11

System V Semaphores

11.1 Introduction

When we described the concept of a semaphore in Chapter 10, we first described

- a *binary semaphore*: a semaphore whose value is 0 or 1. This was similar to a mutex lock (Chapter 7), in which the semaphore value is 0 if the resource is locked, or 1 if the resource is available.

The next level of detail expanded this into

- a *counting semaphore*: a semaphore whose value is between 0 and some limit (which must be at least 32767 for Posix semaphores). We used these to count resources in our producer–consumer problem, with the value of the semaphore being the number of resources available.

In both types of semaphores, the *wait* operation waits for the semaphore value to be greater than 0, and then decrements the value. The *post* operation just increments the semaphore value, waking up any threads awaiting the semaphore value to be greater than 0.

System V semaphores add another level of detail to semaphores by defining

- a *set of counting semaphores*: one or more semaphores (a set), each of which is a counting semaphore. There is a limit to the number of semaphores per set, typically on the order of 25 semaphores (Section 11.7). When we refer to a "System V semaphore," we are referring to a set of counting semaphores. when we refer to a "Posix semaphore," we are referring to a single counting semaphore.

For every set of semaphores in the system, the kernel maintains the following structure of information, defined by including <sys/sem.h>:

```
struct semid_ds {
  struct ipc_perm  sem_perm;  /* operation permission struct */
  struct sem      *sem_base;  /* ptr to array of semaphores in set */
  ushort           sem_nsems; /* # of semaphores in set */
  time_t           sem_otime; /* time of last semop() */
  time_t           sem_ctime; /* time of creation or last IPC_SET */
};
```

The ipc_perm structure was described in Section 3.3 and contains the access permissions for this particular semaphore.

The sem structure is the internal data structure used by the kernel to maintain the set of values for a given semaphore. Every member of a semaphore set is described by the following structure:

```
struct sem {
  ushort_t  semval;   /* semaphore value, nonnegative */
  short     sempid;   /* PID of last successful semop(), SETVAL, SETALL */
  ushort_t  semncnt;  /* # awaiting semval > current value */
  ushort_t  semzcnt;  /* # awaiting semval = 0 */
};
```

Note that sem_base contains a pointer to an array of these sem structures: one array element for each semaphore in the set.

In addition to maintaining the actual values for each semaphore in the set, the kernel also maintains three other pieces of information for each semaphore in the set: the process ID of the process that performed the last operation on this value, a count of the number of processes waiting for the value to increase, and a count of the number of processes waiting for the value to become zero.

> Unix 98 says that the above structure is anonymous. The name that we show, sem, is from the historical System V implementation.

We can picture a particular semaphore in the kernel as being a semid_ds structure that points to an array of sem structures. If the semaphore has two members in its set, we would have the picture shown in Figure 11.1. In this figure, the variable sem_nsems has a value of two, and we have denoted each member of the set with the subscripts [0] and [1].

11.2 semget Function

The semget function creates a semaphore set or accesses an existing semaphore set.

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

int semget(key_t key, int nsems, int oflag);
```

Returns: nonnegative identifier if OK, –1 on error

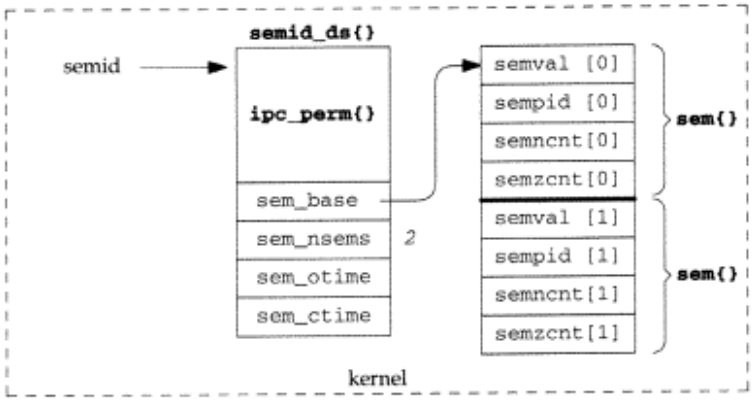

Figure 11.1 Kernel data structures for a semaphore set with two values in the set.

The return value is an integer called the *semaphore identifier* that is used with the `semop` and `semctl` functions.

The *nsems* argument specifies the number of semaphores in the set. If we are not creating a new semaphore set but just accessing an existing set, we can specify this argument as 0. We cannot change the number of semaphores in a set once it is created.

The *oflag* value is a combination of the `SEM_R` and `SEM_A` constants shown in Figure 3.6. `R` stands for "read" and `A` stands for "alter." This can be bitwise-ORed with either `IPC_CREAT` or `IPC_CREAT | IPC_EXCL`, as discussed with Figure 3.4.

When a new semaphore set is created, the following members of the `semid_ds` structure are initialized:

- The `uid` and `cuid` members of the `sem_perm` structure are set to the effective user ID of the process, and the `gid` and `cgid` members are set to the effective group ID of the process.
- The read–write permission bits in `oflag` are stored in `sem_perm.mode`.
- `sem_otime` is set to 0, and `sem_ctime` is set to the current time.
- `sem_nsems` is set to *nsems*.
- The `sem` structure associated with each semaphore in the set is *not* initialized. These structures are initialized when `semctl` is called with either the `SETVAL` or `SETALL` commands.

Initialization of Semaphore Value

Comments in the source code in the 1990 edition of this book incorrectly stated that the semaphore values in the set were initialized to 0 by `semget` when a new set was created. Although some systems do initialize the semaphore values to 0, this is not guaranteed. Indeed, older implementations of System V do not initialize the semaphore

values at all, leaving their values as whatever they were the last time that piece of memory was used.

Most manual pages for `semget` say nothing at all about the initial values of the semaphores when a new set is created. The X/Open XPG3 portability guide (1989) and Unix 98 correct this omission and explicitly state that the semaphore values are not initialized by `semget` and are initialized only by calling `semctl` (which we describe shortly) with a command of either `SETVAL` (set one value in the set) or `SETALL` (set all the values in the set).

This requirement of two function calls to create a semaphore set (`semget`) and then initialize it (`semctl`) is a fatal flaw in the design of System V semaphores. A partial solution is to specify `IPC_CREAT | IPC_EXCL` when calling `semget`, so that only one process (the first one to call `semget`) creates the semaphore. This process then initializes the semaphore. The other processes receive an error of `EEXIST` from `semget` and they then call `semget` again, without specifying either `IPC_CREAT` or `IPC_EXCL`.

But a race condition still exists. Assume that two processes both try to create and initialize a one-member semaphore set at about the same time, both executing the following numbered lines of code:

```
1   oflag = IPC_CREAT | IPC_EXCL | SVSEM_MODE;
2   if ( (semid = semget(key, 1, oflag)) >= 0) {
            /* success, we are the first, so initialize */
3       arg.val = 1;
4       Semctl(semid, 0, SETVAL, arg);

5   } else if (errno == EEXIST) {
            /* already exists, just open */
6       semid = Semget(key, 1, SVSEM_MODE);

7   } else
8       err_sys("semget error");

9   Semop(semid, ...);     /* decrement the semaphore by 1 */
```

The following scenario could occur:

1. The first process executes lines 1–3 and is then stopped by the kernel.
2. The kernel starts the second process, which executes lines 1, 2, 5, 6, and 9.

Even though the first process to create the semaphore will be the only process to initialize the semaphore, since it takes two steps to do the creation and initialization, the kernel can switch to another process between these two steps. That other process can then use the semaphore (line 9 in the code fragment), but the semaphore value has not been initialized by the first process. The semaphore value, when the second process executes line 9, is indeterminate.

Fortunately, there is a way around this race condition. We are guaranteed that the `sem_otime` member of the `semid_ds` structure is set to 0 when a new semaphore set is created. (The System V manuals have stated this fact for a long time, as do the XPG3 and Unix 98 standards.) This member is set to the current time only by a successful call to `semop`. Therefore, the second process in the preceding example must call `semctl`

with a command of IPC_STAT after its second call to semget succeeds (line 6 in the code fragment). It then waits for sem_otime to be nonzero, at which time it knows that the semaphore has been initialized and that the process that did the initialization has successfully called semop. This means the process that creates the semaphore must initialize its value *and* must call semop before any other process can use the semaphore. We show examples of this technique in Figures 10.52 and 11.7.

Posix named semaphores avoid this problem by having one function (sem_open) create and initialize the semaphore. Furthermore, even if O_CREAT is specified, the semaphore is initialized only if it does not already exist.

Whether this potential race condition is a problem also depends on the application. With some applications (e.g., our producer–consumer as in Figure 10.21), one process always creates and initializes the semaphore. No race condition would exist in this scenario. But in other applications (e.g., our file locking example in Figure 10.19), no single process creates and initializes the semaphore: the first process to open the semaphore must create it and initialize it, and the race condition must be avoided.

11.3 semop Function

Once a semaphore set is opened with semget, operations are performed on one or more of the semaphores in the set using the semop function.

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

int semop(int semid, struct sembuf *opsptr, size_t nops);
```

Returns: 0 if OK, –1 on error

opsptr points to an array of the following structures:

```
struct sembuf {
  short   sem_num;  /* semaphore number: 0, 1, ..., nsems-1 */
  short   sem_op;   /* semaphore operation: <0, 0, >0 */
  short   sem_flg;  /* operation flags: 0, IPC_NOWAIT, SEM_UNDO */
};
```

The number of elements in the array of sembuf structures pointed to by *opsptr* is specified by the *nops* argument. Each element in this array specifies an operation for one particular semaphore value in the set. The particular semaphore value is specified by the sem_num value, which is 0 for the first element, one for the second, and so on, up to *nsems–1*, where *nsems* is the number of semaphore values in the set (the second argument in the call to semget when the semaphore set was created).

We are guaranteed only that the structure contains the three members shown. It might contain other members, and we have no guarantee that the members are in the order that we show. This means that we must not statically initialize this structure, as in

```
struct sembuf  ops[2] = {
  0, 0, 0,              /* wait for [0] to be 0 */
  0, 1, SEM_UNDO        /* then increment [0] by 1 */
};
```

but must use run-time initialization, as in

```
struct sembuf  ops[2];

ops[0].sem_num = 0;         /* wait for [0] to be 0 */
ops[0].sem_op  = 0;
ops[0].sem_flg = 0;
ops[1].sem_num = 0;         /* then increment [0] by 1 */
ops[1].sem_op  = 1;
ops[1].sem_flg = SEM_UNDO;
```

The array of operations passed to the `semop` function are guaranteed to be performed *atomically* by the kernel. The kernel either does *all* the operations that are specified, or it does *none* of them. We show an example of this in Section 11.5.

Each particular operation is specified by a `sem_op` value, which can be negative, 0, or positive. In the discussion that follows shortly, we refer to the following items:

- `semval`: the current value of the semaphore (Figure 11.1).
- `semncnt`: the number of threads waiting for `semval` to be greater than its current value (Figure 11.1).
- `semzcnt`: the number of threads waiting for `semval` to be 0 (Figure 11.1).
- `semadj`: the adjustment value for the calling process for the specified semaphore. This value is updated only if the `SEM_UNDO` flag is specified in the `sem_flg` member of the `sembuf` structure for this operation. This is a conceptual variable that is maintained by the kernel for each process that specifies the `SEM_UNDO` flag in a semaphore operation; a structure member with the name of `semadj` need not exist.
- A given semaphore operation is made nonblocking by specifying the `IPC_NOWAIT` flag in the `sem_flg` member of the `sembuf` structure. When this flag is specified and the given operation cannot be completed without putting the calling thread to sleep, `semop` returns an error of `EAGAIN`.
- When a thread is put to sleep waiting for a semaphore operation to complete (we will see that the thread can be waiting either for the semaphore value to be 0 or for the value to be greater than 0), and the thread catches a signal, and the signal handler returns, the `semop` function is interrupted and returns an error of `EINTR`. In the terminology of p. 124 of UNPv1, `semop` is a *slow system call* that is interrupted by a caught signal.
- When a thread is put to sleep waiting for a semaphore operation to complete and that semaphore is removed from the system by some other thread or process, `semop` returns an error of `EIDRM` ("identifier removed").

We now describe the operation of `semop`, based on the three possible values of each specified `sem_op` operation: positive, 0, or negative.

1. If `sem_op` is positive, the value of `sem_op` is added to `semval`. This corresponds to the release of resources that a semaphore controls.

If the SEM_UNDO flag is specified, the value of sem_op is subtracted from the semaphore's semadj value.

2. If sem_op is 0, the caller wants to wait until semval is 0. If semval is already 0, return is made immediately.

   If semval is nonzero, the semaphore's semzcnt value is incremented and the calling thread is blocked until semval becomes 0 (at which time, the semaphore's semzcnt value is decremented). As mentioned earlier, the thread is not put to sleep if IPC_NOWAIT is specified. The sleep returns prematurely with an error if a caught signal interrupts the function or if the semaphore is removed.

3. If sem_op is negative, the caller wants to wait until the semaphore's value becomes greater than or equal to the absolute value of sem_op. This corresponds to the allocation of resources.

   If semval is greater than or equal to the absolute value of sem_op, the absolute value of sem_op is subtracted from semval. If the SEM_UNDO flag is specified, the absolute value of sem_op is added to the semaphore's semadj value.

   If semval is less than the absolute value of sem_op, the semaphore's semncnt value is incremented and the calling thread is blocked until semval becomes greater than or equal to the absolute value of sem_op. When this change occurs, the thread is unblocked, the absolute value of sem_op is subtracted from semval, and the semaphore's semncnt value is decremented. If the SEM_UNDO flag is specified, the absolute value of sem_op is added to the semaphore's semadj value. As mentioned earlier, the thread is not put to sleep if IPC_NOWAIT is specified. Also, the sleep returns prematurely with an error if a caught signal interrupts the function or if the semaphore is removed.

If we compare these operations to the operations allowed on a Posix semaphore, the latter allows operations of only –1 (sem_wait) and +1 (sem_post). System V semaphores allow the value to go up or down by increments other than one, and also allow waiting for the semaphore value to be 0. These more general operations, along with the fact that System V semaphores can have a set of values, is what complicates System V semaphores, compared to the simpler Posix semaphores.

11.4 semctl Function

The semctl function performs various control operations on a semaphore.

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

int semctl(int semid, int semnum, int cmd, ... /* union semun arg */ );
```

Returns: nonnegative value if OK (see text), –1 on error

The first argument *semid* identifies the semaphore, and *semnum* identifies the member of the semaphore set (0, 1, and so on, up to *nsems–1*). The *semnum* value is used only for the GETVAL, SETVAL, GETNCNT, GETZCNT, and GETPID commands.

The fourth argument is optional, depending on the *cmd* (see the comments in the union below). When required, it is the following union:

```
union semun {
  int               val;    /* used for SETVAL only */
  struct semid_ds  *buf;    /* used for IPC_SET and IPC_STAT */
  ushort           *array;  /* used for GETALL and SETALL */
};
```

This union does not appear in any system header and must be declared by the application. (We define it in our unpipc.h header, Figure C.1.) It is passed by value, not by reference. That is, the actual value of the union is the argument, not a pointer to the union.

> Unfortunately, some systems (FreeBSD and Linux) define this union as a result of including the <sys/sem.h> header, making it hard to write portable code. Even though having the system header declare this union makes sense, Unix 98 states that it must be explicitly declared by the application.

The following values for the *cmd* are supported. Unless stated otherwise, a return value of 0 indicates success, and a return value of –1 indicates an error.

GETVAL	Return the current value of semval as the return value of the function. Since a semaphore value is never negative (semval is declared as an unsigned short), a successful return value is always nonnegative.
SETVAL	Set the value of semval to *arg.val*. If this is successful, the semaphore adjustment value for this semaphore is set to 0 in all processes.
GETPID	Return the current value of sempid as the return value of the function.
GETNCNT	Return the current value of semncnt as the return value of the function.
GETZCNT	Return the current value of semzcnt as the return value of the function.
GETALL	Return the values of semval for each member of the semaphore set. The values are returned through the *arg.array* pointer, and the return value of the function is 0. Notice that the caller must allocate an array of unsigned short integers large enough to hold all the values for the set, and then set *arg.array* to point to this array.
SETALL	Set the values of semval for each member of the semaphore set. The values are specified through the *arg.array* pointer.
IPC_RMID	Remove the semaphore set specified by *semid* from the system.
IPC_SET	Set the following three members of the semid_ds structure for the semaphore set from the corresponding members in the structure pointed to by the *arg.buf* argument: sem_perm.uid, sem_perm.gid,

and `sem_perm.mode`. The `sem_ctime` member of the `semid_ds` structure is also set to the current time.

`IPC_STAT` Return to the caller (through the *arg.buf* argument) the current `semid_ds` structure for the specified semaphore set. Notice that the caller must first allocate a `semid_ds` structure and set *arg.buf* to point to this structure.

11.5 Simple Programs

Since System V semaphores have kernel persistence, we can demonstrate their usage by writing a small set of programs to manipulate them and seeing what happens. The values of the semaphores will be maintained by the kernel from one of our programs to the next.

`semcreate` Program

Our first program shown in Figure 11.2 just creates a System V semaphore set. The `-e` command-line option specifies the `IPC_EXCL` flag, and the number of semaphores in the set must be specified by the final command-line argument.

svsem/semcreate.c

```
1 #include    "unpipc.h"
2 int
3 main(int argc, char **argv)
4 {
5     int     c, oflag, semid, nsems;
6     oflag = SVSEM_MODE | IPC_CREAT;
7     while ( (c = Getopt(argc, argv, "e")) != -1) {
8         switch (c) {
9         case 'e':
10            oflag |= IPC_EXCL;
11            break;
12        }
13    }
14    if (optind != argc - 2)
15        err_quit("usage: semcreate [ -e ] <pathname> <nsems>");
16    nsems = atoi(argv[optind + 1]);
17    semid = Semget(Ftok(argv[optind], 1), nsems, oflag);
18    exit(0);
19 }
```

svsem/semcreate.c

Figure 11.2 `semcreate` program.

`semrmid` Program

The next program, shown in Figure 11.3, removes a semaphore set from the system. A command of `IPC_RMID` is executed through the `semctl` function to remove the set.

semsetvalues Program

Our semsetvalues program (Figure 11.4) sets all the values in a semaphore set.

Get number of semaphores in set

11-15 After obtaining the semaphore ID with semget, we issue an IPC_STAT command to semctl to fetch the semid_ds structure for the semaphore. The sem_nsems member is the number of semaphores in the set.

Set all the values

19-24 We allocate memory for an array of unsigned shorts, one per set member, and copy the values from the command-line into the array. A command of SETALL to semctl sets all the values in the semaphore set.

semgetvalues Program

Figure 11.5 shows our semgetvalues program, which fetches and prints all the values in a semaphore set.

Get number of semaphores in set

11-15 After obtaining the semaphore ID with semget, we issue an IPC_STAT command to semctl to fetch the semid_ds structure for the semaphore. The sem_nsems member is the number of semaphores in the set.

Get all the values

16-22 We allocate memory for an array of unsigned shorts, one per set member, and issue a command of GETALL to semctl to fetch all the values in the semaphore set. Each value is printed.

semops Program

Our semops program, shown in Figure 11.6, executes an array of operations on a semaphore set.

Command-line options

7-19 An option of -n specifies the IPC_NOWAIT flag for each operation, and an option of -u specifies the SEM_UNDO flag for each operation. Note that the semop function allows us to specify a different set of flags for each member of the sembuf structure (that is, for the operation on each member of the set), but for simplicity we have these command-line options specify that flag for all specified operations.

Allocate memory for the operations

20-29 After opening the semaphore set with semget, an array of sembuf structures is allocated, one element for each operation specified on the command line. Unlike the previous two programs, this program allows the user to specify fewer operations than members of the semaphore set.

Execute the operations

30 semop executes the array of operations on the semaphore set.

svsem/semrmid.c

```
 1 #include    "unpipc.h"

 2 int
 3 main(int argc, char **argv)
 4 {
 5     int     semid;

 6     if (argc != 2)
 7         err_quit("usage: semrmid <pathname>");

 8     semid = Semget(Ftok(argv[1], 1), 0, 0);
 9     Semctl(semid, 0, IPC_RMID);

10     exit(0);
11 }
```

svsem/semrmid.c

Figure 11.3 semrmid program.

svsem/semsetvalues.c

```
 1 #include    "unpipc.h"

 2 int
 3 main(int argc, char **argv)
 4 {
 5     int     semid, nsems, i;
 6     struct semid_ds seminfo;
 7     unsigned short *ptr;
 8     union semun arg;

 9     if (argc < 2)
10         err_quit("usage: semsetvalues <pathname> [ values ... ]");

11         /* first get the number of semaphores in the set */
12     semid = Semget(Ftok(argv[1], 1), 0, 0);
13     arg.buf = &seminfo;
14     Semctl(semid, 0, IPC_STAT, arg);
15     nsems = arg.buf->sem_nsems;

16         /* now get the values from the command line */
17     if (argc != nsems + 2)
18         err_quit("%d semaphores in set, %d values specified", nsems, argc - 2);

19         /* allocate memory to hold all the values in the set, and store */
20     ptr = Calloc(nsems, sizeof(unsigned short));
21     arg.array = ptr;
22     for (i = 0; i < nsems; i++)
23         ptr[i] = atoi(argv[i + 2]);
24     Semctl(semid, 0, SETALL, arg);

25     exit(0);
26 }
```

svsem/semsetvalues.c

Figure 11.4 semsetvalues program.

svsem/semgetvalues.c

```
 1 #include    "unpipc.h"
 2 int
 3 main(int argc, char **argv)
 4 {
 5     int     semid, nsems, i;
 6     struct semid_ds seminfo;
 7     unsigned short *ptr;
 8     union semun arg;
 9     if (argc != 2)
10         err_quit("usage: semgetvalues <pathname>");
11         /* first get the number of semaphores in the set */
12     semid = Semget(Ftok(argv[1], 1), 0, 0);
13     arg.buf = &seminfo;
14     Semctl(semid, 0, IPC_STAT, arg);
15     nsems = arg.buf->sem_nsems;
16         /* allocate memory to hold all the values in the set */
17     ptr = Calloc(nsems, sizeof(unsigned short));
18     arg.array = ptr;
19         /* fetch the values and print */
20     Semctl(semid, 0, GETALL, arg);
21     for (i = 0; i < nsems; i++)
22         printf("semval[%d] = %d\n", i, ptr[i]);
23     exit(0);
24 }
```

svsem/semgetvalues.c

Figure 11.5 `semgetvalues` program.

svsem/semops.c

```
 1 #include    "unpipc.h"
 2 int
 3 main(int argc, char **argv)
 4 {
 5     int     c, i, flag, semid, nops;
 6     struct sembuf *ptr;
 7     flag = 0;
 8     while ( (c = Getopt(argc, argv, "nu")) != -1) {
 9         switch (c) {
10         case 'n':
11             flag |= IPC_NOWAIT; /* for each operation */
12             break;
13         case 'u':
14             flag |= SEM_UNDO;   /* for each operation */
15             break;
16         }
17     }
18     if (argc - optind < 2)      /* argc - optind = #args remaining */
19         err_quit("usage: semops [ -n ] [ -u ] <pathname> operation ...");
```

```
20      semid = Semget(Ftok(argv[optind], 1), 0, 0);
21      optind++;
22      nops = argc - optind;

23          /* allocate memory to hold operations, store, and perform */
24      ptr = Calloc(nops, sizeof(struct sembuf));
25      for (i = 0; i < nops; i++) {
26          ptr[i].sem_num = i;
27          ptr[i].sem_op = atoi(argv[optind + i]);      /* <0, 0, or >0 */
28          ptr[i].sem_flg = flag;
29      }
30      Semop(semid, ptr, nops);

31      exit(0);
32  }
```

svsem/semops.c

Figure 11.6 semops program.

Examples

We now demonstrate the five programs that we have just shown, looking at some of the features of System V semaphores.

```
solaris % touch /tmp/rich
solaris % semcreate -e /tmp/rich 3
solaris % semsetvalues /tmp/rich 1 2 3
solaris % semgetvalues /tmp/rich
semval[0] = 1
semval[1] = 2
semval[2] = 3
```

We first create a file named `/tmp/rich` that will be used (by `ftok`) to identify the semaphore set. `semcreate` creates a set with three members. `semsetvalues` sets the values to 1, 2, and 3, and these values are then printed by `semgetvalues`.

We now demonstrate the atomicity of the set of operations when performed on a semaphore set.

