msgget`, `msgctl`, `msgsnd`, and `msgrcv` with POSIX IPC functions.

7.2. Implememt `mq_open`, `mq_close`, `mq_unlink`, `mq_send`, and `mq_receive` with System V IPC functions.

7.3. Implement `semget`, `semctl`, and `semop` with POSIX IPC functions, allowing only one semaphore per set. Can you handle increments and decrements of more than one?

7.4. Same as the previous Exercise, but with more than one semaphore per set.

7.5. Implement `sem_open`, `sem_close`, `sem_unlink`, `sem_post`, and `sem_wait` with System V IPC functions.

7.6. Implement `shmget`, `shmctl`, `shmat`, and `shmdt` with POSIX IPC functions.

7.7. There's no Exercise asking you to implement `shm_open`, `shm_unlink`, `ftruncate`, `mmap`, and `munmap` with System V IPC functions. Why not? Can you propose a suitable Exercise? Can you complete it?

7.8. Design and conduct an experiment to compare the efficiency of conventional I/O on a regular file using `read` and `write` with access via a memory-mapped file. Include both sequential and random I/O, and perhaps several variations of each.

7.9. Design and conduct an experiment to compare the efficiency of write locks only (using `lockf`) with a combination of read and write locks (using `fcntl`). Try to create a pattern of reads and writes that shows the greatest difference in efficiency.

7.10. Extend the program you wrote in Exercise 5.14 to include process attributes from Appendix A that are explained in this chapter.