```
solaris % semops -n /tmp/rich -1 -2 -4
semctl error: Resource temporarily unavailable
solaris % semgetvalues /tmp/rich
semval[0] = 1
semval[1] = 2
semval[2] = 3
```

We specify the nonblocking flag (`-n`) and three operations, each of which decrements a value in the set. The first operation is OK (we can subtract 1 from the first member of the set whose value is 1), the second operation is OK (we can subtract 2 from the second member of the set whose value is 2), but the third operation cannot be performed (we cannot subtract 4 from the third member of the set whose value is 3). Since the last operation cannot be performed, and since we specified nonblocking, an error of `EAGAIN` is returned. (Had we not specified the nonblocking flag, our program would have just blocked.) We then verify that none of the values in the set were changed. Even though

the first two operations could be performed, since the final operation could not be performed, none of the three operations are performed. The atomicity of `semop` means that either *all* of the operations are performed or *none* of the operations are performed.

We now demonstrate the `SEM_UNDO` property of System V semaphores.

```
solaris % semsetvalues /tmp/rich 1 2 3      set to known values
solaris % semops -u /tmp/rich -1 -2 -3      specify SEM_UNDO for each operation
solaris % semgetvalues /tmp/rich
semval[0] = 1                               all the changes were undone when semops terminated
semval[1] = 2
semval[2] = 3
solaris % semops /tmp/rich -1 -2 -3         do not specify SEM_UNDO
solaris % semgetvalues /tmp/rich
semval[0] = 0                               the changes were not undone
semval[1] = 0
semval[2] = 0
```

We first reset the three values to 1, 2, and 3 with `semsetvalues` and then specify operations of –1, –2, and –3 with our `semops` program. This causes all three values to become 0, but since we specify the `-u` flag to our `semops` program, the `SEM_UNDO` flag is specified for each of the three operations. This causes the `semadj` value for the three members to be set to 1, 2, and 3, respectively. Then when our `semops` program terminates, these three `semadj` values are added back to the current values of each of the three members (which are all 0), causing their final values to be 1, 2, and 3, as we verify with our `semgetvalues` program. We then execute our `semops` program again, but without the `-u` flag, and this leaves the three values at 0 when our `semops` program terminates.

11.6 File Locking

We can provide a version of our `my_lock` and `my_unlock` functions from Figure 10.19, implemented using System V semaphores. We show this in Figure 11.7.

First try an exclusive create

13-17 We must guarantee that only one process initializes the semaphore, so we specify `IPC_CREAT | IPC_EXCL`. If this succeeds, that process calls `semctl` to initialize the semaphore value to 1. If we start multiple processes at about the same time, each of which calls our `my_lock` function, only one will create the semaphore (assuming it does not already exist), and then that process initializes the semaphore too.

Semaphore already exists; just open

18-20 The first call to `semget` will return an error of `EEXIST` to the other processes, which then call `semget` again, but without the `IPC_CREAT | IPC_EXCL` flags.

Wait for semaphore to be initialized

21-28 We encounter the same race condition that we talked about with the initialization of System V semaphores in Section 11.2. To avoid this, any process that finds that the semaphore already exists must call `semctl` with a command of `IPC_STAT` to look at

lock/locksvsem.c

```
 1 #include    "unpipc.h"

 2 #define LOCK_PATH    "/tmp/svsemlock"
 3 #define MAX_TRIES    10

 4 int     semid, initflag;
 5 struct sembuf postop, waitop;

 6 void
 7 my_lock(int fd)
 8 {
 9     int     oflag, i;
10     union semun arg;
11     struct semid_ds seminfo;

12     if (initflag == 0) {
13         oflag = IPC_CREAT | IPC_EXCL | SVSEM_MODE;
14         if ( (semid = semget(Ftok(LOCK_PATH, 1), 1, oflag)) >= 0) {
15                 /* success, we're the first so initialize */
16             arg.val = 1;
17             Semctl(semid, 0, SETVAL, arg);

18         } else if (errno == EEXIST) {
19                 /* someone else has created; make sure it's initialized */
20             semid = Semget(Ftok(LOCK_PATH, 1), 1, SVSEM_MODE);
21             arg.buf = &seminfo;
22             for (i = 0; i < MAX_TRIES; i++) {
23                 Semctl(semid, 0, IPC_STAT, arg);
24                 if (arg.buf->sem_otime != 0)
25                     goto init;
26                 sleep(1);
27             }
28             err_quit("semget OK, but semaphore not initialized");

29         } else
30             err_sys("semget error");
31       init:
32         initflag = 1;
33         postop.sem_num = 0;      /* and init the two semop() structures */
34         postop.sem_op = 1;
35         postop.sem_flg = SEM_UNDO;
36         waitop.sem_num = 0;
37         waitop.sem_op = -1;
38         waitop.sem_flg = SEM_UNDO;
39     }
40     Semop(semid, &waitop, 1);    /* down by 1 */
41 }

42 void
43 my_unlock(int fd)
44 {
45     Semop(semid, &postop, 1);    /* up by 1 */
46 }
```

lock/locksvsem.c

Figure 11.7 File locking using System V semaphores.

the sem_otime value for the semaphore. Once this value is nonzero, we know that the process that created the semaphore has initialized it, and has called semop (the call to semop is at the end of this function). If the value is still 0 (which should happen very infrequently), we sleep for 1 second and try again. We limit the number of times that we try this, to avoid sleeping forever.

Initialize sembuf structures

33-38 As we mentioned earlier, there is no guaranteed order of the members in the sembuf structure, so we cannot statically initialize them. Instead, we allocate two of these structures and fill them in at run time, when the process calls my_lock for the first time. We specify the SEM_UNDO flag, so that if a process terminates while holding the lock, the kernel will release the lock (see Exercise 10.3).

Creating a semaphore on its first use is easy (each process tries to create it but ignores an error if the semaphore already exists), but removing it after all the processes are done is much harder. In the case of a printer daemon that uses the sequence number file to assign job numbers, the semaphore would remain in existence all the time. But other applications might want to delete the semaphore when the file is deleted. In this case, a record lock might be better than a semaphore.

11.7 Semaphore Limits

As with System V message queues, there are certain system limits with System V semaphores, most of which arise from their original System V implementation (Section 3.8). These are shown in Figure 11.8. The first column is the traditional System V name for the kernel variable that contains this limit.

Name	Description	DUnix 4.0B	Solaris 2.6
semmni	max # unique semaphore sets, systemwide	16	10
semmsl	max # semaphores per semaphore set	25	25
semmns	max # semaphores, systemwide	400	60
semopm	max # operations per semop call	10	10
semmnu	max # of undo structures, systemwide		30
semume	max # of undo entries per undo structure	10	10
semvmx	max value of any semaphore	32767	32767
semaem	max adjust-on-exit value	16384	16384

Figure 11.8 Typical limits for System V semaphores.

Apparently no semmnu limit exists for Digital Unix.

Example

The program in Figure 11.9 determines the limits shown in Figure 11.8.

svsem/limits.c

```
 1 #include    "unpipc.h"
 2         /* following are upper limits of values to try */
 3 #define MAX_NIDS    4096        /* max # semaphore IDs */
 4 #define MAX_VALUE   1024*1024   /* max semaphore value */
 5 #define MAX_MEMBERS 4096        /* max # semaphores per semaphore set */
 6 #define MAX_NOPS    4096        /* max # operations per semop() */
 7 #define MAX_NPROC   Sysconf(_SC_CHILD_MAX)
 8 int
 9 main(int argc, char **argv)
10 {
11     int     i, j, semid, sid[MAX_NIDS], pipefd[2];
12     int     semmni, semvmx, semmsl, semmns, semopn, semaem, semume, semmnu;
13     pid_t  *child;
14     union semun arg;
15     struct sembuf ops[MAX_NOPS];
16         /* see how many sets with one member we can create */
17     for (i = 0; i <= MAX_NIDS; i++) {
18         sid[i] = semget(IPC_PRIVATE, 1, SVSEM_MODE | IPC_CREAT);
19         if (sid[i] == -1) {
20             semmni = i;
21             printf("%d identifiers open at once\n", semmni);
22             break;
23         }
24     }
25         /* before deleting, find maximum value using sid[0] */
26     for (j = 7; j < MAX_VALUE; j += 8) {
27         arg.val = j;
28         if (semctl(sid[0], 0, SETVAL, arg) == -1) {
29             semvmx = j - 8;
30             printf("max semaphore value = %d\n", semvmx);
31             break;
32         }
33     }
34     for (j = 0; j < i; j++)
35         Semctl(sid[j], 0, IPC_RMID);
36         /* determine max # semaphores per semaphore set */
37     for (i = 1; i <= MAX_MEMBERS; i++) {
38         semid = semget(IPC_PRIVATE, i, SVSEM_MODE | IPC_CREAT);
39         if (semid == -1) {
40             semmsl = i - 1;
41             printf("max of %d members per set\n", semmsl);
42             break;
43         }
44         Semctl(semid, 0, IPC_RMID);
45     }
46         /* find max of total # of semaphores we can create */
47     semmns = 0;
48     for (i = 0; i < semmni; i++) {
49         sid[i] = semget(IPC_PRIVATE, semmsl, SVSEM_MODE | IPC_CREAT);
50         if (sid[i] == -1) {
```

```
51              /*
52               * Up to this point each set has been created with semmsl
53               * members.  But this just failed, so try recreating this
54               * final set with one fewer member per set, until it works.
55               */
56              for (j = semmsl - 1; j > 0; j--) {
57                  sid[i] = semget(IPC_PRIVATE, j, SVSEM_MODE | IPC_CREAT);
58                  if (sid[i] != -1) {
59                      semmns += j;
60                      printf("max of %d semaphores\n", semmns);
61                      Semctl(sid[i], 0, IPC_RMID);
62                      goto done;
63                  }
64              }
65              err_quit("j reached 0, semmns = %d", semmns);
66          }
67          semmns += semmsl;
68      }
69      printf("max of %d semaphores\n", semmns);
70  done:
71      for (j = 0; j < i; j++)
72          Semctl(sid[j], 0, IPC_RMID);
73          /* see how many operations per semop() */
74      semid = Semget(IPC_PRIVATE, semmsl, SVSEM_MODE | IPC_CREAT);
75      for (i = 1; i <= MAX_NOPS; i++) {
76          ops[i - 1].sem_num = i - 1;
77          ops[i - 1].sem_op = 1;
78          ops[i - 1].sem_flg = 0;
79          if (semop(semid, ops, i) == -1) {
80              if (errno != E2BIG)
81                  err_sys("expected E2BIG from semop");
82              semopn = i - 1;
83              printf("max of %d operations per semop()\n", semopn);
84              break;
85          }
86      }
87      Semctl(semid, 0, IPC_RMID);
88          /* determine the max value of semadj */
89          /* create one set with one semaphore */
90      semid = Semget(IPC_PRIVATE, 1, SVSEM_MODE | IPC_CREAT);
91      arg.val = semvmx;
92      Semctl(semid, 0, SETVAL, arg);  /* set value to max */
93      for (i = semvmx - 1; i > 0; i--) {
94          ops[0].sem_num = 0;
95          ops[0].sem_op = -i;
96          ops[0].sem_flg = SEM_UNDO;
97          if (semop(semid, ops, 1) != -1) {
98              semaem = i;
99              printf("max value of adjust-on-exit = %d\n", semaem);
100             break;
101         }
102     }
103     Semctl(semid, 0, IPC_RMID);
```

```
104         /* determine max # undo structures */
105         /* create one set with one semaphore; init to 0 */
106     semid = Semget(IPC_PRIVATE, 1, SVSEM_MODE | IPC_CREAT);
107     arg.val = 0;
108     Semctl(semid, 0, SETVAL, arg);  /* set semaphore value to 0 */
109     Pipe(pipefd);
110     child = Malloc(MAX_NPROC * sizeof(pid_t));
111     for (i = 0; i < MAX_NPROC; i++) {
112         if ( (child[i] = fork()) == -1) {
113             semmnu = i - 1;
114             printf("fork failed, semmnu at least %d\n", semmnu);
115             break;
116         } else if (child[i] == 0) {
117             ops[0].sem_num = 0; /* child does the semop() */
118             ops[0].sem_op = 1;
119             ops[0].sem_flg = SEM_UNDO;
120             j = semop(semid, ops, 1);   /* 0 if OK, -1 if error */
121             Write(pipefd[1], &j, sizeof(j));
122             sleep(30);          /* wait to be killed by parent */
123             exit(0);            /* just in case */
124         }
125         /* parent reads result of semop() */
126         Read(pipefd[0], &j, sizeof(j));
127         if (j == -1) {
128             semmnu = i;
129             printf("max # undo structures = %d\n", semmnu);
130             break;
131         }
132     }
133     Semctl(semid, 0, IPC_RMID);
134     for (j = 0; j <= i && child[j] > 0; j++)
135         Kill(child[j], SIGINT);

136         /* determine max # adjust entries per process */
137         /* create one set with max # of semaphores */
138     semid = Semget(IPC_PRIVATE, semmsl, SVSEM_MODE | IPC_CREAT);
139     for (i = 0; i < semmsl; i++) {
140         arg.val = 0;
141         Semctl(semid, i, SETVAL, arg);  /* set semaphore value to 0 */

142         ops[i].sem_num = i;
143         ops[i].sem_op = 1;      /* add 1 to the value */
144         ops[i].sem_flg = SEM_UNDO;
145         if (semop(semid, ops, i + 1) == -1) {
146             semume = i;
147             printf("max # undo entries per process = %d\n", semume);
148             break;
149         }
150     }
151     Semctl(semid, 0, IPC_RMID);

152     exit(0);
153 }
```

svsem/limits.c

Figure 11.9 Determine the system limits on System V semaphores.

11.8 Summary

The following changes occur when moving from Posix semaphores to System V semaphores:

1. System V semaphores consist of a set of values. When specifying a group of semaphore operations to apply to a set, either all of the operations are performed or none of the operations are performed.
2. Three operations may be applied to each member of a semaphore set: test for the value being 0, add an integer to the value, and subtract an integer from the value (assuming that the value remains nonnegative). The only operations allowed for a Posix semaphore are to increment by one and to decrement by one (assuming that the value remains nonnegative).
3. Creating a System V semaphore set is tricky because it requires two operations to create the set and then initialize the values, which can lead to race conditions.
4. System V semaphores provide an "undo" feature that reverses a semaphore operation upon process termination.

Exercises

11.1 Figure 6.8 was a modification to Figure 6.6 that accepted an identifier instead of a pathname to specify the queue. We showed that the identifier is all we need to know to access a System V message queue (assuming we have adequate permission). Make similar modifications to Figure 11.6 and show that the same feature applies to System V semaphores.

11.2 What happens in Figure 11.7 if the `LOCK_PATH` file does not exist?

Part 4

Shared Memory

12

Shared Memory Introduction

12.1 Introduction

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

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

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

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

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

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

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

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

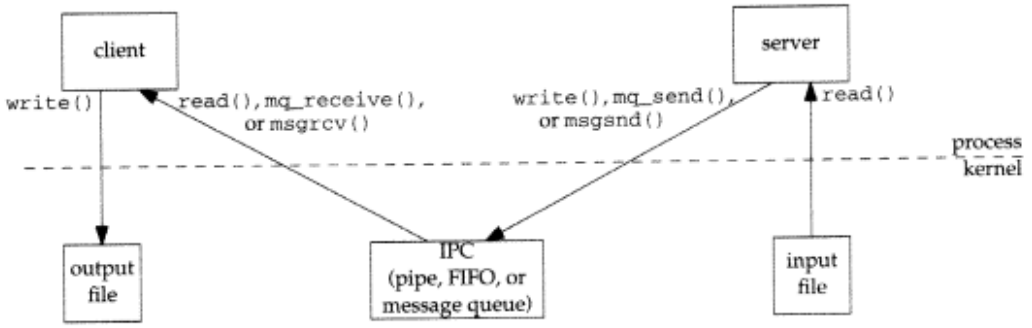

Figure 12.1 Flow of file data from server to client.

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

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

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

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

This scenario is depicted in Figure 12.2.

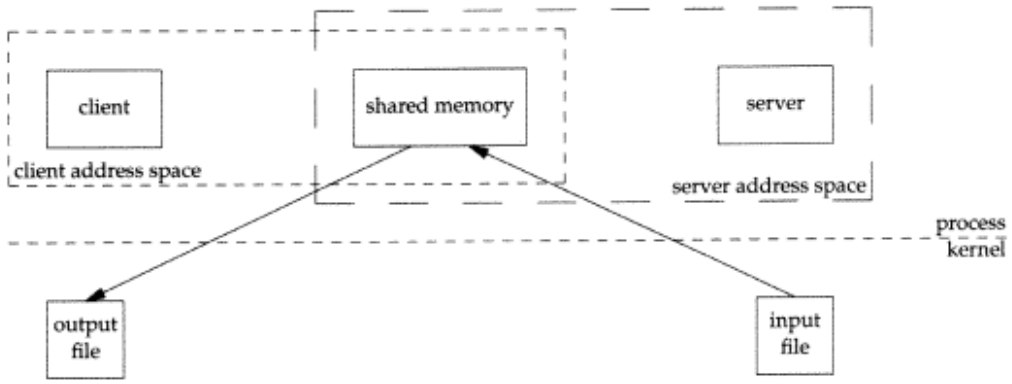

Figure 12.2 Copying file data from server to client using shared memory.

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

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

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

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

Create and initialize semaphore

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

Set standard output unbuffered and `fork`

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

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

shm/incr1.c

```
 1 #include    "unpipc.h"
 2 #define SEM_NAME    "mysem"
 3 int     count = 0;
 4 int
 5 main(int argc, char **argv)
 6 {
 7     int     i, nloop;
 8     sem_t  *mutex;
 9     if (argc != 2)
10         err_quit("usage: incr1 <#loops>");
11     nloop = atoi(argv[1]);
12         /* create, initialize, and unlink semaphore */
13     mutex = Sem_open(Px_ipc_name(SEM_NAME), O_CREAT | O_EXCL, FILE_MODE, 1);
14     Sem_unlink(Px_ipc_name(SEM_NAME));
15     setbuf(stdout, NULL);       /* stdout is unbuffered */
16     if (Fork() == 0) {          /* child */
17         for (i = 0; i < nloop; i++) {
18             Sem_wait(mutex);
19             printf("child: %d\n", count++);
20             Sem_post(mutex);
21         }
22         exit(0);
23     }
24         /* parent */
25     for (i = 0; i < nloop; i++) {
26         Sem_wait(mutex);
27         printf("parent: %d\n", count++);
28         Sem_post(mutex);
29     }
30     exit(0);
31 }
```

shm/incr1.c

Figure 12.3 Parent and child both increment the same global.

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

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

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

                      and so on
```

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

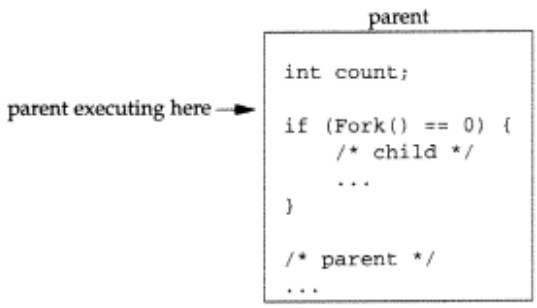

Figure 12.4 Parent before calling `fork`.

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

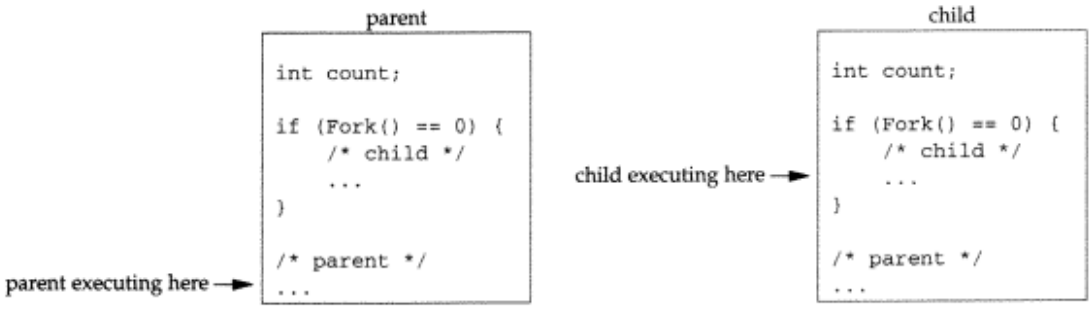

Figure 12.5 Parent and child after `fork` returns.

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

12.2 mmap, munmap, and msync Functions

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

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

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

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

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

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

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

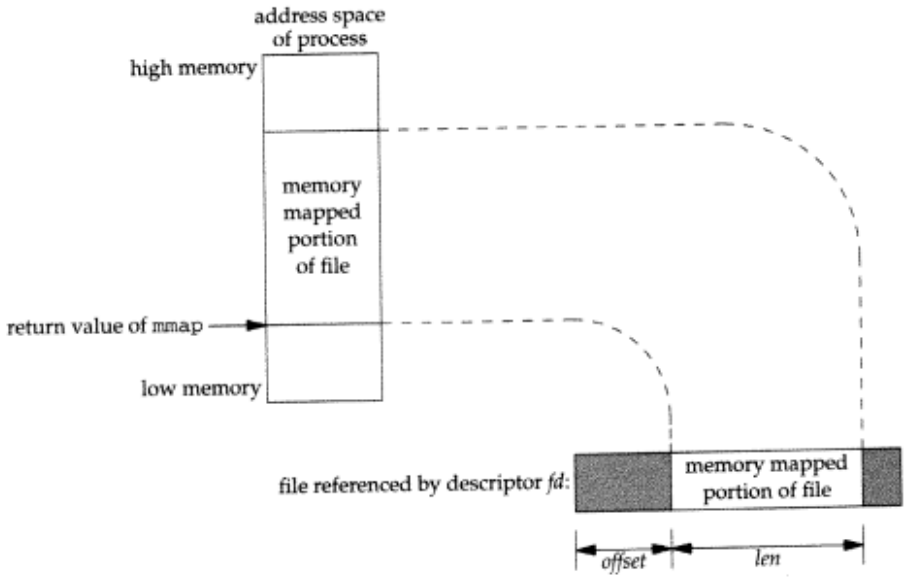

Figure 12.6 Example of memory-mapped file.

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

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

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

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

Figure 12.8 *flags* argument for mmap.

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

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

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

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

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

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

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

Returns: 0 if OK, –1 on error

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

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

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

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

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

Returns: 0 if OK, –1 on error

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

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

Figure 12.9 *flags* for msync function.

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

Why Use mmap?

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

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

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

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

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

12.3 Increment Counter in a Memory-Mapped File

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

New command-line argument

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

`mmap` then close descriptor

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

`fork`

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

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

shm/incr2.c

```
1 #include    "unpipc.h"
2 #define SEM_NAME    "mysem"
3 int
4 main(int argc, char **argv)
5 {
6     int     fd, i, nloop, zero = 0;
7     int    *ptr;
8     sem_t  *mutex;
9     if (argc != 3)
10        err_quit("usage: incr2 <pathname> <#loops>");
11    nloop = atoi(argv[2]);
12        /* open file, initialize to 0, map into memory */
13    fd = Open(argv[1], O_RDWR | O_CREAT, FILE_MODE);
14    Write(fd, &zero, sizeof(int));
15    ptr = Mmap(NULL, sizeof(int), PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
16    Close(fd);
17        /* create, initialize, and unlink semaphore */
18    mutex = Sem_open(Px_ipc_name(SEM_NAME), O_CREAT | O_EXCL, FILE_MODE, 1);
19    Sem_unlink(Px_ipc_name(SEM_NAME));
20    setbuf(stdout, NULL);       /* stdout is unbuffered */
21    if (Fork() == 0) {          /* child */
22        for (i = 0; i < nloop; i++) {
23            Sem_wait(mutex);
24            printf("child: %d\n", (*ptr)++);
25            Sem_post(mutex);
26        }
27        exit(0);
28    }
29        /* parent */
30    for (i = 0; i < nloop; i++) {
31        Sem_wait(mutex);
32        printf("parent: %d\n", (*ptr)++);
33        Sem_post(mutex);
34    }
35    exit(0);
36 }
```

shm/incr2.c

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

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

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

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

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

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

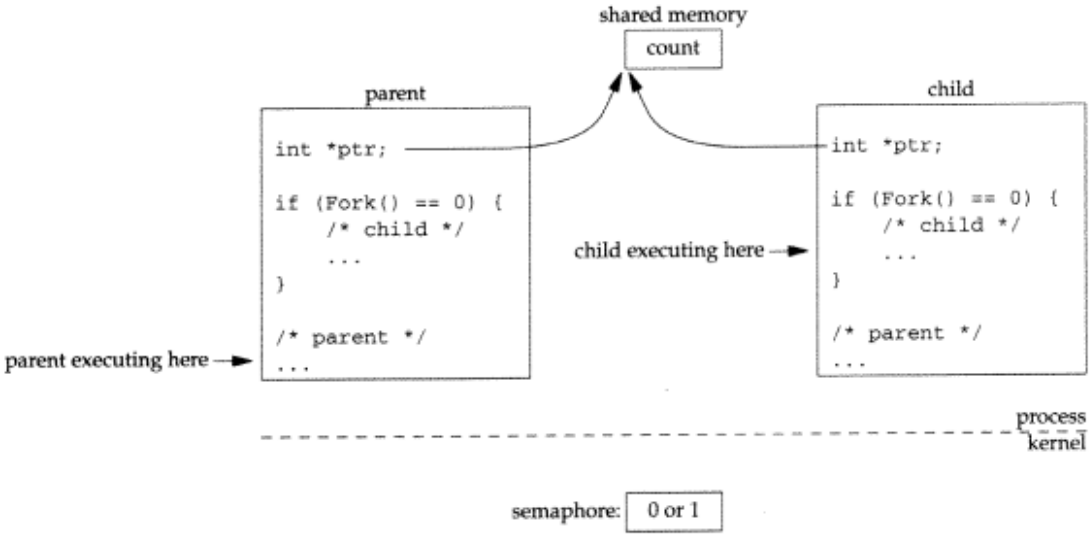

Figure 12.11 Parent and child sharing memory and a semaphore.

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

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

shm/incr3.c

```
1 #include    "unpipc.h"

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

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

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

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

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

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

shm/incr3.c

Figure 12.12 Counter and semaphore are both in shared memory.

Define structure that will be in shared memory

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

Map the memory

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

Initialize semaphore

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

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

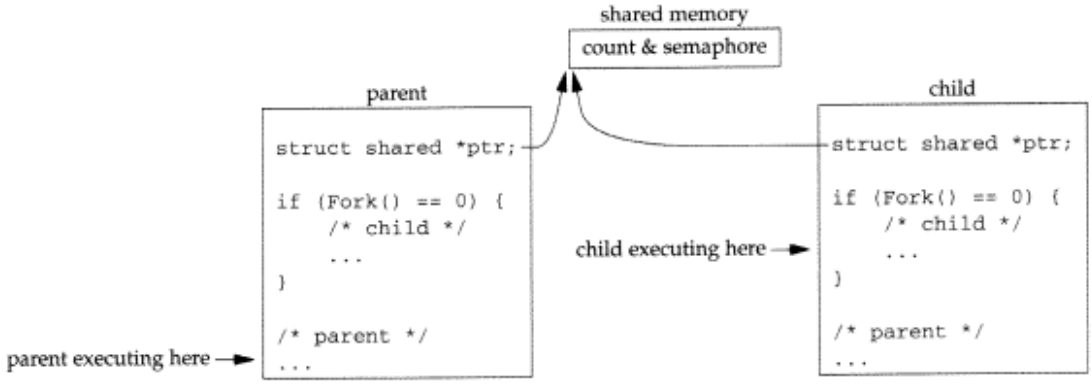

Figure 12.13 Counter and semaphore are now in shared memory.

12.4 4.4BSD Anonymous Memory Mapping

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

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

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

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

shm/incr_map_anon.c

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

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

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

shm/incr_map_anon.c

Figure 12.14 4.4BSD anonymous memory mapping.

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

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

12.5 SVR4 `/dev/zero` Memory Mapping

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

shm/incr_dev_zero.c

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

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

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

shm/incr_dev_zero.c

Figure 12.15 SVR4 memory mapping of `/dev/zero`.

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

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

12.6 Referencing Memory-Mapped Objects

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

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

shm/test1.c

```
1 #include    "unpipc.h"

2 int
3 main(int argc, char **argv)
4 {
5     int     fd, i;
6     char   *ptr;
7     size_t  filesize, mmapsize, pagesize;

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

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

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

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

20    for (i = 0; i < max(filesize, mmapsize); i += pagesize) {
21        printf("ptr[%d] = %d\n", i, ptr[i]);
22        ptr[i] = 1;
23        printf("ptr[%d] = %d\n", i + pagesize - 1, ptr[i + pagesize - 1]);
24        ptr[i + pagesize - 1] = 1;
25    }
26    printf("ptr[%d] = %d\n", i, ptr[i]);

27    exit(0);
28 }
```

shm/test1.c

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

Command-line arguments

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

Create, open, truncate file; set file size

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

Memory map file

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

Print page size

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

Read and store the memory-mapped region

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

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

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

The page size is 4096 bytes, and we are able to read the entire second page (indexes 4096 through 8191), but a reference to the third page (index 8192) generates `SIGSEGV`, which

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

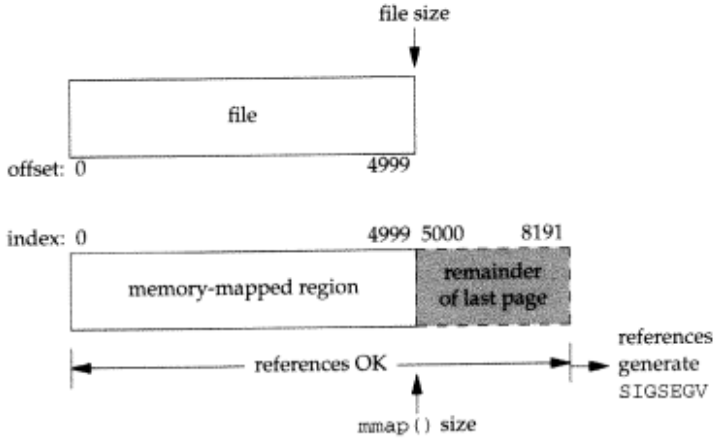

Figure 12.17 Memory mapping when `mmap` size equals file size.

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

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

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

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

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

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

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

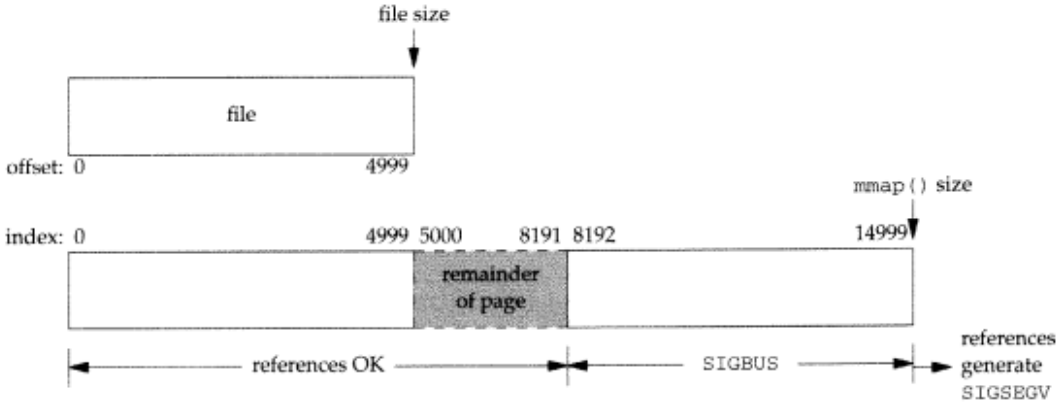

Figure 12.18 Memory mapping when `mmap` size exceeds file size.

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

Open file

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

Increase file size

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

shm/test2.c

```
1 #include    "unpipc.h"

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

4 int
5 main(int argc, char **argv)
6 {
7     int     fd, i;
8     char   *ptr;

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

12    for (i = 4096; i <= SIZE; i += 4096) {
13        printf("setting file size to %d\n", i);
14        Ftruncate(fd, i);
15        printf("ptr[%d] = %d\n", i - 1, ptr[i - 1]);
16    }

17    exit(0);
18 }
```

shm/test2.c

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

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

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

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

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

12.7 Summary

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

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

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

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

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

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

Exercises

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

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

implementation of Posix message queues from Section 5.8 is used, and draw a diagram of the transfer of messages.

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

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

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

13

Posix Shared Memory

13.1 Introduction

The previous chapter described shared memory in general terms, along with the `mmap` function. Examples were shown that used `mmap` to provide shared memory between a parent and child:

- using a memory-mapped file (Figure 12.10),
- using 4.4BSD anonymous memory mapping (Figure 12.14), and
- using `/dev/zero` anonymous memory mapping (Figure 12.15).

We now extend the concept of shared memory to include memory that is shared between unrelated processes. Posix.1 provides two ways to share memory between unrelated processes.

1. *Memory-mapped files*: a file is opened by `open`, and the resulting descriptor is mapped into the address space of the process by `mmap`. We described this technique in Chapter 12 and showed its use when sharing memory between a parent and child. Memory-mapped files can also be shared between unrelated processes.
2. *Shared memory objects*: the function `shm_open` opens a Posix.1 IPC name (perhaps a pathname in the filesystem), returning a descriptor that is then mapped into the address space of the process by `mmap`. We describe this technique in this chapter.

Both techniques require the call to `mmap`. What differs is how the descriptor that is an argument to `mmap` is obtained: by `open` or by `shm_open`. We show this in Figure 13.1. Both are called *memory objects* by Posix.

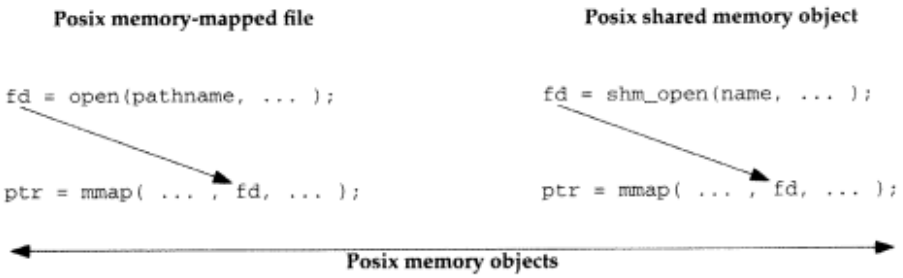

Figure 13.1 Posix memory objects: memory-mapped files and shared memory objects.

13.2 `shm_open` and `shm_unlink` Functions

The two-step process involved with Posix shared memory requires

1. calling `shm_open`, specifying a name argument, to either create a new shared memory object or to open an existing shared memory object, followed by
2. calling `mmap` to map the shared memory into the address space of the calling process.

The name argument used with `shm_open` is then used by any other processes that want to share this memory.

> The reason for this two-step process, instead of a single step that would take a name and return an address within the memory of the calling process, is that `mmap` already existed when Posix invented its form of shared memory. Clearly, a single function could do both steps. The reason that `shm_open` returns a descriptor (recall that `mq_open` returns an `mqd_t` value and `sem_open` returns a pointer to a `sem_t` value) is that an open descriptor is what `mmap` uses to map the memory object into the address space of the process.

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

int shm_open(const char *name, int oflag, mode_t mode);
                                    Returns: nonnegative descriptor if OK, –1 on error

int shm_unlink(const char *name);
                                    Returns: 0 if OK, –1 on error
```

We described the rules about the *name* argument in Section 2.2.

The *oflag* argument must contain either O_RDONLY (read-only) or O_RDWR (read–write), and the following flags can also be specified: O_CREAT, O_EXCL, or O_TRUNC. The O_CREAT and O_EXCL flags were described in Section 2.3. If O_TRUNC is specified along with O_RDWR, then if the shared memory object already exists, it is truncated to 0-length.

mode specifies the permission bits (Figure 2.4) and is used when the O_CREAT flag is specified. Note that unlike the mq_open and sem_open functions, the *mode* argument to shm_open must always be specified. If the O_CREAT flag is not specified, then this argument can be specified as 0.

The return value from shm_open is an integer descriptor that is then used as the fifth argument to mmap.

The shm_unlink function removes the name of a shared memory object. As with all the other unlink functions (the unlink of a pathname in the filesystem, the mq_unlink of a Posix message queue, and the sem_unlink of a Posix named semaphore), unlinking a name has no effect on existing references to the underlying object, until all references to that object are closed. Unlinking a name just prevents any subsequent call to open, mq_open, or sem_open from succeeding.

13.3 ftruncate and fstat Functions

When dealing with mmap, the size of either a regular file or a shared memory object can be changed by calling ftruncate.

```
#include <unistd.h>

int ftruncate(int fd, off_t length);
```

Returns: 0 if OK, –1 on error

Posix defines the function slightly differently for regular files versus shared memory objects.

- For a regular file: If the size of the file was larger than *length*, the extra data is discarded. If the size of the file was smaller than *length*, whether the file is changed or its size is increased is unspecified. Indeed, for a regular file, the portable way to extend the size of the file to *length* bytes is to lseek to offset *length–1* and write 1 byte of data. Fortunately, almost all Unix implementations support extending a file with ftruncate.
- For a shared memory object: ftruncate sets the size of the object to *length*.

We call ftruncate to specify the size of a newly created shared memory object or to change the size of an existing object. When we open an existing shared memory object, we can call fstat to obtain information about the object.

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

int fstat(int fd, struct stat *buf);
```

Returns: 0 if OK, –1 on error

A dozen or more members are in the `stat` structure (Chapter 4 of APUE talks about all the members in detail), but only four contain information when *fd* refers to a shared memory object.

```
struct stat {
  ...
  mode_t  st_mode;    /* mode: S_I{RW}{USR,GRP,OTH} */
  uid_t   st_uid;     /* user ID of owner */
  gid_t   st_gid;     /* group ID of owner */
  off_t   st_size;    /* size in bytes */
  ...
};
```

We show examples of these two function in the next section.

> Unfortunately, Posix.1 does not specify the initial contents of a newly created shared memory object. The description of the `shm_open` function states that "The shared memory object shall have a size of 0." The description of `ftruncate` specifies that for a regular file (not shared memory), "If the file is extended, the extended area shall appear as if it were zero-filled." But nothing is said in the description of `ftruncate` about the new contents of a shared memory object that is extended. The Posix.1 Rationale states that "If the memory object is extended, the contents of the extended areas are zeros" but this is the Rationale, not the official standard. When the author asked on the `comp.std.unix` newsgroup about this detail, the opinion was expressed that some vendors objected to an initialize-to-0 requirement, because of the overhead. If a newly extended piece of shared memory is not initialized to some value (i.e., if the contents are left as is), this could be a security hole.

13.4 Simple Programs

We now develop some simple programs that operate on Posix shared memory.

`shmcreate` Program

Our `shmcreate` program, shown in Figure 13.2, creates a shared memory object with a specified name and length.

19-22 `shm_open` creates the shared memory object. If the `-e` option is specified, it is an error if the object already exists. `ftruncate` sets the length, and `mmap` maps the object into the address space of the process. The program then terminates. Since Posix shared memory has at least kernel persistence, this does not remove the shared memory object.

pxshm/shmcreate.c

```
1 #include    "unpipc.h"

2 int
3 main(int argc, char **argv)
4 {
5     int     c, fd, flags;
6     char   *ptr;
7     off_t   length;

8     flags = O_RDWR | O_CREAT;
9     while ( (c = Getopt(argc, argv, "e")) != -1) {
10        switch (c) {
11        case 'e':
12            flags |= O_EXCL;
13            break;
14        }
15    }
16    if (optind != argc - 2)
17        err_quit("usage: shmcreate [ -e ] <name> <length>");
18    length = atoi(argv[optind + 1]);

19    fd = Shm_open(argv[optind], flags, FILE_MODE);
20    Ftruncate(fd, length);

21    ptr = Mmap(NULL, length, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);

22    exit(0);
23 }
```

pxshm/shmcreate.c

Figure 13.2 Create a Posix shared memory object of a specified size.

`shmunlink` Program

Figure 13.3 shows our trivial program that calls `shm_unlink` to remove the name of a shared memory object from the system.

pxshm/shmunlink.c

```
1 #include    "unpipc.h"

2 int
3 main(int argc, char **argv)
4 {
5     if (argc != 2)
6         err_quit("usage: shmunlink <name>");

7     Shm_unlink(argv[1]);

8     exit(0);
9 }
```

pxshm/shmunlink.c

Figure 13.3 Unlink the name of a Posix shared memory object.

`shmwrite` Program

Figure 13.4 is our `shmwrite` program, which writes a pattern of 0, 1, 2, ..., 254, 255, 0, 1, and so on, to a shared memory object.

pxshm/shmwrite.c

```
 1 #include    "unpipc.h"

 2 int
 3 main(int argc, char **argv)
 4 {
 5     int     i, fd;
 6     struct stat stat;
 7     unsigned char *ptr;

 8     if (argc != 2)
 9         err_quit("usage: shmwrite <name>");

10         /* open, get size, map */
11     fd = Shm_open(argv[1], O_RDWR, FILE_MODE);
12     Fstat(fd, &stat);
13     ptr = Mmap(NULL, stat.st_size, PROT_READ | PROT_WRITE,
14                MAP_SHARED, fd, 0);
15     Close(fd);

16         /* set: ptr[0] = 0, ptr[1] = 1, etc. */
17     for (i = 0; i < stat.st_size; i++)
18         *ptr++ = i % 256;

19     exit(0);
20 }
```

pxshm/shmwrite.c

Figure 13.4 Open a shared memory object and fill it with a pattern.

10-15 The shared memory object is opened by `shm_open`, and we fetch its size with `fstat`. We then map it using `mmap` and `close` the descriptor.

16-18 The pattern is written to the shared memory.

`shmread` Program

Our `shmread` program, shown in Figure 13.5, verifies the pattern that was written by `shmwrite`.

pxshm/shmread.c

```
 1 #include    "unpipc.h"

 2 int
 3 main(int argc, char **argv)
 4 {
 5     int     i, fd;
 6     struct stat stat;
 7     unsigned char c, *ptr;

 8     if (argc != 2)
 9         err_quit("usage: shmread <name>");
```

```
10          /* open, get size, map */
11      fd = Shm_open(argv[1], O_RDONLY, FILE_MODE);
12      Fstat(fd, &stat);
13      ptr = Mmap(NULL, stat.st_size, PROT_READ,
14                 MAP_SHARED, fd, 0);
15      Close(fd);

16          /* check that ptr[0] = 0, ptr[1] = 1, etc. */
17      for (i = 0; i < stat.st_size; i++)
18          if ( (c = *ptr++) != (i % 256))
19              err_ret("ptr[%d] = %d", i, c);

20      exit(0);
21  }
```

pxshm/shmread.c

Figure 13.5 Open a shared memory object and verify its data pattern.

10–15 The shared memory object is opened read-only, its size is obtained by `fstat`, it is mapped by `mmap` (for reading only), and the descriptor is closed.

16–19 The pattern written by `shmwrite` is verified.

Examples

We create a shared memory object whose length is 123,456 bytes under Digital Unix 4.0B named `/tmp/myshm`.

```
alpha % shmcreate /tmp/myshm 123456
alpha % ls -l /tmp/myshm
-rw-r--r--   1 rstevens system    123456 Dec 10 14:33 /tmp/myshm
alpha % od -c /tmp/myshm
0000000   \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
*
0361100
```

We see that a file with the same name is created in the filesystem. Using the `od` program, we can verify that the object's initial contents are all 0. (Octal 0361100, the byte offset just beyond the final byte of the file, equals 123,456.)

Next, we run our `shmwrite` program and use `od` to verify that the initial contents are as expected.

```
alpha % shmwrite /tmp/myshm
alpha % od -x /tmp/myshm | head -4
0000000  0100 0302 0504 0706 0908 0b0a 0d0c 0f0e
0000020  1110 1312 1514 1716 1918 1b1a 1d1c 1f1e
0000040  2120 2322 2524 2726 2928 2b2a 2d2c 2f2e
0000060  3130 3332 3534 3736 3938 3b3a 3d3c 3f3e
alpha % shmread /tmp/myshm
alpha % shmunlink /tmp/myshm
```

We verify the shared memory object's contents with `shmread` and then unlink the name.

If we run our `shmcreate` program under Solaris 2.6, we see that a file is created in the `/tmp` directory with the specified size.

```
solaris % shmcreate -e /testshm 123
solaris % ls -l /tmp/.*testshm*
-rw-r--r--   1 rstevens other1   123 Dec 10 14:40 /tmp/.SHMtestshm
```

Example

We now provide a simple example in Figure 13.6 to demonstrate that a shared memory object can be memory mapped starting at different addresses in different processes.

pxshm/test3.c

```
1  #include    "unpipc.h"

2  int
3  main(int argc, char **argv)
4  {
5      int     fd1, fd2, *ptr1, *ptr2;
6      pid_t   childpid;
7      struct stat stat;

8      if (argc != 2)
9          err_quit("usage: test3 <name>");

10     shm_unlink(Px_ipc_name(argv[1]));
11     fd1 = Shm_open(Px_ipc_name(argv[1]), O_RDWR | O_CREAT | O_EXCL, FILE_MODE);
12     Ftruncate(fd1, sizeof(int));
13     fd2 = Open("/etc/motd", O_RDONLY);
14     Fstat(fd2, &stat);

15     if ( (childpid = Fork()) == 0) {
16             /* child */
17         ptr2 = Mmap(NULL, stat.st_size, PROT_READ, MAP_SHARED, fd2, 0);
18         ptr1 = Mmap(NULL, sizeof(int), PROT_READ | PROT_WRITE,
19                     MAP_SHARED, fd1, 0);
20         printf("child: shm ptr = %p, motd ptr = %p\n", ptr1, ptr2);

21         sleep(5);
22         printf("shared memory integer = %d\n", *ptr1);
23         exit(0);
24     }
25         /* parent: mmap in reverse order from child */
26     ptr1 = Mmap(NULL, sizeof(int), PROT_READ | PROT_WRITE, MAP_SHARED, fd1, 0);
27     ptr2 = Mmap(NULL, stat.st_size, PROT_READ, MAP_SHARED, fd2, 0);
28     printf("parent: shm ptr = %p, motd ptr = %p\n", ptr1, ptr2);
29     *ptr1 = 777;
30     Waitpid(childpid, NULL, 0);

31     exit(0);
32 }
```

pxshm/test3.c

Figure 13.6 Shared memory can appear at different addresses in different processes.

10-14 We create a shared memory segment whose name is the command-line argument, set its size to the size of an integer, and then open the file `/etc/motd`.

15-30 We `fork`, and both the parent and child call `mmap` twice, but in a different order. Each prints the starting address of each memory-mapped region. The child then sleeps

for 5 seconds, the parent stores the value 777 in the shared memory region, and then the child prints this value.

When we run this program, we see that the shared memory object is memory mapped at different starting addresses in the parent and child.

```
solaris % test3 test3.data
parent: shm ptr = eee30000, motd ptr = eee20000
child: shm ptr = eee20000, motd ptr = eee30000
shared memory integer = 777
```

Nevertheless, the parent stores 777 into `0xeee30000`, and the child reads this value from `0xeee20000`. The pointers `ptr1` in the parent and child both point to the same shared memory segment, even though the value of each pointer is different in each process.

13.5 Incrementing a Shared Counter

We now develop an example similar to the one shown in Section 12.3, in which multiple processes increment a counter that is stored in shared memory. We store the counter in shared memory and use a named semaphore for synchronization, but we no longer need a parent–child relationship. Since Posix shared memory objects and Posix named semaphores are referenced by names, the various processes that are incrementing the counter can be unrelated, as long as each knows the IPC names and each has adequate permission for the IPC objects (shared memory and semaphore).

Figure 13.7 shows the server that creates the shared memory object, creates and initializes the semaphore, and then terminates.

Create shared memory object

13-19 We call `shm_unlink` in case the shared memory object still exists, followed by `shm_open` to create the object. The size of the object is set to the size of our `shmstruct` structure by `ftruncate`, and then `mmap` maps the object into our address space. The descriptor is closed.

Create and initialize semaphore

20-22 We call `sem_unlink`, in case the semaphore still exists, followed by `sem_open` to create the named semaphore and initialize it to 1. It will be used as a mutex by any process that increments the counter in the shared memory object. The semaphore is then closed.

Terminate

23 The process terminates. Since Posix shared memory has at least kernel persistence, the object remains in existence until all open references are closed (when this process terminates there are no open references) *and* explicitly unlinked.

Our program must use different names for the shared memory object and the semaphore. There is no guarantee that the implementation adds anything to the Posix IPC names to differentiate among message queues, semaphores, and shared memory. We have seen that Solaris prefixes these three types of names with `.MQ`, `.SEM`, and `.SHM`, but Digital Unix does not.

pxshm/server1.c

```
 1 #include    "unpipc.h"
 2 struct shmstruct {              /* struct stored in shared memory */
 3     int     count;
 4 };
 5 sem_t  *mutex;                  /* pointer to named semaphore */
 6 int
 7 main(int argc, char **argv)
 8 {
 9     int     fd;
10     struct shmstruct *ptr;
11     if (argc != 3)
12         err_quit("usage: server1 <shmname> <semname>");
13     shm_unlink(Px_ipc_name(argv[1]));   /* OK if this fails */
14         /* create shm, set its size, map it, close descriptor */
15     fd = Shm_open(Px_ipc_name(argv[1]), O_RDWR | O_CREAT | O_EXCL, FILE_MODE);
16     Ftruncate(fd, sizeof(struct shmstruct));
17     ptr = Mmap(NULL, sizeof(struct shmstruct), PROT_READ | PROT_WRITE,
18                MAP_SHARED, fd, 0);
19     Close(fd);
20     sem_unlink(Px_ipc_name(argv[2]));   /* OK if this fails */
21     mutex = Sem_open(Px_ipc_name(argv[2]), O_CREAT | O_EXCL, FILE_MODE, 1);
22     Sem_close(mutex);
23     exit(0);
24 }
```

pxshm/server1.c

Figure 13.7 Program that creates and initializes shared memory and semaphore.

Figure 13.8 shows our client program that increments the counter in shared memory some number of times, obtaining the semaphore each time it increments the counter.

Open shared memory

15–18 `shm_open` opens the shared memory object, which must already exist (since `O_CREAT` is not specified). The memory is mapped into the address space of the process by `mmap`, and the descriptor is then closed.

Open semaphore

19 The named semaphore is opened.

Obtain semaphore and increment counter

20–26 The counter is incremented the number of times specified by the command-line argument. We print the old value of the counter each time, along with the process ID, since we will run multiple copies of this program at the same time.

pxshm/client1.c

```
1 #include    "unpipc.h"
2 struct shmstruct {              /* struct stored in shared memory */
3     int     count;
4 };
5 sem_t  *mutex;                  /* pointer to named semaphore */
6 int
7 main(int argc, char **argv)
8 {
9     int     fd, i, nloop;
10    pid_t   pid;
11    struct shmstruct *ptr;
12    if (argc != 4)
13        err_quit("usage: client1 <shmname> <semname> <#loops>");
14    nloop = atoi(argv[3]);
15    fd = Shm_open(Px_ipc_name(argv[1]), O_RDWR, FILE_MODE);
16    ptr = Mmap(NULL, sizeof(struct shmstruct), PROT_READ | PROT_WRITE,
17               MAP_SHARED, fd, 0);
18    Close(fd);
19    mutex = Sem_open(Px_ipc_name(argv[2]), 0);
20    pid = getpid();
21    for (i = 0; i < nloop; i++) {
22        Sem_wait(mutex);
23        printf("pid %ld: %d\n", (long) pid, ptr->count++);
24        Sem_post(mutex);
25    }
26    exit(0);
27 }
```

pxshm/client1.c

Figure 13.8 Program that increments a counter in shared memory.

We first start the server and then run three copies of the client in the background.

```
solaris % server1 shm1 sem1    creates and initializes shared memory and semaphore

solaris % client1 shm1 sem1 10000 &  client1 shm1 sem1 10000 & \
client1 shm1 sem1 10000 &
[2] 17976                      process IDs output by shell
[3] 17977
[4] 17978
pid 17977: 0                   and this process runs first
pid 17977: 1
. . .                          process 17977 continues
pid 17977: 32
pid 17976: 33                  kernel switches processes
. . .                          process 17976 continues
pid 17976: 707
pid 17978: 708                 kernel switches processes
. . .                          process 17978 continues
```

```
pid 17978: 852
pid 17977: 853                  kernel switches processes
. . .                           and so on
pid 17977: 29998
pid 17977: 29999                final value output, which is correct
```

13.6 Sending Messages to a Server

We now modify our producer–consumer example as follows. A server is started that creates a shared memory object in which messages are placed by client processes. Our server just prints these messages, although this could be generalized to do things similarly to the `syslog` daemon, which is described in Chapter 13 of UNPv1. We call these other processes clients, because that is how they appear to our server, but they may well be servers of some form to other clients. For example, a Telnet server is a client of the `syslog` daemon when it sends log messages to the daemon.

Instead of using one of the message passing techniques that we described in Part 2, shared memory is used to contain the messages. This, of course, necessitates some form of synchronization between the clients that are storing messages and the server that is retrieving and printing the messages. Figure 13.9 shows the overall design.

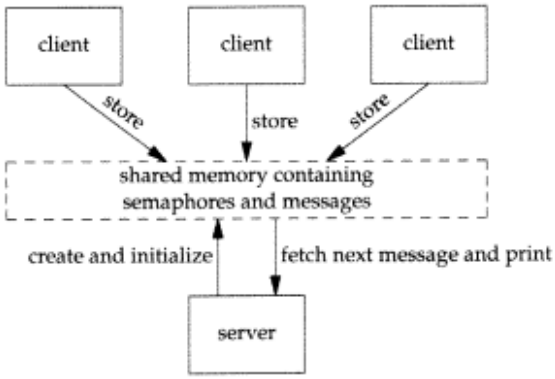

Figure 13.9 Multiple clients sending messages to a server through shared memory.

What we have here are multiple producers (the clients) and a single consumer (the server). The shared memory appears in the address space of the server and in the address space of each client.

Figure 13.10 is our `cliserv2.h` header, which defines a structure with the layout of the shared memory object.

Basic semaphores and variables

5-8 The three Posix memory-based semaphores, `mutex`, `nempty`, and `nstored`, serve the same purpose as the semaphores in our producer–consumer example in Section 10.6. The variable `nput` is the index (0, 1, ... `NMESG-1`) of the next location to store a message. Since we have multiple producers, this variable must be in the shared memory and can be referenced only while the `mutex` is held.

pxshm/cliserv2.h

```
1 #include    "unpipc.h"

2 #define MESGSIZE    256         /* max #bytes per message, incl. null at end */
3 #define NMESG        16         /* max #messages */

4 struct shmstruct {               /* struct stored in shared memory */
5     sem_t   mutex;               /* three Posix memory-based semaphores */
6     sem_t   nempty;
7     sem_t   nstored;
8     int     nput;                /* index into msgoff[] for next put */
9     long    noverflow;           /* #overflows by senders */
10    sem_t   noverflowmutex;      /* mutex for noverflow counter */
11    long    msgoff[NMESG];       /* offset in shared memory of each message */
12    char    msgdata[NMESG * MESGSIZE];  /* the actual messages */
13 };
```

pxshm/cliserv2.h

Figure 13.10 Header that defines layout of shared memory.

Overflow counter

9–10 The possibility exists that a client wants to send a message but all the message slots are taken. But if the client is actually a server of some type (perhaps an FTP server or an HTTP server), the client does not want to wait for the server to free up a slot. Therefore, we will write our clients so that they do not block but increment the `noverflow` counter when this happens. Since this overflow counter is also shared among all the clients and the server, it too requires a mutex so that its value is not corrupted.

Message offsets and data

11–12 The array `msgoff` contains offsets into the `msgdata` array of where each message begins. That is, `msgoff[0]` is 0, `msgoff[1]` is 256 (the value of `MESGSIZE`), `msgoff[2]` is 512, and so on.

Be sure to understand that we must use *offsets* such as these when dealing with shared memory, because the shared memory object can get mapped into a different physical address in each process that maps the object. That is, the return value from `mmap` can be different for each process that calls `mmap` for the same shared memory object. For this reason, we cannot use *pointers* within the shared memory object that contain actual addresses of variables within the object.

Figure 13.11 is our server that waits for a message to be placed into shared memory by one of the clients, and then prints the message.

Create shared memory object

10–16 `shm_unlink` is called first to remove the shared memory object, if it still exists. The object is created by `shm_open` and then mapped into the address space by `mmap`. The descriptor is then closed.

Initialize array of offsets

17–19 The array of offsets is initialized to contain the offset of each message.

pxshm/server2.c

```
 1 #include    "cliserv2.h"

 2 int
 3 main(int argc, char **argv)
 4 {
 5     int     fd, index, lastnoverflow, temp;
 6     long    offset;
 7     struct shmstruct *ptr;

 8     if (argc != 2)
 9         err_quit("usage: server2 <name>");

10         /* create shm, set its size, map it, close descriptor */
11     shm_unlink(Px_ipc_name(argv[1]));   /* OK if this fails */
12     fd = Shm_open(Px_ipc_name(argv[1]), O_RDWR | O_CREAT | O_EXCL, FILE_MODE);
13     ptr = Mmap(NULL, sizeof(struct shmstruct), PROT_READ | PROT_WRITE,
14                MAP_SHARED, fd, 0);
15     Ftruncate(fd, sizeof(struct shmstruct));
16     Close(fd);

17         /* initialize the array of offsets */
18     for (index = 0; index < NMESG; index++)
19         ptr->msgoff[index] = index * MESGSIZE;

20         /* initialize the semaphores in shared memory */
21     Sem_init(&ptr->mutex, 1, 1);
22     Sem_init(&ptr->nempty, 1, NMESG);
23     Sem_init(&ptr->nstored, 1, 0);
24     Sem_init(&ptr->noverflowmutex, 1, 1);

25         /* this program is the consumer */
26     index = 0;
27     lastnoverflow = 0;
28     for ( ; ; ) {
29         Sem_wait(&ptr->nstored);
30         Sem_wait(&ptr->mutex);
31         offset = ptr->msgoff[index];
32         printf("index = %d: %s\n", index, &ptr->msgdata[offset]);
33         if (++index >= NMESG)
34             index = 0;          /* circular buffer */
35         Sem_post(&ptr->mutex);
36         Sem_post(&ptr->nempty);

37         Sem_wait(&ptr->noverflowmutex);
38         temp = ptr->noverflow;  /* don't printf while mutex held */
39         Sem_post(&ptr->noverflowmutex);
40         if (temp != lastnoverflow) {
41             printf("noverflow = %d\n", temp);
42             lastnoverflow = temp;
43         }
44     }

45     exit(0);
46 }
```

pxshm/server2.c

Figure 13.11 Our server that fetches and prints the messages from shared memory.

Initialize semaphores

20-24 The four memory-based semaphores in the shared memory object are initialized. The second argument to `sem_init` is nonzero for each call, since the semaphore is in shared memory and will be shared between processes.

Wait for message, and then print

25-36 The first half of the `for` loop is the standard consumer algorithm: wait for `nstored` to be greater than 0, wait for the `mutex`, process the data, release the `mutex`, and increment `nempty`.

Handle overflows

37-43 Each time around the loop, we also check for overflows. We test whether the counter `noverflows` has changed from its previous value, and if so, print and save the new value. Notice that we fetch the current value of the counter while the `noverflowmutex` is held, but then release it before comparing and possibly printing it. This demonstrates the general rule that we should always write our code to perform the minimum number of operations while a mutex is held.

Our client program is shown in Figure 13.12.

Command-line arguments

10-13 The first command-line argument is the name of the shared memory object, the next is the number of messages to store for the server, and the last one is the number of microseconds to pause between each message. By starting multiple copies of our client and specifying a small value for this pause, we can force an overflow to occur, and verify that the server handles it correctly.

Open and map shared memory

14-18 We open the shared memory object, assuming that it has already been created and initialized by the server, and then map it into our address space. The descriptor can then be closed.

Store messages

19-31 Our client follows the basic algorithm for the consumer but instead of calling `sem_wait(nempty)`, which is where the consumer blocks if there is no room in the buffer for its message, we call `sem_trywait`, which will not block. If the value of the semaphore is 0, an error of `EAGAIN` is returned. We detect this error and increment the overflow counter.

> `sleep_us` is a function from Figures C.9 and C.10 of APUE. It sleeps for the specified number of microseconds, and is implemented by calling either `select` or `poll`.

32-37 While the `mutex` semaphore is held we obtain the value of `offset` and increment `nput`, but we then release the `mutex` before copying the message into the shared memory. We should do only those operations that must be protected while holding the semaphore.

pxshm/client2.c

```
1 #include    "cliserv2.h"

2 int
3 main(int argc, char **argv)
4 {
5     int     fd, i, nloop, nusec;
6     pid_t   pid;
7     char    mesg[MESGSIZE];
8     long    offset;
9     struct shmstruct *ptr;

10    if (argc != 4)
11        err_quit("usage: client2 <name> <#loops> <#usec>");
12    nloop = atoi(argv[2]);
13    nusec = atoi(argv[3]);

14        /* open and map shared memory that server must create */
15    fd = Shm_open(Px_ipc_name(argv[1]), O_RDWR, FILE_MODE);
16    ptr = Mmap(NULL, sizeof(struct shmstruct), PROT_READ | PROT_WRITE,
17               MAP_SHARED, fd, 0);
18    Close(fd);

19    pid = getpid();
20    for (i = 0; i < nloop; i++) {
21        Sleep_us(nusec);
22        snprintf(mesg, MESGSIZE, "pid %ld: message %d", (long) pid, i);

23        if (sem_trywait(&ptr->nempty) == -1) {
24            if (errno == EAGAIN) {
25                Sem_wait(&ptr->noverflowmutex);
26                ptr->noverflow++;
27                Sem_post(&ptr->noverflowmutex);
28                continue;
29            } else
30                err_sys("sem_trywait error");
31        }
32        Sem_wait(&ptr->mutex);
33        offset = ptr->msgoff[ptr->nput];
34        if (++(ptr->nput) >= NMESG)
35            ptr->nput = 0;      /* circular buffer */
36        Sem_post(&ptr->mutex);
37        strcpy(&ptr->msgdata[offset], mesg);
38        Sem_post(&ptr->nstored);
39    }
40    exit(0);
41 }
```

pxshm/client2.c

Figure 13.12 Client that stores messages in shared memory for server.

We first start our server in the background and then run our client, specifying 50 messages with no pause between each message.

```
solaris % server2 serv2 &
[2]     27223
solaris % client2 serv2 50 0
index = 0: pid 27224: message 0
index = 1: pid 27224: message 1
index = 2: pid 27224: message 2
. . .                                   continues like this
index = 15: pid 27224: message 47
index = 0: pid 27224: message 48
index = 1: pid 27224: message 49        no messages lost
```

But if we run our client again, we see some overflows.

```
solaris % client2 serv2 50 0
index = 2: pid 27228: message 0
index = 3: pid 27228: message 1
. . .                                   continues OK
index = 10: pid 27228: message 8
index = 11: pid 27228: message 9
noverflow = 25                          server detects 25 messages lost
index = 12: pid 27228: message 10
index = 13: pid 27228: message 11
. . .                                   continues OK for messages 12–22
index = 9: pid 27228: message 23
index = 10: pid 27228: message 24
```

This time, the client appears to have stored messages 0 through 9, which were then fetched and printed by the server. The client then ran again, storing messages 10 through 49, but there was room for only the first 15 of these, and the remaining 25 (messages 25 through 49) were not stored because of overflow.

Obviously, in this example, we caused the overflow by having the client generate the messages as fast as it can, with no pause between each message, which is not a typical real-world scenario. The purpose of this example, however, is to demonstrate how to handle situations in which no room is available for the client's message but the client does not want to block. This is not unique to shared memory—the same scenario can happen with message queues, pipes, and FIFOs.

> Overrunning a receiver with data is not unique to this example. Section 8.13 of UNPv1 talks about this with regard to UDP datagrams, and the UDP socket receive buffer. Section 18.2 of TCPv3 describes how Unix domain datagram sockets return an error of ENOBUFS to the sender when the receiver's buffer overflows, which differs from UDP. In Figure 13.12, our client (the sender) knows when the server's buffer has overflowed, so if this code were placed into a general-purpose function for other programs to call, the function could return an error to the caller when the server's buffer overflows.

13.7 Summary

Posix shared memory is built upon the `mmap` function from the previous chapter. We first call `shm_open`, specifying a Posix IPC name for the shared memory object, obtain a descriptor, and then memory map the descriptor with `mmap`. The result is similar to a memory-mapped file, but the shared memory object need not be implemented as a file.

Since shared memory objects are represented by descriptors, their size is set with `ftruncate`, and information about an existing object (protection bits, user ID, group ID, and size) is returned by `fstat`.

When we covered Posix message queues and Posix semaphores, we provided sample implementations based on memory-mapped I/O in Sections 5.8 and 10.15. We do not do this for Posix shared memory, because the implementation would be trivial. If we are willing to memory map a file (as is done by the Solaris and Digital Unix implementations), then `shm_open` is implemented by calling `open`, and `shm_unlink` is implemented by calling `unlink`.

Exercises

13.1 Modify Figures 12.16 and 12.19 to work with Posix shared memory instead of a memory-mapped file, and verify that the results are the same as shown for a memory-mapped file.

13.2 In the `for` loops in Figures 13.4 and 13.5, the C idiom `*ptr++` is used to step through the array. Would it be preferable to use `ptr[i]` instead?

14

System V Shared Memory

14.1 Introduction

System V shared memory is similar in concept to Posix shared memory. Instead of calling `shm_open` followed by `mmap`, we call `shmget` followed by `shmat`.

For every shared memory segment, the kernel maintains the following structure of information, defined by including `<sys/shm.h>`:

```
struct shmid_ds {
  struct ipc_perm shm_perm;    /* operation permission struct */
  size_t          shm_segsz;   /* segment size */
  pid_t           shm_lpid;    /* pid of last operation */
  pid_t           shm_cpid;    /* creator pid */
  shmatt_t        shm_nattch;  /* current # attached */
  shmat_t         shm_cnattch; /* in-core # attached */
  time_t          shm_atime;   /* last attach time */
  time_t          shm_dtime;   /* last detach time */
  time_t          shm_ctime;   /* last change time of this structure */
};
```

We described the `ipc_perm` structure in Section 3.3, and it contains the access permissions for the shared memory segment.

14.2 `shmget` Function

A shared memory segment is created, or an existing one is accessed, by the `shmget` function.

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

int shmget(key_t key, size_t size, int oflag);
```

Returns: shared memory identifier if OK, –1 on error

The return value is an integer called the *shared memory identifier* that is used with the three other shm*XXX* functions to refer to this segment.

key can be either a value returned by `ftok` or the constant `IPC_PRIVATE`, as discussed in Section 3.2.

size specifies the size of the segment, in bytes. When a new shared memory segment is created, a nonzero value for *size* must be specified. If an existing shared memory segment is being referenced, *size* should be 0.

oflag is a combination of the read–write permission values shown in Figure 3.6. This can be bitwise-ORed with either `IPC_CREAT` or `IPC_CREAT | IPC_EXCL`, as discussed with Figure 3.4.

When a new shared memory segment is created, it is initialized to *size* bytes of 0.

Note that `shmget` creates or opens a shared memory segment, but does not provide access to the segment for the calling process. That is the purpose of the `shmat` function, which we describe next.

14.3 `shmat` Function

After a shared memory segment has been created or opened by `shmget`, we attach it to our address space by calling `shmat`.

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

void *shmat(int shmid, const void *shmaddr, int flag);
```

Returns: starting address of mapped region if OK, –1 on error

shmid is an identifier returned by `shmget`. The return value from `shmat` is the starting address of the shared memory segment within the calling process. The rules for determining this address are as follows:

- If *shmaddr* is a null pointer, the system selects the address for the caller. This is the recommended (and most portable) method.
- If *shmaddr* is a nonnull pointer, the returned address depends on whether the caller specifies the `SHM_RND` value for the *flag* argument:
  - If `SHM_RND` is not specified, the shared memory segment is attached at the address specified by the *shmaddr* argument.
  - If `SHM_RND` is specified, the shared memory segment is attached at the address specified by the *shmaddr* argument, rounded down by the constant `SHMLBA`. LBA stands for "lower boundary address."

By default, the shared memory segment is attached for both reading and writing by the calling process, if the process has read–write permissions for the segment. The SHM_RDONLY value can also be specified in the *flag* argument, specifying read-only access.

14.4 shmdt Function

When a process is finished with a shared memory segment, it detaches the segment by calling shmdt.

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

int shmdt(const void *shmaddr);
```

Returns: 0 if OK, –1 on error

When a process terminates, all shared memory segments currently attached by the process are detached.

Note that this call does not delete the shared memory segment. Deletion is accomplished by calling shmctl with a command of IPC_RMID, which we describe in the next section.

14.5 shmctl Function

shmctl provides a variety of operations on a shared memory segment.

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

int shmctl(int shmid, int cmd, struct shmid_ds *buff);
```

Returns: 0 if OK, –1 on error

Three commands are provided:

IPC_RMID Remove the shared memory segment identified by *shmid* from the system and destroy the shared memory segment.

IPC_SET Set the following three members of the shmid_ds structure for the shared memory segment from the corresponding members in the structure pointed to by the *buff* argument: shm_perm.uid, shm_perm.gid, and shm_perm.mode. The shm_ctime value is also replaced with the current time.

IPC_STAT Return to the caller (through the *buff* argument) the current shmid_ds structure for the specified shared memory segment.

14.6 Simple Programs

We now develop some simple programs that operate on System V shared memory.

`shmget` Program

Our `shmget` program, shown in Figure 14.1, creates a shared memory segment using a specified pathname and length.

svshm/shmget.c

```
 1 #include    "unpipc.h"

 2 int
 3 main(int argc, char **argv)
 4 {
 5     int     c, id, oflag;
 6     char   *ptr;
 7     size_t  length;

 8     oflag = SVSHM_MODE | IPC_CREAT;
 9     while ( (c = Getopt(argc, argv, "e")) != -1) {
10         switch (c) {
11         case 'e':
12             oflag |= IPC_EXCL;
13             break;
14         }
15     }
16     if (optind != argc - 2)
17         err_quit("usage: shmget [ -e ] <pathname> <length>");
18     length = atoi(argv[optind + 1]);

19     id = Shmget(Ftok(argv[optind], 1), length, oflag);
20     ptr = Shmat(id, NULL, 0);

21     exit(0);
22 }
```

svshm/shmget.c

Figure 14.1 Create a System V shared memory segment of a specified size.

19 `shmget` creates the shared memory segment of the specified size. The pathname passed as a command-line argument is mapped into a System V IPC key by `ftok`. If the `-e` option is specified, it is an error if the segment already exists. If we know that the segment already exists, the length on the command line should be specified as 0.

20 `shmat` attaches the segment into the address space of the process. The program then terminates. Since System V shared memory has at least kernel persistence, this does not remove the shared memory segment.

`shmrmid` Program

Figure 14.2 shows our trivial program that calls `shmctl` with a command of `IPC_RMID` to remove a shared memory segment from the system.

svshm/shmrmid.c

```
 1 #include    "unpipc.h"

 2 int
 3 main(int argc, char **argv)
 4 {
 5     int     id;

 6     if (argc != 2)
 7         err_quit("usage: shmrmid <pathname>");

 8     id = Shmget(Ftok(argv[1], 1), 0, SVSHM_MODE);
 9     Shmctl(id, IPC_RMID, NULL);

10     exit(0);
11 }
```

svshm/shmrmid.c

Figure 14.2 Remove a System V shared memory segment.

`shmwrite` Program

Figure 14.3 is our `shmwrite` program, which writes a pattern of 0, 1, 2, ..., 254, 255, 0, 1, and so on, to a shared memory segment.

svshm/shmwrite.c

```
 1 #include    "unpipc.h"

 2 int
 3 main(int argc, char **argv)
 4 {
 5     int     i, id;
 6     struct shmid_ds buff;
 7     unsigned char *ptr;

 8     if (argc != 2)
 9         err_quit("usage: shmwrite <pathname>");

10     id = Shmget(Ftok(argv[1], 1), 0, SVSHM_MODE);
11     ptr = Shmat(id, NULL, 0);
12     Shmctl(id, IPC_STAT, &buff);

13         /* set: ptr[0] = 0, ptr[1] = 1, etc. */
14     for (i = 0; i < buff.shm_segsz; i++)
15         *ptr++ = i % 256;

16     exit(0);
17 }
```

svshm/shmwrite.c

Figure 14.3 Open a shared memory segment and fill it with a pattern.

10-12 The shared memory segment is opened by `shmget` and attached by `shmat`. We fetch its size by calling `shmctl` with a command of `IPC_STAT`.

13-15 The pattern is written to the shared memory.

shmread Program

Our shmread program, shown in Figure 14.4, verifies the pattern that was written by shmwrite.

svshm/shmread.c

```
 1 #include    "unpipc.h"

 2 int
 3 main(int argc, char **argv)
 4 {
 5     int     i, id;
 6     struct shmid_ds buff;
 7     unsigned char c, *ptr;

 8     if (argc != 2)
 9         err_quit("usage: shmread <pathname>");

10     id = Shmget(Ftok(argv[1], 1), 0, SVSHM_MODE);
11     ptr = Shmat(id, NULL, 0);
12     Shmctl(id, IPC_STAT, &buff);

13         /* check that ptr[0] = 0, ptr[1] = 1, etc. */
14     for (i = 0; i < buff.shm_segsz; i++)
15         if ( (c = *ptr++) != (i % 256))
16             err_ret("ptr[%d] = %d", i, c);

17     exit(0);
18 }
```

svshm/shmread.c

Figure 14.4 Open a shared memory segment and verify its data pattern.

10-12 The shared memory segment is opened and attached. Its size is obtained by calling shmctl with a command of IPC_STAT.

13-16 The pattern written by shmwrite is verified.

Examples

We create a shared memory segment whose length is 1234 bytes under Solaris 2.6. The pathname used to identify the segment (e.g., the pathname passed to ftok) is the pathname of our shmget executable. Using the pathname of a server's executable file often provides a unique identifier for a given application.

```
solaris % shmget shmget 1234
solaris % ipcs -bmo
IPC status from <running system> as of Thu Jan  8 13:17:06 1998
T         ID      KEY        MODE        OWNER    GROUP NATTCH      SEGSZ
Shared Memory:
m          1   0x0000f12a --rw-r--r-- rstevens   other1      0       1234
```

We run the ipcs program to verify that the segment has been created. We notice that the number of attaches (which is stored in the shm_nattch member of the shmid_ds structure) is 0, as we expect.

Next, we run our `shmwrite` program to set the contents of the shared memory segment to the pattern. We verify the shared memory segment's contents with `shmread` and then remove the identifier.

```
solaris % shmwrite shmget
solaris % shmread shmget
solaris % shmrmid shmget
solaris % ipcs -bmo
IPC status from <running system> as of Thu Jan  8 13:18:01 1998
T         ID      KEY        MODE        OWNER    GROUP NATTCH      SEGSZ
Shared Memory:
```

We run `ipcs` to verify that the shared memory segment has been removed.

When the name of the server executable is used as an argument to `ftok` to identify some form of System V IPC, the absolute pathname would normally be specified, such as `/usr/bin/myserverd`, and not a relative pathname as we have used (`shmget`). We have been able to use a relative pathname for the examples in this section because all of the programs have been run from the directory containing the server executable. Realize that `ftok` uses the i-node of the file to form the IPC identifier (e.g., Figure 3.2), and whether a given file is referenced by an absolute pathname or by a relative pathname has no effect on the i-node.

14.7 Shared Memory Limits

As with System V message queues and System V semaphores, certain system limits exist on System V shared memory (Section 3.8). Figure 14.5 shows the values for some different implementations. The first column is the traditional System V name for the kernel variable that contains this limit.

Name	Description	DUnix 4.0B	Solaris 2.6
`shmmax`	max #bytes for a shared memory segment	4,194,304	1,048,576
`shmmnb`	min #bytes for a shared memory segment	1	1
`shmmni`	max #shared memory identifiers, systemwide	128	100
`shmseg`	max #shared memory segments attached per process	32	6

Figure 14.5 Typical system limits for System V shared memory.

Example

The program in Figure 14.6 determines the four limits shown in Figure 14.5.

svshm/limits.c

```
1 #include    "unpipc.h"

2 #define MAX_NIDS    4096

3 int
4 main(int argc, char **argv)
```

```
 5 {
 6      int     i, j, shmid[MAX_NIDS];
 7      void    *addr[MAX_NIDS];
 8      unsigned long size;

 9          /* see how many identifiers we can "open" */
10      for (i = 0; i <= MAX_NIDS; i++) {
11          shmid[i] = shmget(IPC_PRIVATE, 1024, SVSHM_MODE | IPC_CREAT);
12          if (shmid[i] == -1) {
13              printf("%d identifiers open at once\n", i);
14              break;
15          }
16      }
17      for (j = 0; j < i; j++)
18          Shmctl(shmid[j], IPC_RMID, NULL);

19          /* now see how many we can "attach" */
20      for (i = 0; i <= MAX_NIDS; i++) {
21          shmid[i] = Shmget(IPC_PRIVATE, 1024, SVSHM_MODE | IPC_CREAT);
22          addr[i] = shmat(shmid[i], NULL, 0);
23          if (addr[i] == (void *) -1) {
24              printf("%d shared memory segments attached at once\n", i);
25              Shmctl(shmid[i], IPC_RMID, NULL);    /* the one that failed */
26              break;
27          }
28      }
29      for (j = 0; j < i; j++) {
30          Shmdt(addr[j]);
31          Shmctl(shmid[j], IPC_RMID, NULL);
32      }

33          /* see how small a shared memory segment we can create */
34      for (size = 1;; size++) {
35          shmid[0] = shmget(IPC_PRIVATE, size, SVSHM_MODE | IPC_CREAT);
36          if (shmid[0] != -1) {    /* stop on first success */
37              printf("minimum size of shared memory segment = %lu\n", size);
38              Shmctl(shmid[0], IPC_RMID, NULL);
39              break;
40          }
41      }

42          /* see how large a shared memory segment we can create */
43      for (size = 65536;; size += 4096) {
44          shmid[0] = shmget(IPC_PRIVATE, size, SVSHM_MODE | IPC_CREAT);
45          if (shmid[0] == -1) {    /* stop on first failure */
46              printf("maximum size of shared memory segment = %lu\n", size - 4096);
47              break;
48          }
49          Shmctl(shmid[0], IPC_RMID, NULL);
50      }

51      exit(0);
52 }
```

svshm/limits.c

Figure 14.6 Determine the system limits on shared memory.

We run this program under Digital Unix 4.0B.

```
alpha % limits
127 identifiers open at once
32 shared memory segments attached at once
minimum size of shared memory segment = 1
maximum size of shared memory segment = 4194304
```

The reason that Figure 14.5 shows 128 identifiers but our program can create only 127 identifiers is that one shared memory segment has already been created by a system daemon.

14.8 Summary

System V shared memory is similar in concept to Posix shared memory. The most common function calls are

- `shmget` to obtain an identifier,
- `shmat` to attach the shared memory segment to the address space of the process,
- `shmctl` with a command of `IPC_STAT` to fetch the size of an existing shared memory segment, and
- `shmctl` with a command of `IPC_RMID` to remove a shared memory object.

One difference is that the size of a Posix shared memory object can be changed at any time by calling `ftruncate` (as we demonstrated in Exercise 13.1), whereas the size of a System V shared memory object is fixed by `shmget`.

Exercises

14.1 Figure 6.8 was a modification to Figure 6.6 that accepted an identifier instead of a pathname to specify the queue. We showed that the identifier is all we need to know to access a System V message queue (assuming we have adequate permission). Make similar modifications to Figure 14.4 and show that the same feature applies to System V shared memory.

Part 5

Remote Procedure Calls

15

Doors

15.1 Introduction

When discussing client–server scenarios and procedure calls, there are three different types of procedure calls, which we show in Figure 15.1.

1. A *local procedure call* is what we are familiar with from our everyday C programming: the procedure (function) being called and the calling procedure are both in the same process. Typically, some machine instruction is executed that transfers control to the new procedure, and the called procedure saves machine registers and allocates space on the stack for its local variables.
2. A *remote procedure call* (RPC) is when the procedure being called and the calling procedure are in different processes. We normally refer to the caller as the client and the procedure being called as the server. In the middle scenario in Figure 15.1, we show the client and server executing on the same host. This is a frequently occurring special case of the bottom scenario in this figure, and this is what *doors* provide us: the ability for a process to call a procedure (function) in another process on the same host. One process (a server) makes a procedure available within that process for other processes (clients) to call by creating a door for that procedure. We can also think of doors as a special type of IPC, since information, in the form function arguments and return values, is exchanged between the client and server.
3. RPC in general allows a client on one host to call a server procedure on another host, as long as the two hosts are connected by some form of network (the bottom scenario in Figure 15.1). This is what we describe in Chapter 16.

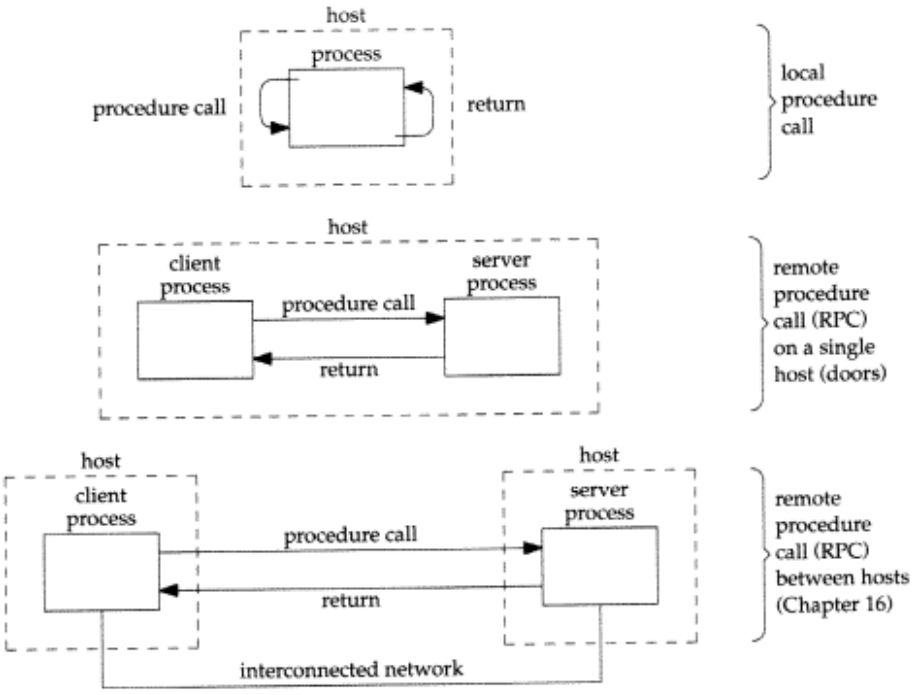

Figure 15.1 Three different types of procedure calls.

Historically, doors were developed for the Spring distributed operating system, details of which are available at `http://www.sun.com/tech/projects/spring`. A description of the doors IPC mechanism in this operating system is in [Hamilton and Kougiouris 1993].

Doors then appeared in Solaris 2.5, although the only manual page contained just a warning that doors were an experimental interface used only by some Sun applications. With Solaris 2.6, the interface was documented in eight manual pages, but these manual pages list the stability of the interface as "evolving." Expect that changes might occur to the API that we describe in this chapter with future releases of Solaris. A preliminary version of doors for Linux is being developed: `http://www.cs.brown.edu/~tor/doors`.

The implementation of doors in Solaris 2.6 involves a library (containing the `door_XXX` functions that we describe in this chapter), which is linked with the user's application (`-ldoor`), and a kernel filesystem (`/kernel/sys/doorfs`).

Even though doors are a Solaris-only feature, we describe them in detail because they provide a nice introduction to remote procedure calls, without having to deal with any networking details. We will also see in Appendix A that they are as fast, if not faster, than all other forms of message passing.

Local procedure calls are *synchronous*: the caller does not regain control until the called procedure returns. Threads can be thought of as providing a form of *asynchronous* procedure call: a function is called (the third argument to `pthread_create`), and both that function and the caller appear to execute at the same

time. The caller can wait for the new thread to finish by calling `pthread_join`. Remote procedure calls can be either synchronous or asynchronous, but we will see that door calls are synchronous.

Within a process (client or server), doors are identified by descriptors. Externally, doors may be identified by pathnames in the filesystem. A server creates a door by calling `door_create`, whose argument is a pointer to the procedure that will be associated with this door, and whose return value is a descriptor for the newly created door. The server then associates a pathname with the door descriptor by calling `fattach`. A client opens a door by calling `open`, whose argument is the pathname that the server associated with the door, and whose return value is the client's descriptor for this door. The client then calls the server procedure by calling `door_call`. Naturally, a server for one door could be a client for another door.

We said that door calls are *synchronous*: when the client calls `door_call`, this function does not return until the server procedure returns (or some error occurs). The Solaris implementation of doors is also tied to threads. Each time a client calls a server procedure, a thread in the server process handles this client's call. Thread management is normally done automatically by the doors library, creating new threads as they are needed, but we will see how a server process can manage these threads itself, if desired. This also means that a given server can be servicing multiple client calls of the same server procedure at the same time, with one thread per client. This is a *concurrent* server. Since multiple instances of a given server procedure can be executing at the same time (each instance as one thread), the server procedures must be thread safe.

When a server procedure is called, both *data* and *descriptors* can be passed from the client to the server. Both data and descriptors can also be passed back from the server to the client. Descriptor passing is inherent to doors. Furthermore, since doors are identified by descriptors, this allows a process to pass a door to some other process. We say more about descriptor passing in Section 15.8.

Example

We begin our description of doors with a simple example: the client passes a long integer to the server, and the server returns the square of that value as the long integer result. Figure 15.2 shows the client. (We gloss over many details in this example, all of which we cover later in the chapter.)

Open the door

8-10 The door is specified by the pathname on the command line, and it is opened by calling `open`. The returned descriptor is called the *door descriptor*, but sometimes we just call it the *door*.

Set up arguments and pointer to result

11-18 The `arg` structure contains a pointer to the arguments and a pointer to the results. `data_ptr` points to the first byte of the arguments, and `data_size` specifies the number of argument bytes. The two members `desc_ptr` and `desc_num` deal with the passing of descriptors, which we describe in Section 15.8. `rbuf` points to the first byte of the result buffer, and `rsize` is its size.

doors/client1.c

```
 1 #include    "unpipc.h"
 2 int
 3 main(int argc, char **argv)
 4 {
 5     int     fd;
 6     long    ival, oval;
 7     door_arg_t arg;
 8     if (argc != 3)
 9         err_quit("usage: client1 <server-pathname> <integer-value>");
10     fd = Open(argv[1], O_RDWR); /* open the door */
11         /* set up the arguments and pointer to result */
12     ival = atol(argv[2]);
13     arg.data_ptr = (char *) &ival;  /* data arguments */
14     arg.data_size = sizeof(long);   /* size of data arguments */
15     arg.desc_ptr = NULL;
16     arg.desc_num = 0;
17     arg.rbuf = (char *) &oval;  /* data results */
18     arg.rsize = sizeof(long);   /* size of data results */
19         /* call server procedure and print result */
20     Door_call(fd, &arg);
21     printf("result: %ld\n", oval);
22     exit(0);
23 }
```

doors/client1.c

Figure 15.2 Client that sends a long integer to the server to be squared.

Call server procedure and print result

19–21 We call the server procedure by calling `door_call`, specifying as arguments the door descriptor and a pointer to the argument structure. Upon return, we print the result.

The server program is shown in Figure 15.3. It consists of a server procedure named `servproc` and a `main` function.

Server procedure

2–10 The server procedure is called with five arguments, but the only one we use is `dataptr`, which points to the first byte of the arguments. The long integer argument is fetched through this pointer and squared. Control is passed back to the client, along with the result, by `door_return`. The first argument points to the result, the second is the size of the result, and the remaining two deal with the returning of descriptors.

Create a door descriptor and attach to pathname

17–21 A door descriptor is created by `door_create`. The first argument is a pointer to the function that will be called for this door (`servproc`). After this descriptor is obtained, it must be associated with a pathname in the filesystem, because this pathname is how the client identifies the door. This association is done by creating a regular

doors/server1.c

```
 1 #include    "unpipc.h"
 2 void
 3 servproc(void *cookie, char *dataptr, size_t datasize,
 4           door_desc_t *descptr, size_t ndesc)
 5 {
 6     long    arg, result;
 7     arg = *((long *) dataptr);
 8     result = arg * arg;
 9     Door_return((char *) &result, sizeof(result), NULL, 0);
10 }
11 int
12 main(int argc, char **argv)
13 {
14     int     fd;
15     if (argc != 2)
16         err_quit("usage: server1 <server-pathname>");
17         /* create a door descriptor and attach to pathname */
18     fd = Door_create(servproc, NULL, 0);
19     unlink(argv[1]);
20     Close(Open(argv[1], O_CREAT | O_RDWR, FILE_MODE));
21     Fattach(fd, argv[1]);
22         /* servproc() handles all client requests */
23     for ( ; ; )
24         pause();
25 }
```

doors/server1.c

Figure 15.3 Server that returns the square of a long integer.

file in the filesystem (we call `unlink` first, in case the file already exists, ignoring any error return) and calling `fattach`, an SVR4 function that associates a descriptor with a pathname.

Main server thread does nothing

22–24 The main server thread then blocks in a call to `pause`. All the work is done by the `servproc` function, which will be executed as another thread in the server process each time a client request arrives.

To run this client and server, we first start the server in one window

```
solaris % server1 /tmp/server1
```

and then start the client in another window, specifying the same pathname argument that we passed to the server:

```
solaris % client1 /tmp/server1 9
result: 81
solaris % ls -l /tmp/server1
Drw-r--r--   1 rstevens other1         0 Apr  9 10:09 /tmp/server1
```

The result is what we expect, and when we execute `ls`, we see that it prints the character `D` as the first character to indicate that this pathname is a door.

Figure 15.4 shows a diagram of what appears to be happening with this example. It appears that `door_call` calls the server procedure, which then returns.

Figure 15.5 shows what is actually going on when we call a procedure in a different process on the same host.

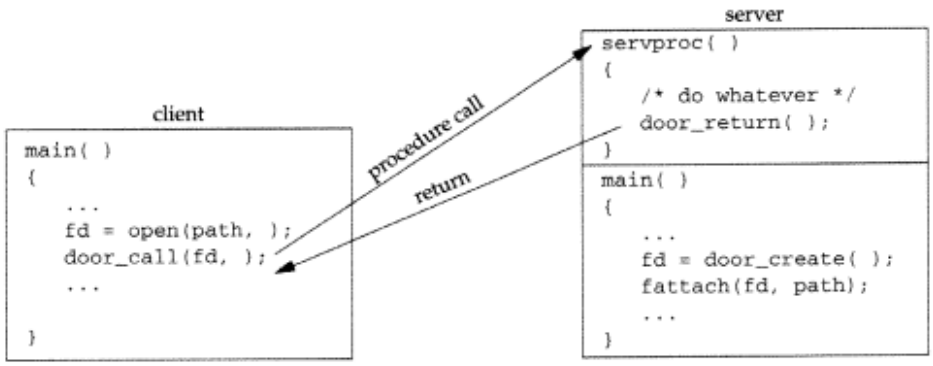

Figure 15.4 Apparent procedure call from one process to another.

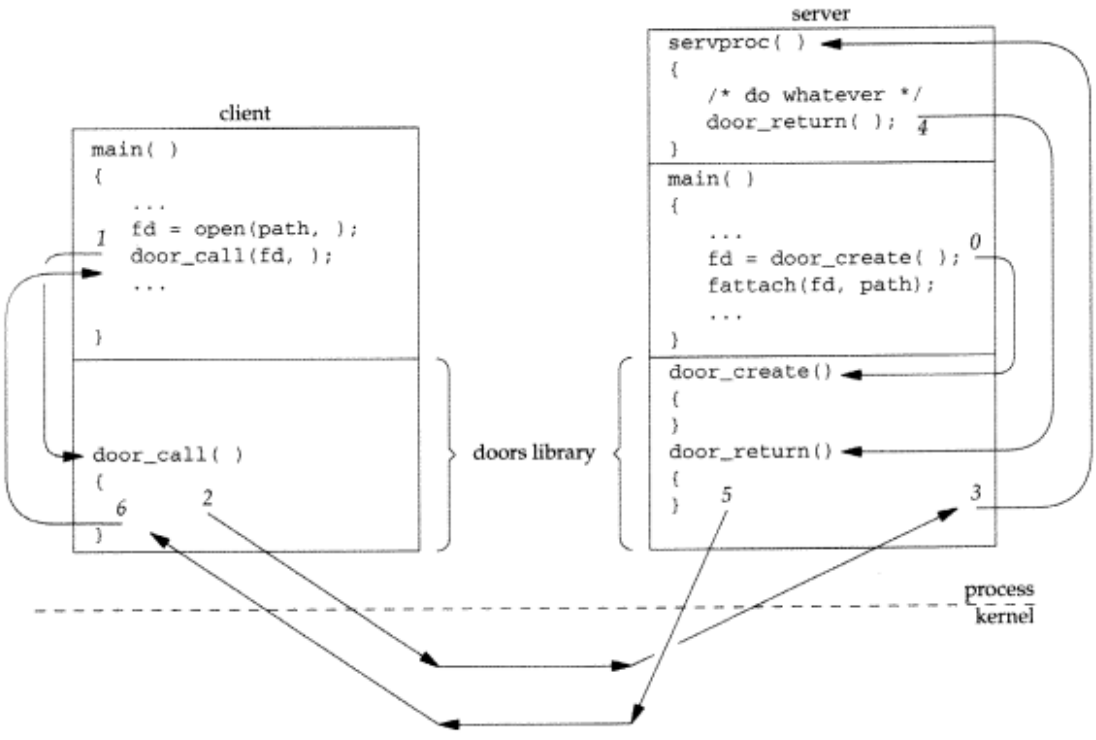

Figure 15.5 Actual flow of control for a procedure call from one process to another.

The following numbered steps in Figure 15.5 take place.

0. The server process starts first, calls `door_create` to create a door descriptor referring to the function `servproc`, and then attaches this descriptor to a pathname in the filesystem.
1. The client process starts and calls `door_call`. This is actually a function in the doors library.
2. The `door_call` library function performs a system call into the kernel. The target procedure is identified and control is passed to some doors library function in the target process.
3. The actual server procedure (named `servproc` in our example) is called.
4. The server procedure does whatever it needs to do to handle the client request and calls `door_return` when it is done.
5. `door_return` is actually a function in the doors library, and it performs a system call into the kernel.
6. The client is identified and control is passed back to the client.

The remaining sections describe the doors API in more detail looking at many examples. In Appendix A, we will see that doors provide the fastest form of IPC, in terms of latency.

15.2 `door_call` Function

The `door_call` function is called by a client, and it calls a server procedure that is executing in the address space of the server process.

```
#include <door.h>

int door_call(int fd, door_arg_t *argp);
```

Returns: 0 if OK, –1 on error

The descriptor *fd* is normally returned by `open` (e.g., Figure 15.2). The pathname opened by the client identifies the server procedure that is called by `door_call` when this descriptor is the first argument.

The second argument *argp* points to a structure describing the arguments and the buffer to be used to hold the return values:

```
typedef struct door_arg {
  char         *data_ptr;   /* call: ptr to data arguments;
                               return: ptr to data results */
  size_t        data_size;  /* call: #bytes of data arguments;
                               return: actual #bytes of data results */
  door_desc_t  *desc_ptr;   /* call: ptr to descriptor arguments;
                               return: ptr to descriptor results */
  size_t        desc_num;   /* call: number of descriptor arguments;
                               return: number of descriptor results */
  char         *rbuf;       /* ptr to result buffer */
  size_t        rsize;      /* #bytes of result buffer */
} door_arg_t;
```

Upon return, this structure describes the return values. All six members of this structure can change on return, as we now describe.

> The use of `char *` for the two pointers is strange and necessitates explicit casts in our code to avoid compiler warnings. We would expect `void *` pointers. We will see the same use of `char *` with the first argument to `door_return`. Solaris 2.7 will probably change the datatype of `desc_num` to be an `unsigned int`, and the final argument to `door_return` would change accordingly.

Two types of arguments and two types of results exist: data and descriptors.

- The *data arguments* are a sequence of `data_size` bytes pointed to by `data_ptr`. The client and server must somehow "know" the format of these arguments (and the results). For example, no special coding tells the server the datatypes of the arguments. In Figures 15.2 and 15.3, the client and server were written to know that the argument was one long integer and that the result was also one long integer. One way to encapsulate this information (for someone reading the code years later) is to put all the arguments into one structure, all the results into another structure, and define both structures in a header that the client and server include. We show an example of this with Figures 15.11 and 15.12. If there are no data arguments, we specify `data_ptr` as a null pointer and `data_size` as 0.

  > Since the client and server deal with binary arguments and results that are packed into an argument buffer and a result buffer, the implication is that the client and server must be compiled with the same compiler. Sometimes different compilers, on the same system, pack structures differently.

- The *descriptor arguments* are an array of `door_desc_t` structures, each one containing one descriptor that is passed from the client to the server procedure. The number of `door_desc_t` structures passed is `desc_num`. (We describe this structure and what it means to "pass a descriptor" in Section 15.8.) If there are no descriptor arguments, we specify `desc_ptr` as a null pointer and `desc_num` as 0.
- Upon return, `data_ptr` points to the *data results*, and `data_size` specifies the size of these results. If there are no data results, `data_size` will be 0, and we should ignore `data_ptr`.

- Upon return, there can also be *descriptor results*: `desc_ptr` points to an array of `door_desc_t` structures, each one containing one descriptor that was passed by the server procedure to the client. The number of `door_desc_t` structures returned is contained in `desc_num`. If there are no descriptor results, `desc_num` will be 0, and we should ignore `desc_ptr`.

Using the same buffer for the arguments and results is OK. That is, `data_ptr` and `desc_ptr` can point into the buffer specified by `rbuf` when `door_call` is called.

Before calling `door_call`, the client sets `rbuf` to point to a buffer where the results will be stored, and `rsize` is the buffer size. Normally upon return, `data_ptr` and `desc_ptr` both point into this result buffer. If this buffer is too small to hold the server's results, the doors library automatically allocates a new buffer in the caller's address space using `mmap` (Section 12.2) and updates `rbuf` and `rsize` accordingly. `data_ptr` and `desc_ptr` will then point into this newly allocated buffer. It is the caller's responsibility to notice that `rbuf` has changed and at some later time to return this buffer to the system by calling `munmap` with `rbuf` and `rsize` as the arguments to `munmap`. We show an example of this with Figure 15.7.

15.3 door_create Function

A server process establishes a server procedure by calling `door_create`.

```
#include <door.h>

typedef void Door_server_proc(void *cookie, char *dataptr, size_t datasize,
                              door_desc_t *descptr, size_t ndesc);

int door_create(Door_server_proc *proc, void *cookie, u_int attr);
```

Returns: nonnegative descriptor if OK, –1 on error

In this declaration, we have added our own `typedef`, which simplifies the function prototype. This `typedef` says that door server procedures (e.g., `servproc` in Figure 15.3) are called with five arguments and return nothing.

When `door_create` is called by a server, the first argument *proc* is the address of the server procedure that will be associated with the door descriptor that is the return value of this function. When this server procedure is called, its first argument *cookie* is the value that was passed as the second argument to `door_create`. This provides a way for the server to cause some pointer to be passed to this procedure every time that procedure is called by a client. The next four arguments to the server procedure, *dataptr*, *datasize*, *descptr*, and *ndesc*, describe the data arguments and the descriptor arguments from the client: the information described by the first four members of the `door_arg_t` structure that we described in the previous section.

The final argument to `door_create`, *attr*, describes special attributes of this server procedure, and is either 0 or the bitwise-OR of the following two constants:

DOOR_PRIVATE The doors library automatically creates new threads in the server process as needed to call the server procedures as client requests arrive. By default, these threads are placed into a process-wide thread pool and can be used to service a client request for any door in the server process.

Specifying the DOOR_PRIVATE attribute tells the library that this door is to have its own pool of server threads, separate from the process-wide pool.

DOOR_UNREF When the number of descriptors referring to this door goes from two to one, the server procedure is called with a second argument (*dataptr*) of DOOR_UNREF_DATA. The *descptr* argument is a null pointer, and both *datasize* and *ndesc* are 0. We show some examples of this attribute starting with Figure 15.16.

The return value from a server procedure is declared as void because a server procedure never returns by calling return or by falling off the end of the function. Instead, the server procedure calls door_return, which we describe in the next section.

We saw in Figure 15.3 that after obtaining a door descriptor from door_create, the server normally calls fattach to associate that descriptor with a pathname in the filesystem. The client opens that pathname to obtain its door descriptor for its call to door_call.

> fattach is not a Posix.1 function but it is required by Unix 98. Also, a function named fdetach undoes this association, and a command named fdetach just invokes this function.

Door descriptors created by door_create have the FD_CLOEXEC bit set in the descriptor's file descriptor flags. This means the descriptor will be closed by the kernel if this process calls any of the exec functions. With regard to fork, even though all descriptors open in the parent are then shared by the child, only the parent will receive door invocations from clients; none are delivered to the child, even though the descriptor returned by door_create is open in the child.

> If we consider that a door is identified by a process ID and the address of a server procedure to call (which we will see in the door_info_t structure in Section 15.6), then these two rules regarding fork and exec make sense. A child will never get any door invocations, because the process ID associated with the door is the process ID of the parent that called door_create. A door descriptor must be closed upon an exec, because even though the process ID does not change, the address of the server procedure associated with the door has no meaning in the newly invoked program that runs after exec.

15.4 door_return Function

When a server procedure is done it returns by calling door_return. This causes the associated door_call in the client to return.

```
#include <door.h>

int door_return(char *dataptr, size_t datasize, door_desc_t *descptr, size_t ndesc);
```

Returns: no return to caller if OK, –1 on error

The data results are specified by *dataptr* and *datasize*, and the descriptor results are specified by *descptr* and *ndesc*.

15.5 door_cred Function

One nice feature of doors is that the server procedure can obtain the client's credentials on every call. This is done with the door_cred function.

```
#include <door.h>

int door_cred(door_cred_t *cred);
```

Returns: 0 if OK, –1 on error

The door_cred_t structure that is pointed to by *cred* contains the client's credentials on return.

```
typedef struct door_cred {
  uid_t  dc_euid;  /* effective user ID of client */
  gid_t  dc_egid;  /* effective group ID of client */
  uid_t  dc_ruid;  /* real user ID of client */
  gid_t  dc_rgid;  /* real group ID of client */
  pid_t  dc_pid;   /* process ID of client */
} door_cred_t;
```

Section 4.4 of APUE talks about the difference between the effective and real IDs, and we show an example with Figure 15.8.

Notice that there is no descriptor argument to this function. It returns information about the client of the current door invocation, and must therefore be called by the server procedure or some function called by the server procedure.

15.6 door_info Function

The door_cred function that we just described provides information for the server about the client. The client can find information about the server by calling the door_info function.

```
#include <door.h>

int door_info(int fd, door_info_t *info);
```

Returns: 0 if OK, –1 on error

fd specifies an open door. The `door_info_t` structure that is pointed to by *info* contains information about the server on return.

```
typedef struct door_info {
  pid_t        di_target;     /* server process ID */
  door_ptr_t   di_proc;       /* server procedure */
  door_ptr_t   di_data;       /* cookie for server procedure */
  door_attr_t  di_attributes; /* attributes associated with door */
  door_id_t    di_uniquifier; /* unique number */
} door_info_t;
```

`di_target` is the process ID of the server, and `di_proc` is the address of the server procedure within the server process (which is probably of little use to the client). The cookie pointer that is passed as the first argument to the server procedure is returned as `di_data`.

The current attributes of the door are contained in `di_attributes`, and we described two of these in Section 15.3: `DOOR_PRIVATE` and `DOOR_UNREF`. Two new attributes are `DOOR_LOCAL` (the procedure is local to this process) and `DOOR_REVOKE` (the server has revoked the procedure associated with this door by calling the `door_revoke` function).

Each door is assigned a systemwide unique number when created, and this is returned as `di_uniquifier`.

This function is normally called by the client, to obtain information about the server. But it can also be issued by a server procedure with a first argument of `DOOR_QUERY`: this returns information about the calling thread. In this scenario, the address of the server procedure (`di_proc`) and the cookie (`di_data`) might be of interest.

15.7 Examples

We now show some examples of the five functions that we have described.

`door_info` Function

Figure 15.6 shows a program that opens a door, then calls `door_info`, and prints information about the door.

doors/doorinfo.c

```
 1 #include    "unpipc.h"
 2 int
 3 main(int argc, char **argv)
 4 {
 5     int     fd;
 6     struct stat stat;
 7     struct door_info info;
 8     if (argc != 2)
 9         err_quit("usage: doorinfo <pathname>");
10     fd = Open(argv[1], O_RDONLY);
11     Fstat(fd, &stat);
```

```
12      if (S_ISDOOR(stat.st_mode) == 0)
13          err_quit("pathname is not a door");

14      Door_info(fd, &info);
15      printf("server PID = %ld, uniquifier = %ld",
16              (long) info.di_target, (long) info.di_uniquifier);
17      if (info.di_attributes & DOOR_LOCAL)
18          printf(", DOOR_LOCAL");
19      if (info.di_attributes & DOOR_PRIVATE)
20          printf(", DOOR_PRIVATE");
21      if (info.di_attributes & DOOR_REVOKED)
22          printf(", DOOR_REVOKED");
23      if (info.di_attributes & DOOR_UNREF)
24          printf(", DOOR_UNREF");
25      printf("\n");

26      exit(0);
27  }
```

doors/doorinfo.c

Figure 15.6 Print information about a door.

We `open` the specified pathname and first verify that it is a door. The `st_mode` member of the `stat` structure for a door will contain a value so that the `S_ISDOOR` macro is true. We then call `door_info`.

We first run the program specifying a pathname that is not a door, and then run it on the two doors that are used by Solaris 2.6.

```
solaris % doorinfo /etc/passwd
pathname is not a door

solaris % doorinfo /etc/.name_service_door
server PID = 308, uniquifier = 18, DOOR_UNREF
solaris % doorinfo /etc/.syslog_door
server PID = 282, uniquifier = 1635

solaris % ps -f -p 308
    root   308     1  0   Apr 01 ?        0:34 /usr/sbin/nscd
solaris % ps -f -p 282
    root   282     1  0   Apr 01 ?        0:10 /usr/sbin/syslogd -n -z 14
```

We use the `ps` command to see what program is running with the process ID returned by `door_info`.

Result Buffer Too Small

When describing the `door_call` function, we mentioned that if the result buffer is too small for the server's results, a new buffer is automatically allocated. We now show an example of this. Figure 15.7 shows the new client, a simple modification of Figure 15.2.

19–23 In this version of our program, we print the address of our `oval` variable, the contents of `data_ptr`, which points to the result on return from `door_call`, and the address and size of the result buffer (`rbuf` and `rsize`).

doors/client2.c

```
1 #include    "unpipc.h"
2 int
3 main(int argc, char **argv)
4 {
5     int     fd;
6     long    ival, oval;
7     door_arg_t arg;
8     if (argc != 3)
9         err_quit("usage: client2 <server-pathname> <integer-value>");
10    fd = Open(argv[1], O_RDWR); /* open the door */
11        /* set up the arguments and pointer to result */
12    ival = atol(argv[2]);
13    arg.data_ptr = (char *) &ival;  /* data arguments */
14    arg.data_size = sizeof(long);   /* size of data arguments */
15    arg.desc_ptr = NULL;
16    arg.desc_num = 0;
17    arg.rbuf = (char *) &oval;  /* data results */
18    arg.rsize = sizeof(long);   /* size of data results */
19        /* call server procedure and print result */
20    Door_call(fd, &arg);
21    printf("&oval = %p, data_ptr = %p, rbuf = %p, rsize  = %d\n",
22           &oval, arg.data_ptr, arg.rbuf, arg.rsize);
23    printf("result: %ld\n", *((long *) arg.data_ptr));
24    exit(0);
25 }
```

doors/client2.c

Figure 15.7 Print address of result.

When we run this program, we have not changed the size of the result buffer from Figure 15.2, so we expect to find that `data_ptr` and `rbuf` both point to our `oval` variable, and that `rsize` is 4 bytes. Indeed, this is what we see:

```
solaris % client2 /tmp/server2 22
&oval = effff740, data_ptr = effff740, rbuf = effff740, rsize  = 4
result: 484
```

We now change only one line in Figure 15.7, decreasing the size of the client's result buffer by 1 byte. The new version of line 18 from Figure 15.7 is

```
    arg.rsize = sizeof(long) - 1;   /* size of data results */
```

When we execute this new client program, we see that a new result buffer has been allocated and `data_ptr` points to this new buffer.

```
solaris % client3 /tmp/server3 33
&oval = effff740, data_ptr = ef620000, rbuf = ef620000, rsize  = 4096
result: 1089
```

The allocated size of 4096 is the page size on this system, which we saw in Section 12.6. We can see from this example that we should always reference the server's result

through the `data_ptr` pointer, and not through our variables whose addresses were passed in `rbuf`. That is, in our example, we should reference the long integer result as `*(long *) arg.data_ptr)` and not as `oval` (which we did in Figure 15.2).

This new buffer is allocated by `mmap` and can be returned to the system using `munmap`. The client can also just keep using this buffer for subsequent calls to `door_call`.

`door_cred` Function and Client Credentials

This time, we make one change to our `servproc` function from Figure 15.3: we call the `door_cred` function to obtain the client credentials. Figure 15.8 shows the new server procedure; the client and the server `main` function do not change from Figures 15.2 and 15.3.

doors/server4.c

```
 1 #include    "unpipc.h"
 2 void
 3 servproc(void *cookie, char *dataptr, size_t datasize,
 4          door_desc_t *descptr, size_t ndesc)
 5 {
 6     long    arg, result;
 7     door_cred_t info;
 8         /* obtain and print client credentials */
 9     Door_cred(&info);
10     printf("euid = %ld, ruid = %ld, pid = %ld\n",
11            (long) info.dc_euid, (long) info.dc_ruid, (long) info.dc_pid);
12     arg = *((long *) dataptr);
13     result = arg * arg;
14     Door_return((char *) &result, sizeof(result), NULL, 0);
15 }
```

doors/server4.c

Figure 15.8 Server procedure that obtains and prints client credentials.

We first run the client and will see that the effective user ID equals the real user ID, as we expect. We then become the superuser, change the owner of the executable file to `root`, enable the set-user-ID bit, and run the client again.

```
solaris % client4 /tmp/server4 77      first run of client
result: 5929

solaris % su                           become superuser
Password:
Sun Microsystems Inc.   SunOS 5.6       Generic August 1997
solaris # cd directory containing executable
solaris # ls -l client4
-rwxrwxr-x   1 rstevens other1    139328 Apr 13 06:02 client4
solaris # chown root client4           change owner to root
solaris # chmod u+s client4            and turn on the set-user-ID bit
solaris # ls -l client4                check file permissions and owner
-rwsrwxr-x   1 root     other1    139328 Apr 13 06:02 client4
solaris # exit
```

```
solaris % ls -l client4
-rwsrwxr-x   1 root     other1     139328 Apr 13 06:02 client4
solaris % client4 /tmp/server4 77     and run the client again
result: 5929
```

If we look at the server output, we can see the change in the effective user ID the second time we ran the client.

```
solaris % server4 /tmp/server4
euid = 224, ruid = 224, pid = 3168
euid = 0, ruid = 224, pid = 3176
```

The effective user ID of 0 means the superuser.

Automatic Thread Management by Server

To see the thread management performed by the server, we have the server procedure print its thread ID when the procedure starts executing, and then we have it sleep for 5 seconds, to simulate a long running server procedure. The sleep lets us start multiple clients while an existing client is being serviced. Figure 15.9 shows the new server procedure.

doors/server5.c

```
 1 #include    "unpipc.h"

 2 void
 3 servproc(void *cookie, char *dataptr, size_t datasize,
 4           door_desc_t *descptr, size_t ndesc)
 5 {
 6     long    arg, result;

 7     arg = *((long *) dataptr);
 8     printf("thread id %ld, arg = %ld\n", pr_thread_id(NULL), arg);
 9     sleep(5);

10     result = arg * arg;
11     Door_return((char *) &result, sizeof(result), NULL, 0);
12 }
```

doors/server5.c

Figure 15.9 Server procedure that prints thread ID and sleeps.

We introduce a new function from our library, `pr_thread_id`. It has one argument (a pointer to a thread ID or a null pointer to use the calling thread's ID) and returns a `long` integer identifier for this thread (often a small integer). A process can always be identified by an integer value, its process ID. Even though we do not know whether the process ID is an `int` or a `long`, we just cast the return value from `getpid` to a `long` and print the value (Figure 9.2). But the identifier for a thread is a `pthread_t` datatype (called a thread ID), and this need not be an integer. Indeed, Solaris 2.6 uses small integers as the thread ID, whereas Digital Unix uses pointers. Often, however, we want to print a small integer identifier for a thread (as in this example) for debugging purposes. Our library function, shown in Figure 15.10, handles this problem.

lib/wrappthread.c

```
245 long
246 pr_thread_id(pthread_t * ptr)
247 {
248 #if defined(sun)
249     return ((ptr == NULL) ? pthread_self() : *ptr);     /* Solaris */
250 #elif defined(__osf__) && defined(__alpha)
251     pthread_t tid;
252     tid = (ptr == NULL) ? pthread_self() : *ptr;    /* Digital Unix */
253     return (pthread_getsequence_np(tid));
254 #else
255         /* everything else */
256     return ((ptr == NULL) ? pthread_self() : *ptr);
257 #endif
258 }
```

lib/wrappthread.c

Figure 15.10 `pr_thread_id` function: return small integer identifier for calling thread.

If the implementation does not provide a small integer identifier for a thread, the function could be more sophisticated, mapping the `pthread_t` values to small integers and remembering this mapping (in an array or linked list) for future calls. This is done in the `thread_name` function in [Lewis and Berg 1998].

Returning to Figure 15.9, we run the client three times in a row. Since we wait for the shell prompt before starting the next client, we know that the 5-second wait is complete at the server each time.

```
solaris % client5 /tmp/server5 55
result: 3025
solaris % client5 /tmp/server5 66
result: 4356
solaris % client5 /tmp/server5 77
result: 5929
```

Looking at the server output, we see that the same server thread services each client:

```
solaris % server5 /tmp/server5
thread id 4, arg = 55
thread id 4, arg = 66
thread id 4, arg = 77
```

We now start three clients at the same time:

```
solaris % client5 /tmp/server5 11 & client5 /tmp/server5 22 & \
client5 /tmp/server5 33 &
[2]     3812
[3]     3813
[4]     3814
solaris % result: 484
result: 121
result: 1089
```

The server output shows that two new threads are created to handle the second and third invocations of the server procedure:

```
thread id 4, arg = 22
thread id 5, arg = 11
thread id 6, arg = 33
```

We then start two more clients at the same time:

```
solaris % client5 /tmp/server5 11 & client5 /tmp/server5 22 &
[2]     3830
[3]     3831
solaris % result: 484
result: 121
```

and see that the server uses the previously created threads:

```
thread id 6, arg = 22
thread id 5, arg = 11
```

What we can see with this example is that the server process (i.e., the doors library that is linked with our server code) automatically creates server threads as they are needed. If an application wants to handle the thread management itself, it can, using the functions that we describe in Section 15.9.

We have also verified that the server procedure is a *concurrent* server: multiple instances of the same server procedure can be running at the same time, as separate threads, servicing different clients. Another way we know that the server is concurrent is that when we run three clients at the same time, all three results are printed 5 seconds later. If the server were *iterative*, one result would be printed 5 seconds after all three clients were started, the next result 5 seconds later, and the last result 5 seconds later.

Automatic Thread Management by Server: Multiple Server Procedures

The previous example had only one server procedure in the server process. Our next question is whether multiple server procedures in the same process can use the same thread pool. To test this, we add another server procedure to the server process and also recode this example to show a better style for handling the arguments and results between different processes.

Our first file is a header named `squareproc.h` that defines one datatype for the input arguments to our square function and one datatype for the output arguments. It also defines the pathname for this procedure. We show this in Figure 15.11.

Our new procedure takes a long integer input value and returns a `double` containing the square root of the input. We define the pathname, input structure, and output structure in our `sqrtproc.h` header, which we show in Figure 15.12.

We show our client program in Figure 15.13. It just calls the two procedures, one after the other, and prints the result. This program is similar to the other client programs that we have shown in this chapter.

Our two server procedures are shown in Figure 15.14. Each prints its thread ID and argument, sleeps for 5 seconds, computes the result, and returns.

The `main` function, shown in Figure 15.15, opens two door descriptors and associates each one with one of the two server procedures.

doors/squareproc.h

```
1 #define PATH_SQUARE_DOOR    "/tmp/squareproc_door"

2 typedef struct {                  /* input to squareproc() */
3     long    arg1;
4 } squareproc_in_t;

5 typedef struct {                  /* output from squareproc() */
6     long    res1;
7 } squareproc_out_t;
```

doors/squareproc.h

Figure 15.11 `squareproc.h` header.

doors/sqrtproc.h

```
1 #define PATH_SQRT_DOOR  "/tmp/sqrtproc_door"

2 typedef struct {                  /* input to sqrtproc() */
3     long    arg1;
4 } sqrtproc_in_t;

5 typedef struct {                  /* output from sqrtproc() */
6     double  res1;
7 } sqrtproc_out_t;
```

doors/sqrtproc.h

Figure 15.12 `sqrtproc.h` header.

doors/client7.c

```
1 #include    "unpipc.h"
2 #include    "squareproc.h"
3 #include    "sqrtproc.h"

4 int
5 main(int argc, char **argv)
6 {
7     int     fdsquare, fdsqrt;
8     door_arg_t arg;
9     squareproc_in_t square_in;
10    squareproc_out_t square_out;
11    sqrtproc_in_t sqrt_in;
12    sqrtproc_out_t sqrt_out;

13    if (argc != 2)
14        err_quit("usage: client7 <integer-value>");

15    fdsquare = Open(PATH_SQUARE_DOOR, O_RDWR);
16    fdsqrt = Open(PATH_SQRT_DOOR, O_RDWR);

17        /* set up the arguments and call squareproc() */
18    square_in.arg1 = atol(argv[1]);
19    arg.data_ptr = (char *) &square_in;
20    arg.data_size = sizeof(square_in);
21    arg.desc_ptr = NULL;
22    arg.desc_num = 0;
23    arg.rbuf = (char *) &square_out;
24    arg.rsize = sizeof(square_out);
25    Door_call(fdsquare, &arg);
```

```
26          /* set up the arguments and call sqrtproc() */
27      sqrt_in.arg1 = atol(argv[1]);
28      arg.data_ptr = (char *) &sqrt_in;
29      arg.data_size = sizeof(sqrt_in);
30      arg.desc_ptr = NULL;
31      arg.desc_num = 0;
32      arg.rbuf = (char *) &sqrt_out;
33      arg.rsize = sizeof(sqrt_out);
34      Door_call(fdsqrt, &arg);
35      printf("result: %ld %g\n", square_out.res1, sqrt_out.res1);
36      exit(0);
37  }
```

doors/client7.c

Figure 15.13 Client program that calls our square and square root procedures.

doors/server7.c

```
 1  #include    "unpipc.h"
 2  #include    <math.h>
 3  #include    "squareproc.h"
 4  #include    "sqrtproc.h"
 5  void
 6  squareproc(void *cookie, char *dataptr, size_t datasize,
 7             door_desc_t *descptr, size_t ndesc)
 8  {
 9      squareproc_in_t in;
10      squareproc_out_t out;
11      memcpy(&in, dataptr, min(sizeof(in), datasize));
12      printf("squareproc: thread id %ld, arg = %ld\n",
13             pr_thread_id(NULL), in.arg1);
14      sleep(5);
15      out.res1 = in.arg1 * in.arg1;
16      Door_return((char *) &out, sizeof(out), NULL, 0);
17  }
18  void
19  sqrtproc(void *cookie, char *dataptr, size_t datasize,
20           door_desc_t *descptr, size_t ndesc)
21  {
22      sqrtproc_in_t in;
23      sqrtproc_out_t out;
24      memcpy(&in, dataptr, min(sizeof(in), datasize));
25      printf("sqrtproc: thread id %ld, arg = %ld\n",
26             pr_thread_id(NULL), in.arg1);
27      sleep(5);
28      out.res1 = sqrt((double) in.arg1);
29      Door_return((char *) &out, sizeof(out), NULL, 0);
30  }
```

doors/server7.c

Figure 15.14 Two server procedures.

doors/server7.c

```
31 int
32 main(int argc, char **argv)
33 {
34      int     fd;

35      if (argc != 1)
36          err_quit("usage: server7");

37      fd = Door_create(squareproc, NULL, 0);
38      unlink(PATH_SQUARE_DOOR);
39      Close(Open(PATH_SQUARE_DOOR, O_CREAT | O_RDWR, FILE_MODE));
40      Fattach(fd, PATH_SQUARE_DOOR);

41      fd = Door_create(sqrtproc, NULL, 0);
42      unlink(PATH_SQRT_DOOR);
43      Close(Open(PATH_SQRT_DOOR, O_CREAT | O_RDWR, FILE_MODE));
44      Fattach(fd, PATH_SQRT_DOOR);

45      for ( ; ; )
46          pause();
47 }
```

doors/server7.c

Figure 15.15 `main` function.

If we run the client, it takes 10 seconds to print the results (as we expect).

```
solaris % client7 77
result: 5929 8.77496
```

If we look at the server output, we see that the same thread in the server process handles both client requests.

```
solaris % server7
squareproc: thread id 4, arg = 77
sqrtproc: thread id 4, arg = 77
```

This tells us that any thread in the pool of server threads for a given process can handle a client request for any server procedure.

DOOR_UNREF Attribute for Servers

We mentioned in Section 15.3 that the `DOOR_UNREF` attribute can be specified to `door_create` as an attribute of a newly created door. The manual page says that when the number of descriptors referring to the door drops to one (that is, the reference count goes from two to one), a special invocation is made of the door's server procedure. What is special is that the second argument to the server procedure (the pointer to the data arguments) is the constant `DOOR_UNREF_DATA`. We will demonstrate three ways in which the door is referenced.

1. The descriptor returned by `door_create` in the server counts as one reference. In fact, the reason that the trigger for an unreferenced procedure is the transition of the reference count from two to one, and not from one to 0, is that the server process normally keeps this descriptor open for the duration of the process.

2. The pathname attached to the door in the filesystem also counts as one reference. We can remove this reference by calling the `fdetach` function, running the `fdetach` program, or unlinking the pathname from the filesystem (either the `unlink` function or the `rm` command).
3. The descriptor returned by `open` in the client counts as an open reference until the descriptor is closed, either explicitly by calling `close` or implicitly by the termination of the client process. In all the client processes that we have shown in this chapter, this close is implicit.

Our first example shows that if the server closes its door descriptor after calling `fattach`, an unreferenced invocation of the server procedure occurs immediately. Figure 15.16 shows our server procedure and the server `main` function.

doors/serverunref1.c

```
 1 #include    "unpipc.h"
 2 void
 3 servproc(void *cookie, char *dataptr, size_t datasize,
 4           door_desc_t *descptr, size_t ndesc)
 5 {
 6     long    arg, result;
 7     if (dataptr == DOOR_UNREF_DATA) {
 8         printf("door unreferenced\n");
 9         Door_return(NULL, 0, NULL, 0);
10     }
11     arg = *((long *) dataptr);
12     printf("thread id %ld, arg = %ld\n", pr_thread_id(NULL), arg);
13     sleep(6);
14     result = arg * arg;
15     Door_return((char *) &result, sizeof(result), NULL, 0);
16 }
17 int
18 main(int argc, char **argv)
19 {
20     int     fd;
21     if (argc != 2)
22         err_quit("usage: server1 <server-pathname>");
23         /* create a door descriptor and attach to pathname */
24     fd = Door_create(servproc, NULL, DOOR_UNREF);
25     unlink(argv[1]);
26     Close(Open(argv[1], O_CREAT | O_RDWR, FILE_MODE));
27     Fattach(fd, argv[1]);
28     Close(fd);
29         /* servproc() handles all client requests */
30     for ( ; ; )
31         pause();
32 }
```

doors/serverunref1.c

Figure 15.16 Server procedure that handles an unreferenced invocation.

7–10 Our server procedure recognizes the special invocation and prints a message. The thread returns from this special call by calling `door_return` with two null pointers and two sizes of 0.

28 We now `close` the door descriptor after `fattach` returns. The only use that the server has for this descriptor after `fattach` is if it needs to call `door_bind`, `door_info`, or `door_revoke`.

When we start the server, we notice that the unreferenced invocation occurs immediately:

```
solaris % serverunref1 /tmp/door1
door unreferenced
```

If we follow the reference count for this door, it becomes one after `door_create` returns and then two after `fattach` returns. The server's call to `close` reduces the count from two to one, triggering the unreferenced invocation. The only reference left for this door is its pathname in the filesystem, and that is what the client needs to refer to this door. That is, the client continues to work fine:

```
solaris % clientunref1 /tmp/door1 11
result: 121
solaris % clientunref1 /tmp/door1 22
result: 484
```

Furthermore, no further unreferenced invocations of the server procedure occur. Indeed, only one unreferenced invocation is delivered for a given door.

We now change our server back to the common scenario in which it does not `close` its door descriptor. We show the server procedure and the server `main` function in Figure 15.17. We leave in the 6-second sleep and also print when the server procedure returns. We start the server in one window, and then from another window we verify that the door's pathname exists in the filesystem and then remove the pathname with `rm`:

```
solaris % ls -l /tmp/door2
Drw-r--r--   1 rstevens other1          0 Apr 16 08:58 /tmp/door2
solaris % rm /tmp/door2
```

As soon at the pathname is removed, the unreferenced invocation is made of the server procedure:

```
solaris % serverunref2 /tmp/door2
door unreferenced                    as soon as pathname is removed from filesystem
```

If we follow the reference count for this door, it becomes one after `door_create` returns and then two after `fattach` returns. When we `rm` the pathname, this command reduces the count from two to one, triggering the unreferenced invocation.

In our final example of this attribute, we again remove the pathname from the filesystem, but only after starting three client invocations of the door. What we show is that each client invocation increases the reference count, and only when all three clients

doors/serverunref2.c

```
 1 #include    "unpipc.h"

 2 void
 3 servproc(void *cookie, char *dataptr, size_t datasize,
 4           door_desc_t *descptr, size_t ndesc)
 5 {
 6     long    arg, result;

 7     if (dataptr == DOOR_UNREF_DATA) {
 8         printf("door unreferenced\n");
 9         Door_return(NULL, 0, NULL, 0);
10     }
11     arg = *((long *) dataptr);
12     printf("thread id %ld, arg = %ld\n", pr_thread_id(NULL), arg);
13     sleep(6);

14     result = arg * arg;
15     printf("thread id %ld returning\n", pr_thread_id(NULL));
16     Door_return((char *) &result, sizeof(result), NULL, 0);
17 }

18 int
19 main(int argc, char **argv)
20 {
21     int     fd;

22     if (argc != 2)
23         err_quit("usage: server1 <server-pathname>");

24         /* create a door descriptor and attach to pathname */
25     fd = Door_create(servproc, NULL, DOOR_UNREF);

26     unlink(argv[1]);
27     Close(Open(argv[1], O_CREAT | O_RDWR, FILE_MODE));
28     Fattach(fd, argv[1]);

29         /* servproc() handles all client requests */
30     for ( ; ; )
31         pause();
32 }
```

doors/serverunref2.c

Figure 15.17 Server that does not `close` its door descriptor.

terminate does the unreferenced invocation take place. We use our previous server from Figure 15.17, and our client is unchanged from Figure 15.2.

```
solaris % clientunref2 /tmp/door2 44 & clientunref2 /tmp/door2 55 & \
clientunref2 /tmp/door2 55 &
[2]     13552
[3]     13553
[4]     13554
solaris % rm /tmp/door2                    while the three clients are running
solaris % result: 1936
result: 3025
result: 4356
```

Here is the server output:

```
solaris % serverunref2 /tmp/door2
thread id 4, arg = 44
thread id 5, arg = 55
thread id 6, arg = 66
thread id 4 returning
thread id 5 returning
thread id 6 returning
door unreferenced
```

If we follow the reference count for this door, it becomes one after `door_create` returns and then two after `fattach` returns. As each client calls `open`, the reference count is incremented, going from two to three, from three to four, and then from four to five. When we `rm` the pathname, the count reduces from five to four. Then as each client terminates, the count goes from four to three, then three to two, then two to one, and this final decrement triggers the unreferenced invocation.

What we have shown with these examples is that even though the description of the `DOOR_UNREF` attribute is simple ("the unreferenced invocation occurs when the reference count goes from two to one"), we must understand this reference count to use this feature.

15.8 Descriptor Passing

When we think of passing an open descriptor from one process to another, we normally think of either

- a child sharing all the open descriptors with the parent after a call to `fork`, or
- all descriptors normally remaining open when `exec` is called.

In the first example, the process opens a descriptor, calls `fork`, and then the parent closes the descriptor, letting the child handle the descriptor. This passes an open descriptor from the parent to the child.

Current Unix systems extend this notion of descriptor passing and provide the ability to pass any open descriptor from one process to any other process, related or unrelated. Doors provide one API for the passing of descriptors from the client to the server, and from the server to the client.

> We described descriptor passing using Unix domain sockets in Section 14.7 of UNPv1. Berkeley-derived kernels pass descriptors using these sockets, and all the details are provided in Chapter 18 of TCPv3. SVR4 kernels use a different technique to pass a descriptor, the `I_SENDFD` and `I_RECVFD` `ioctl` commands, described in Section 15.5.1 of APUE. But an SVR4 process can still access this kernel feature using a Unix domain socket.

Be sure to understand what we mean by *passing a descriptor*. In Figure 4.7, the server opens the file and then copies the entire file across the bottom pipe. If the file's size is 1 megabyte, then 1 megabyte of data goes across the bottom pipe from the server to the client. But if the server passes a descriptor back to the client, instead of the file

itself, then only the descriptor is passed across the bottom pipe in Figure 4.7 (which we assume is some small amount of kernel-specific information). The client then takes this descriptor and reads the file, writing its contents to standard output. All the file reading takes place in the client, and the server only opens the file.

Realize that the server *cannot* just write the descriptor number across the bottom pipe in Figure 4.7, as in

```
int    fd;

fd = Open( ... );
Write(pipefd, &fd, sizeof(int));
```

This approach does not work. Descriptor numbers are a per-process attribute. Suppose the value of `fd` is 4 in the server. Even if this descriptor is open in the client, it almost certainly does not refer to the same file as descriptor 4 in the server process. (The only time descriptor numbers mean something from one process to another is across a `fork` or across an `exec`.) If the lowest unused descriptor in the server is 4, then a successful `open` in the server will return 4. If the server "passes" its descriptor 4 to the client and the lowest unused descriptor in the client is 7, then we want descriptor 7 in the client to refer to the same file as descriptor 4 in the server. Figures 15.4 of APUE and 18.4 of TCPv3 show what must happen from the kernel's perspective: the two descriptors (4 in the server and 7 in the client, in our example) must both point to the same file table entry within the kernel. Some kernel black magic is involved in descriptor passing, but APIs like doors and Unix domain sockets hide all these internal details, allowing processes to pass descriptors easily from one process to another.

Descriptors are passed across a door from the client to server by setting the `desc_ptr` member of the `door_arg_t` structure to point to an array of `door_desc_t` structures, and setting `door_num` to the number of these structures. Descriptors are passed from the server to the client by setting the third argument of `door_return` to point to an array of `door_desc_t` structures, and setting the fourth argument to the number of descriptors being passed.

```
typedef struct door_desc {
  door_attr_t d_attributes;     /* tag for union */
  union {
    struct {                    /* valid if tag = DOOR_DESCRIPTOR */
      int        d_descriptor;  /* descriptor number */
      door_id_t  d_id;          /* unique id */
    } d_desc;
  } d_data;
} door_desc_t;
```

This structure contains a `union`, and the first member of the structure is a tag that identifies what is contained in the `union`. But currently only one member of the `union` is defined (a `d_desc` structure that describes a descriptor), and the tag (`d_attributes`) must be set to `DOOR_DESCRIPTOR`.

Example

We modify our file server example (recall Figure 1.9) so that the server opens the file, passes the open descriptor to the client, and the client then copies the file to standard output. Figure 15.18 shows the arrangement.

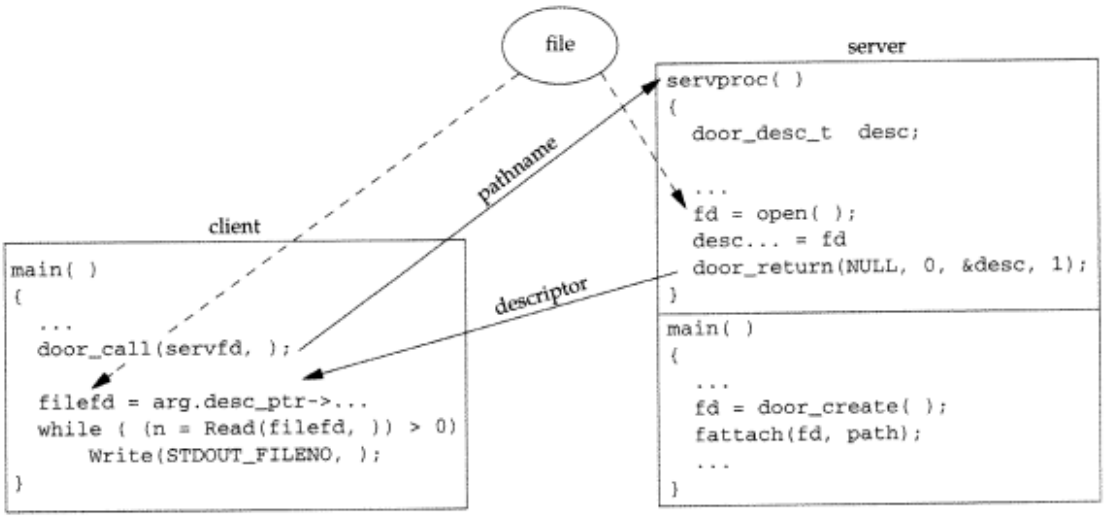

Figure 15.18 File server example with server passing back open descriptor.

Figure 15.19 shows the client program.

Open door, read pathname from standard input

9-15 The pathname associated with the door is a command-line argument and the door is opened. The filename that the client wants opened is read from standard input and the trailing newline is deleted.

Set up arguments and pointer to result

16-22 The `door_arg_t` structure is set up. We add one to the size of the pathname to allow the server to null terminate the pathname.

Call server procedure and check result

23-31 We call the server procedure and then check that the result is what we expect: no data and one descriptor. We will see shortly that the server returns data (containing an error message) only if it cannot open the file, in which case, our call to `err_quit` prints that error.

Fetch descriptor and copy file to standard output

32-34 The descriptor is fetched from the `door_desc_t` structure, and the file is copied to standard output.

doors/clientfd1.c

```
 1 #include    "unpipc.h"

 2 int
 3 main(int argc, char **argv)
 4 {
 5     int     door, fd;
 6     char    argbuf[BUFFSIZE], resbuf[BUFFSIZE], buff[BUFFSIZE];
 7     size_t  len, n;
 8     door_arg_t arg;

 9     if (argc != 2)
10         err_quit("usage: clientfd1 <server-pathname>");

11     door = Open(argv[1], O_RDWR);    /* open the door */

12     Fgets(argbuf, BUFFSIZE, stdin);      /* read pathname of file to open */
13     len = strlen(argbuf);
14     if (argbuf[len - 1] == '\n')
15         len--;                   /* delete newline from fgets() */

16         /* set up the arguments and pointer to result */
17     arg.data_ptr = argbuf;       /* data argument */
18     arg.data_size = len + 1;     /* size of data argument */
19     arg.desc_ptr = NULL;
20     arg.desc_num = 0;
21     arg.rbuf = resbuf;           /* data results */
22     arg.rsize = BUFFSIZE;        /* size of data results */

23     Door_call(door, &arg);       /* call server procedure */

24     if (arg.data_size != 0)
25         err_quit("%.*s", arg.data_size, arg.data_ptr);
26     else if (arg.desc_ptr == NULL)
27         err_quit("desc_ptr is NULL");
28     else if (arg.desc_num != 1)
29         err_quit("desc_num = %d", arg.desc_num);
30     else if (arg.desc_ptr->d_attributes != DOOR_DESCRIPTOR)
31         err_quit("d_attributes = %d", arg.desc_ptr->d_attributes);

32     fd = arg.desc_ptr->d_data.d_desc.d_descriptor;
33     while ( (n = Read(fd, buff, BUFFSIZE)) > 0)
34         Write(STDOUT_FILENO, buff, n);

35     exit(0);
36 }
```

doors/clientfd1.c

Figure 15.19 Client program for descriptor passing file server example.

Figure 15.20 shows the server procedure. The server `main` function has not changed from Figure 15.3.

Open file for client

9-14 We null terminate the client's pathname and try to `open` the file. If an error occurs, the data result is a string containing the error message.

doors/serverfd1.c

```
1 #include    "unpipc.h"
2 void
3 servproc(void *cookie, char *dataptr, size_t datasize,
4           door_desc_t *descptr, size_t ndesc)
5 {
6     int     fd;
7     char    resbuf[BUFFSIZE];
8     door_desc_t desc;
9     dataptr[datasize - 1] = 0;  /* null terminate */
10    if ( (fd = open(dataptr, O_RDONLY)) == -1) {
11            /* error: must tell client */
12        snprintf(resbuf, BUFFSIZE, "%s: can't open, %s",
13                 dataptr, strerror(errno));
14        Door_return(resbuf, strlen(resbuf), NULL, 0);
15    } else {
16            /* open succeeded: return descriptor */
17        desc.d_data.d_desc.d_descriptor = fd;
18        desc.d_attributes = DOOR_DESCRIPTOR;
19        Door_return(NULL, 0, &desc, 1);
20    }
21 }
```

doors/serverfd1.c

Figure 15.20 Server procedure that opens a file and passes back its descriptor.

Success

15-20 If the open succeeds, only the descriptor is returned; there are no data results.

We start the server and specify its door pathname as /tmp/fd1 and then run the client:

```
solaris % clientfd1 /tmp/fd1
/etc/shadow
/etc/shadow: can't open, Permission denied
solaris % clientfd1 /tmp/fd1
/no/such/file
/no/such/file: can't open, No such file or directory
solaris % clientfd1 /tmp/fd1
/etc/ntp.conf                                   a 2-line file
multicastclient 224.0.1.1
driftfile /etc/ntp.drift
```

The first two times, we specify a pathname that causes an error return, and the third time, the server returns the descriptor for a 2-line file.

There is a problem with descriptor passing across a door. To see the problem in our example, just add a printf to the server procedure after a successful open. You will see that each descriptor value is one greater than the previous descriptor value. The problem is that the server is not closing the descriptors after it passes them to the client. But there is no easy way to do this. The logical place to perform the close would be after door_return returns, once the descriptor has been sent to the client, but door_return does not return! If we had been

using either `sendmsg` to pass the descriptor across a Unix domain socket, or `ioctl` to pass the descriptor across an SVR4 pipe, we could `close` the descriptor when `sendmsg` or `ioctl` returns. But the doors paradigm for passing descriptors is different from these two techniques, since no return occurs from the function that passes the descriptor. The only way around this problem is for the server procedure to somehow remember that it has a descriptor open and close it at some later time, which becomes very messy.

This problem should be fixed in Solaris 2.7 with the addition of a new `DOOR_RELEASE` attribute. The sender sets `d_attributes` to `DOOR_DESCRIPTOR | DOOR_RELEASE`, which tells the system to close the descriptor after passing it to the receiver.

15.9 `door_server_create` Function

We showed with Figure 15.9 that the doors library automatically creates new threads as needed to handle the client requests as they arrive. These are created by the library as detached threads, with the default thread stack size, with thread cancellation disabled, and with a signal mask and scheduling class that are initially inherited from the thread that called `door_create`. If we want to change any of these features or if we want to manage the pool of server threads ourselves, we call `door_server_create` and specify our own *server creation procedure*.

```
#include <door.h>

typedef void Door_create_proc(door_info_t *);

Door_create_proc *door_server_create(Door_create_proc *proc);
```

Returns: pointer to previous server creation procedure

As with our declaration of `door_create` in Section 15.3, we use C's `typedef` to simplify the function prototype for the library function. Our new datatype defines a server creation procedure as taking a single argument (a pointer to a `door_info_t` structure), and returning nothing (`void`). When we call `door_server_create`, the argument is a pointer to our server creation procedure, and the return value is a pointer to the previous server creation procedure.

Our server creation procedure is called whenever a new thread is needed to service a client request. Information on which server procedure needs the thread is in the `door_info_t` structure whose address is passed to the creation procedure. The `di_proc` member contains the address of the server procedure, and `di_data` contains the cookie pointer that is passed to the server procedure each time it is called.

An example is the easiest way to see what is happening. Our client does not change from Figure 15.2. In our server, we add two new functions in addition to our server procedure function and our server `main` function. Figure 15.21 shows an overview of the four functions in our server process, when some are registered, and when they are all called.

Figure 15.22 shows the server `main` function.

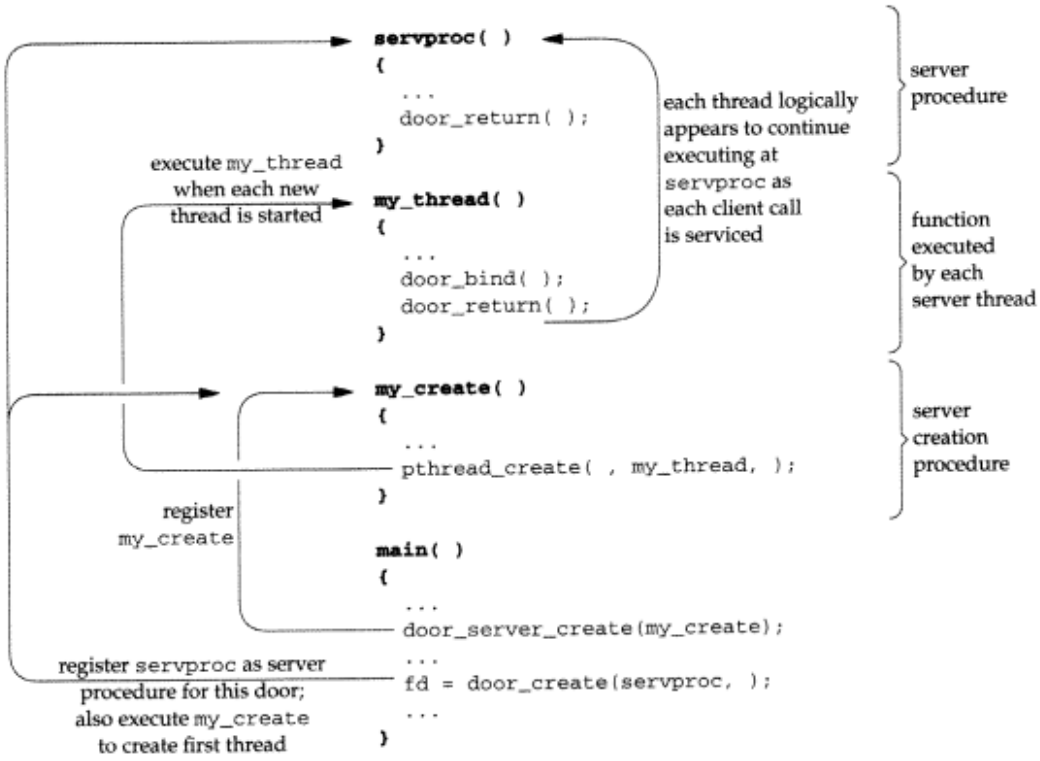

Figure 15.21 Overview of the four functions in our server process.

doors/server6.c

```
42 int
43 main(int argc, char **argv)
44 {
45     if (argc != 2)
46         err_quit("usage: server6 <server-pathname>");
47     Door_server_create(my_create);
48         /* create a door descriptor and attach to pathname */
49     Pthread_mutex_lock(&fdlock);
50     fd = Door_create(servproc, NULL, DOOR_PRIVATE);
51     Pthread_mutex_unlock(&fdlock);
52     unlink(argv[1]);
53     Close(Open(argv[1], O_CREAT | O_RDWR, FILE_MODE));
54     Fattach(fd, argv[1]);
55         /* servproc() handles all client requests */
56     for ( ; ; )
57         pause();
58 }
```

doors/server6.c

Figure 15.22 `main` function for example of thread pool management.

We have made four changes from Figure 15.3: (1) the declaration of the door descriptor `fd` is gone (it is now a global variable that we show and describe in Figure 15.23), (2) we protect the call to `door_create` with a mutex (which we also describe in Figure 15.23), (3) we call `door_server_create` before creating the door, specifying our server creation procedure (`my_thread`, which we show next), and (4) in the call to `door_create`, the final argument (the attributes) is now `DOOR_PRIVATE` instead of 0. This tells the library that this door will have its own pool of threads, called a *private server pool*.

Specifying a private server pool with `DOOR_PRIVATE` and specifying a server creation procedure with `door_server_create` are independent. Four scenarios are possible.

1. Default: *no* private server pools and *no* server creation procedure. The system creates threads as needed, and they all go into the process-wide thread pool.
2. `DOOR_PRIVATE` and *no* server creation procedure. The system creates threads as needed, and they go into the process-wide pool for doors created without `DOOR_PRIVATE` or into a door's private server pool for doors created with `DOOR_PRIVATE`.
3. *No* private server pools, but a server creation procedure is specified. The server creation procedure is called whenever a new thread is needed, and these threads all go into the process-wide thread pool.
4. `DOOR_PRIVATE` and a server creation procedure are both specified. The server creation procedure is called whenever a new thread is needed. When a thread is created, it should call `door_bind` to assign itself to the appropriate private server pool, or the thread will be assigned to the process-wide pool.

Figure 15.23 shows our two new functions: `my_create` is our server creation procedure, and it calls `my_thread` as the function that is executed by each thread that it creates.

Server creation procedure

30-41 Each time `my_create` is called, we create a new thread. But before calling `pthread_create`, we initialize its attributes, set the contention scope to `PTHREAD_SCOPE_SYSTEM`, and specify the thread as a detached thread. The thread is created and starts executing the `my_thread` function. The argument to this function is a pointer to the `door_info_t` structure. If we have a server with multiple doors and we specify a server creation procedure, this one server creation procedure is called when a new thread is needed for any of the doors. The only way for this server creation procedure and the thread start function that it specifies to `pthread_create` to differentiate between the different server procedures is to look at the `di_proc` pointer in the `door_info_t` structure.

Setting the contention scope to `PTHREAD_SCOPE_SYSTEM` means this thread will compete for processor resources against threads in other processes. The alternative,

doors/server6.c

```
13 pthread_mutex_t fdlock = PTHREAD_MUTEX_INITIALIZER;
14 static int fd = -1;            /* door descriptor */

15 void *
16 my_thread(void *arg)
17 {
18     int     oldstate;
19     door_info_t *iptr = arg;

20     if ((Door_server_proc *) iptr->di_proc == servproc) {
21         Pthread_mutex_lock(&fdlock);
22         Pthread_mutex_unlock(&fdlock);

23         Pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, &oldstate);
24         Door_bind(fd);
25         Door_return(NULL, 0, NULL, 0);
26     } else
27         err_quit("my_thread: unknown function: %p", arg);
28     return (NULL);              /* never executed */
29 }

30 void
31 my_create(door_info_t *iptr)
32 {
33     pthread_t tid;
34     pthread_attr_t attr;

35     Pthread_attr_init(&attr);
36     Pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);
37     Pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
38     Pthread_create(&tid, &attr, my_thread, (void *) iptr);
39     Pthread_attr_destroy(&attr);
40     printf("my_thread: created server thread %ld\n", pr_thread_id(&tid));
41 }
```

doors/server6.c

Figure 15.23 Our own thread management functions.

PTHREAD_SCOPE_PROCESS, means this thread will compete for processor resources only against other threads in this process. The latter will not work with doors, because the doors library requires that the kernel lightweight process performing the door_return be the same lightweight process that originated the invocation. An unbound thread (PTHREAD_SCOPE_PROCESS) could change lightweight processes during execution of the server procedure.

The reason for requiring that the thread be created as a detached thread is to prevent the system from saving any information about the thread when it terminates, because no one will be calling pthread_join.

Thread start function

15-20 my_thread is the thread start function specified by the call to pthread_create. The argument is the pointer to the door_info_t structure that was passed to my_create. The only server procedure that we have in this process is servproc, and we just verify that the argument references this procedure.

Wait for descriptor to be valid

21-22 The server creation procedure is called for the first time when `door_create` is called, to create an initial server thread. This call is issued from within the doors library before `door_create` returns. But the variable `fd` will not contain the door descriptor until `door_create` returns. (This is a chicken-and-egg problem.) Since we know that `my_thread` is running as a separate thread from the main thread that calls `door_create`, our solution to this timing problem is to use the mutex `fdlock` as follows: the main thread locks the mutex before calling `door_create` and unlocks the mutex when `door_create` returns and a value has been stored into `fd` (Figure 15.22). Our `my_thread` function just locks the mutex (probably blocking until the main thread has unlocked the mutex) and then unlocks it. We could have added a condition variable that the main thread signals, but we don't need it here, since we know the sequence of calls that will occur.

Disable thread cancellation

23 When a new Posix thread is created by `pthread_create`, thread cancellation is enabled by default. When cancellation is enabled, and a client aborts a `door_call` that is in progress (which we will demonstrate in Figure 15.31), the thread cancellation handlers (if any) are called, and the thread is then terminated. When cancellation is disabled (as we are doing here), and a client aborts a `door_call` that is in progress, the server procedure completes (the thread is not terminated), and the results from `door_return` are just discarded. Since the server thread is terminated when cancellation is enabled, and since the server procedure may be in the middle of an operation for the client (it may hold some locks or semaphores), the doors library disables thread cancellation for all the threads that it creates. If a server procedure wants to be canceled when a client terminates prematurely, that thread must enable cancellation and must be prepared to deal with it.

> Notice that the contention scope of `PTHREAD_SCOPE_SYSTEM` and the detached state are specified as attributes when the thread is created. But the cancellation mode can be set only by the thread itself once it is running. Indeed, even though we just disable cancellation, a thread can enable and disable cancellation whenever it wants.

Bind this thread to a door

24 We call `door_bind` to bind the calling thread to the private server pool associated with the door whose descriptor is the argument to `door_bind`. Since we need the door descriptor for this call, we made `fd` a global variable for this version of our server.

Make thread available for a client call

25 The thread makes itself available for incoming door invocations by calling `door_return` with two null pointers and two 0 lengths as the arguments.

We show the server procedure in Figure 15.24. This version is identical to the one in Figure 15.9.

To demonstrate what happens, we just start the server:

```
solaris % server6 /tmp/door6
my_thread: created server thread 4
```

doors/server6.c

```
1 #include    "unpipc.h"

2 void
3 servproc(void *cookie, char *dataptr, size_t datasize,
4           door_desc_t *descptr, size_t ndesc)
5 {
6     long    arg, result;

7     arg = *((long *) dataptr);
8     printf("thread id %ld, arg = %ld\n", pr_thread_id(NULL), arg);
9     sleep(5);

10    result = arg * arg;
11    Door_return((char *) &result, sizeof(result), NULL, 0);
12 }
```

doors/server6.c

Figure 15.24 Server procedure.

As soon as the server starts and `door_create` is called, our server creation procedure is called the first time, even though we have not even started the client. This creates the first thread, which will wait for the first client call. We then run the client three times in a row:

```
solaris % client6 /tmp/door6 11
result: 121
solaris % client6 /tmp/door6 22
result: 484
solaris % client6 /tmp/door6 33
result: 1089
```

If we look at the corresponding server output, another thread is created when the first client call occurs (thread ID 5), and then thread number 4 services each of the client requests. The doors library appears to always keep one extra thread ready.

```
my_thread: created server thread 5
thread id 4, arg = 11
thread id 4, arg = 22
thread id 4, arg = 33
```

We then execute the client three times, all at about the same time in the background.

```
solaris % client6 /tmp/door6 44 & client6 /tmp/door6 55 & \
client6 /tmp/door6 66 &
[2]     4919
[3]     4920
[4]     4921
solaris % result: 1936
result: 4356
result: 3025
```

Looking at the corresponding server output, we see that two new threads are created (thread IDs 6 and 7), and threads 4, 5, and 6 service the three client requests:

```
thread id 4, arg = 44
my_thread: created server thread 6
thread id 5, arg = 66
my_thread: created server thread 7
thread id 6, arg = 55
```

15.10 door_bind, door_unbind, and door_revoke Functions

Three additional functions complete the doors API.

```
#include <door.h>

int door_bind(int fd);

int door_unbind(void);

int door_revoke(int fd);
```

All three return: 0 if OK, −1 on error

We introduced the `door_bind` function in Figure 15.23. It binds the calling thread to the private server pool associated with the door whose descriptor is *fd*. If the calling thread is already bound to some other door, an implicit unbind is performed.

`door_unbind` explicitly unbinds the calling thread from the door to which it has been bound.

`door_revoke` revokes access to the door identified by *fd*. A door descriptor can be revoked only by the process that created the descriptor. Any door invocation that is in progress when this function is called is allowed to complete normally.

15.11 Premature Termination of Client or Server

All our examples so far have assumed that nothing abnormal happens to either the client or server. We now consider what happens when errors occur at either the client or server. Realize that when the client and server are part of the same process (the local procedure call in Figure 15.1), the client does not need to worry about the server crashing and vice versa, because if either crashes the entire process crashes. But when the client and server are distributed to two processes, we must consider what happens if one of the two crashes and how the peer is notified of this failure. This is something we must worry about regardless of whether the client and server are on the same host or on different hosts.

Premature Termination of Server

While the client is blocked in a call to `door_call`, waiting for results, it needs to know if the server thread terminates for some reason. To see what happens, we have the

server procedure thread terminate by calling `thread_exit`. This terminates just this thread, not the entire server process. Figure 15.25 shows the server procedure.

doors/serverintr1.c

```
 1 #include    "unpipc.h"
 2 void
 3 servproc(void *cookie, char *dataptr, size_t datasize,
 4          door_desc_t *descptr, size_t ndesc)
 5 {
 6     long    arg, result;
 7     pthread_exit(NULL);          /* and see what happens at client */
 8     arg = *((long *) dataptr);
 9     result = arg * arg;
10     Door_return((char *) &result, sizeof(result), NULL, 0);
11 }
```

doors/serverintr1.c

Figure 15.25 Server procedure that terminates itself after being invoked.

The remainder of the server does not change from Figure 15.3, and the client does not change from Figure 15.2.

When we run our client, we see that an error of `EINTR` is returned by `door_call` if the server procedure terminates before returning.

```
solaris % clientintr1 /tmp/door1 11
door_call error: Interrupted system call
```

Uninterruptability of `door_call` System Call

The `door_call` manual page warns that this function is not a restartable system call. (The `door_call` function in the doors library invokes a system call of the same name.) We can see this by changing our server so that the server procedure just sleeps for 6 seconds before returning, which we show in Figure 15.26.

doors/serverintr2.c

```
 1 #include    "unpipc.h"
 2 void
 3 servproc(void *cookie, char *dataptr, size_t datasize,
 4          door_desc_t *descptr, size_t ndesc)
 5 {
 6     long    arg, result;
 7     sleep(6);                    /* let client catch SIGCHLD */
 8     arg = *((long *) dataptr);
 9     result = arg * arg;
10     Door_return((char *) &result, sizeof(result), NULL, 0);
11 }
```

doors/serverintr2.c

Figure 15.26 Server procedure sleeps for 6 seconds.

We then modify our client from Figure 15.2 to establish a signal handler for `SIGCHLD`, `fork` a child process, and have the child sleep for 2 seconds and then

terminate. Therefore, about 2 seconds after the client parent calls `door_call`, the parent catches `SIGCHLD` and the signal handler returns, interrupting the `door_call` system call. We show this client in Figure 15.27.

doors/clientintr2.c

```
 1 #include    "unpipc.h"

 2 void
 3 sig_chld(int signo)
 4 {
 5     return;                       /* just interrupt door_call() */
 6 }

 7 int
 8 main(int argc, char **argv)
 9 {
10     int     fd;
11     long    ival, oval;
12     door_arg_t arg;

13     if (argc != 3)
14         err_quit("usage: clientintr2 <server-pathname> <integer-value>");

15     fd = Open(argv[1], O_RDWR); /* open the door */

16         /* set up the arguments and pointer to result */
17     ival = atol(argv[2]);
18     arg.data_ptr = (char *) &ival;  /* data arguments */
19     arg.data_size = sizeof(long);   /* size of data arguments */
20     arg.desc_ptr = NULL;
21     arg.desc_num = 0;
22     arg.rbuf = (char *) &oval;  /* data results */
23     arg.rsize = sizeof(long);   /* size of data results */

24     Signal(SIGCHLD, sig_chld);
25     if (Fork() == 0) {
26         sleep(2);               /* child */
27         exit(0);                /* generates SIGCHLD */
28     }
29         /* parent: call server procedure and print result */
30     Door_call(fd, &arg);
31     printf("result: %ld\n", oval);

32     exit(0);
33 }
```

doors/clientintr2.c

Figure 15.27 Client that catches `SIGCHLD` after 2 seconds.

The client sees the same error as if the server procedure terminated prematurely: `EINTR`.

```
solaris % clientintr2 /tmp/door2 22
door_call error: Interrupted system call
```

This means we must block any signals that might be generated during a call to `door_call` from being delivered to the process, because those signals will interrupt `door_call`.

Idempotent versus Nonidempotent Procedures

What if we know that we just caught a signal, detect the error of `EINTR` from `door_call`, and call the server procedure again, since we know that the error is from our caught signal and not from the server procedure terminating prematurely? This can lead to problems, as we will show.

First, we modify our server to (1) print its thread ID when it is called, (2) sleep for 6 seconds, and (3) print its thread ID when it returns. Figure 15.28 shows this version of our server procedure.

doors/serverintr3.c

```
 1 #include    "unpipc.h"

 2 void
 3 servproc(void *cookie, char *dataptr, size_t datasize,
 4          door_desc_t *descptr, size_t ndesc)
 5 {
 6     long    arg, result;

 7     printf("thread id %ld called\n", pr_thread_id(NULL));
 8     sleep(6);                    /* let client catch SIGCHLD */
 9     arg = *((long *) dataptr);
10     result = arg * arg;
11     printf("thread id %ld returning\n", pr_thread_id(NULL));
12     Door_return((char *) &result, sizeof(result), NULL, 0);
13 }
```

doors/serverintr3.c

Figure 15.28 Server procedure that prints its thread ID when called and when returning.

Figure 15.29 shows our client program.

2-8 We declare the global `caught_sigchld` and set this to one when the `SIGCHLD` signal is caught.

31-42 We now call `door_call` in a loop as long as the error is `EINTR` and this was caused by our signal handler.

If we look at just the client output, it appears OK:

```
solaris % clientintr3 /tmp/door3 33
calling door_call
calling door_call
result: 1089
```

`door_call` is called the first time, our signal handler is invoked about 2 seconds later and `caught_sigchld` is set to one, `door_call` returns `EINTR`, and we call `door_call` again. This second time, the server procedure proceeds to completion and the expected result is returned.

But looking at the server output, we see that the server procedure is called twice.

```
solaris % serverintr3 /tmp/door3
thread id 4 called
thread id 4 returning
thread id 5 called
thread id 5 returning
```

doors/clientintr3.c

```
 1 #include    "unpipc.h"
 2 volatile sig_atomic_t caught_sigchld;
 3 void
 4 sig_chld(int signo)
 5 {
 6     caught_sigchld = 1;
 7     return;                     /* just interrupt door_call() */
 8 }
 9 int
10 main(int argc, char **argv)
11 {
12     int     fd, rc;
13     long    ival, oval;
14     door_arg_t arg;
15     if (argc != 3)
16         err_quit("usage: clientintr3 <server-pathname> <integer-value>");
17     fd = Open(argv[1], O_RDWR); /* open the door */
18         /* set up the arguments and pointer to result */
19     ival = atol(argv[2]);
20     arg.data_ptr = (char *) &ival;  /* data arguments */
21     arg.data_size = sizeof(long);   /* size of data arguments */
22     arg.desc_ptr = NULL;
23     arg.desc_num = 0;
24     arg.rbuf = (char *) &oval;  /* data results */
25     arg.rsize = sizeof(long);   /* size of data results */
26     Signal(SIGCHLD, sig_chld);
27     if (Fork() == 0) {
28         sleep(2);               /* child */
29         exit(0);                /* generates SIGCHLD */
30     }
31         /* parent: call server procedure and print result */
32     for ( ; ; ) {
33         printf("calling door_call\n");
34         if ( (rc = door_call(fd, &arg)) == 0)
35             break;              /* success */
36         if (errno == EINTR && caught_sigchld) {
37             caught_sigchld = 0;
38             continue;           /* call door_call() again */
39         }
40         err_sys("door_call error");
41     }
42     printf("result: %ld\n", oval);
43     exit(0);
44 }
```

doors/clientintr3.c

Figure 15.29 Client that calls `door_call` again after receiving `EINTR`.

When the client calls `door_call` the second time, after the first call is interrupted by the caught signal, this starts another thread that calls the server procedure a second time. If the server procedure is *idempotent*, this is OK. But if the server procedure is not idempotent, this is a problem.

The term *idempotent*, when describing a procedure, means the procedure can be called any number of times without harm. Our server procedure, which calculates the square of a number, is idempotent: we get the correct result whether we call it once or twice. Another example is a procedure that returns the current time and date. Even though this procedure may return different information each time (say it is called twice, 1 second apart, causing the returned times to differ by 1 second), it is still OK. The classic example of a nonidempotent procedure is one that subtracts some amount from a bank account: the end result is wrong unless this procedure is called only once.

Premature Termination of Client

We now see how a server procedure is notified if the client terminates after calling `door_call` but before the server returns. We show our client in Figure 15.30.

doors/clientintr4.c

```
 1 #include    "unpipc.h"

 2 int
 3 main(int argc, char **argv)
 4 {
 5     int     fd;
 6     long    ival, oval;
 7     door_arg_t arg;

 8     if (argc != 3)
 9         err_quit("usage: clientintr4 <server-pathname> <integer-value>");

10     fd = Open(argv[1], O_RDWR); /* open the door */

11         /* set up the arguments and pointer to result */
12     ival = atol(argv[2]);
13     arg.data_ptr = (char *) &ival;  /* data arguments */
14     arg.data_size = sizeof(long);   /* size of data arguments */
15     arg.desc_ptr = NULL;
16     arg.desc_num = 0;
17     arg.rbuf = (char *) &oval;  /* data results */
18     arg.rsize = sizeof(long);   /* size of data results */

19         /* call server procedure and print result */
20     alarm(3);
21     Door_call(fd, &arg);
22     printf("result: %ld\n", oval);

23     exit(0);
24 }
```

doors/clientintr4.c

Figure 15.30 Client that terminates prematurely after calling `door_call`.

20 The only change from Figure 15.2 is the call to `alarm(3)` right before the call to `door_call`. This function schedules a `SIGALRM` signal for 3 seconds in the future, but since we do not catch this signal, its default action terminates the process. This will cause the client to terminate before `door_call` returns, because we will put a 6-second sleep in the server procedure.

Figure 15.31 shows our server procedure and its thread cancellation handler.

doors/serverintr4.c

```
 1 #include    "unpipc.h"

 2 void
 3 servproc_cleanup(void *arg)
 4 {
 5     printf("servproc cancelled, thread id %ld\n", pr_thread_id(NULL));
 6 }

 7 void
 8 servproc(void *cookie, char *dataptr, size_t datasize,
 9           door_desc_t *descptr, size_t ndesc)
10 {
11     int     oldstate, junk;
12     long    arg, result;

13     Pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, &oldstate);
14     pthread_cleanup_push(servproc_cleanup, NULL);
15     sleep(6);
16     arg = *((long *) dataptr);
17     result = arg * arg;
18     pthread_cleanup_pop(0);
19     Pthread_setcancelstate(oldstate, &junk);
20     Door_return((char *) &result, sizeof(result), NULL, 0);
21 }
```

doors/serverintr4.c

Figure 15.31 Server procedure that detects premature termination of client.

Recall our discussion of thread cancellation in Section 8.5 and our discussion of this with Figure 15.23. When the system detects that the client is terminating with a `door_call` in progress, the server thread handling that call is sent a cancellation request.

- If the server thread has cancellation disabled, nothing happens, the thread executes to completion (when it calls `door_return`), and the results are then discarded.
- If cancellation is enabled for the server thread, any cleanup handlers are called, and the thread is then terminated.

In our server procedure, we first call `pthread_setcancelstate` to enable cancellation, because when the doors library creates new threads, it disables thread cancellation. This function also saves the current cancellation state in the variable `oldstate`, and we restore this state at the end of the function. We then call `pthread_cleanup_push` to

register our function `servproc_cleanup` as the cancellation handler. All our function does is print that the thread has been canceled, but this is where a server procedure can do whatever must be done to clean up after the terminated client: release mutexes, write a log file record, or whatever. When our cleanup handler returns, the thread is terminated.

We also put a 6-second sleep in our server procedure, to allow the client to abort while its `door_call` is in progress.

When we run our client twice, we see that the shell prints "Alarm clock" when our process is killed by a `SIGALRM` signal.

```
solaris % clientintr4 /tmp/door4 44
Alarm Clock
solaris % clientintr4 /tmp/door4 44
Alarm Clock
```

If we look at the corresponding server output, we see that each time the client terminates prematurely, the server thread is indeed canceled and our cleanup handler is called.

```
solaris % serverintr4 /tmp/door4
servproc canceled, thread id 4
servproc canceled, thread id 5
```

The reason we ran our client twice is to show that after the thread with an ID of 4 is canceled, a new thread is created by the doors library to handle the second client invocation.

15.12 Summary

Doors provide the ability to call a procedure in another process on the *same* host. In the next chapter we extend this concept of remote procedure calls by describing the calling of a procedure in another process on *another* host.

The basic functions are simple. A server calls `door_create` to create a door and associate it with a server procedure, and then calls `fattach` to attach the door to a pathname in the filesystem. The client calls `open` on this pathname and then `door_call` to call the server procedure in the server process. The server procedure returns by calling `door_return`.

Normally, the only permission testing performed for a door is that done by `open` when it creates the door, based on the client's user IDs and group IDs, along with the permission bits and owner IDs of the pathname. One nice feature of doors that we have not seen with the other forms of IPC in this text is the ability of the server to determine the client's credentials: the client's effective and real user IDs, and effective and real group IDs. These can be used by the server to determine whether it wants to service this client's request.

Doors allow the passing of descriptors from the client to the server and vice versa. This is a powerful technique, because so much in Unix is represented by a descriptor:

access to files for file or device I/O, access to sockets or XTI for network communication (UNPv1), and access to doors for RPC.

When calling procedures in another process, we must worry about premature termination of the peer, something we do not need to worry about with local procedure calls. A doors client is notified if the server thread terminates prematurely by an error return of `EINTR` from `door_call`. A doors server thread is notified if its client terminates while the client is blocked in a call to `door_call` by the receipt of a cancellation request for the server thread. The server thread must decide whether to handle this cancellation or not.

Exercises

15.1 How many bytes of information are passed as arguments by `door_call` from the client to the server?

15.2 In Figure 15.6, do we need to call `fstat` to first verify that the descriptor is a door? Remove this call and see what happens.

15.3 The Solaris 2.6 manual page for `sleep(3C)` states that "The current process is suspended from execution." In Figure 15.9, why is the doors library able to create the second and third threads (thread IDs 5 and 6) once the first thread (ID 4) starts running, since this statement would imply that the entire server process blocks as soon as one thread calls `sleep`?

15.4 In Section 15.3, we said that the `FD_CLOEXEC` bit is automatically set for descriptors created by `door_create`. But we can call `fcntl` after `door_create` returns and turn this bit off. What will happen if we do this, call `exec`, and then invoke the server procedure from a client?

15.5 In Figures 15.28 and 15.29, print the current time in the two calls to `printf` in the server and in the two calls to `printf` in the client. Run the client and server. Why does the first invocation of the server procedure return after 2 seconds?

15.6 Remove the mutex lock that protects `fd` in Figures 15.22 and 15.23 and verify that the program no longer works. What error do you see?

15.7 If the only characteristic of a server thread that we want to change is to enable cancellation, do we need to establish a server creation procedure?

15.8 Verify that `door_revoke` allows a client call that is in progress to complete, and determine what happens to `door_call` once the server procedure has been revoked.

15.9 In our solution to the previous exercise and in Figure 15.22, we said that the door descriptor needs to be a global when either the server procedure or the server creation procedure needs to use the descriptor. That statement is not true. Recode the solution to the previous exercise, keeping `fd` as an automatic variable in the `main` function.

15.10 In Figure 15.23, we call `pthread_attr_init` and `pthread_attr_destroy` every time a thread is created. Is this optimal?

16

Sun RPC

16.1 Introduction

When we build an application, our first choice is whether to

1. build one huge monolithic program that does everything, or
2. distribute the application among multiple processes that communicate with each other.

If we choose the second option, the next choice is whether to

2a. assume that all the processes run on the same host (allowing IPC to be used for communication between the processes), or

2b. assume that some of the processes will run on other hosts (necessitating some form of network communication between the processes).

If we look at Figure 15.1, the top scenario is case 1, the middle scenario is case 2a, and the bottom scenario is case 2b. Most of this text has focused on case (2a): IPC between processes on the same host, using message passing, shared memory, and possibly some form of synchronization. IPC between threads within the same process, or within threads in different processes, is just a special case of this scenario.

When we require network communications among the various pieces of the application, most applications are written using *explicit network programming*, that is, direct calls to either the sockets API or the XTI API, as described in UNPv1. Using the sockets API, clients call `socket`, `connect`, `read`, and `write`, whereas servers call `socket`, `bind`, `listen`, `accept`, `read`, and `write`. Most applications that we are familiar with (Web browsers, Web servers, Telnet clients, Telnet servers, etc.) are written this way.

An alternative way to write a distributed application is to use *implicit network programming*. Remote procedure calls, or RPC, provide such a tool. We code our application using the familiar procedure call, but the calling process (the client) and the process containing the procedure being called (the server) can be executing on different hosts. The fact that the client and server are running on different hosts, and that network I/O is involved in the procedure call, is for the most part transparent. Indeed, one metric by which to measure any RPC package is how transparent it makes the underlying networking.

Example

As an example of RPC, we recode Figures 15.2 and 15.3 to use Sun RPC instead of doors. The client calls the server's procedure with a long integer argument, and the return value is the square of that value. Figure 16.1 is our first file, `square.x`.

sunrpc/square1/square.x

```
1 struct square_in {                    /* input (argument) */
2     long    arg1;
3 };

4 struct square_out {                   /* output (result) */
5     long    res1;
6 };

7 program SQUARE_PROG {
8     version SQUARE_VERS {
9         square_out SQUAREPROC(square_in) = 1;    /* procedure number = 1 */
10    } =     1;                        /* version number */
11 } =     0x31230000;                  /* program number */
```

sunrpc/square1/square.x

Figure 16.1 RPC specification file.

These files whose name end in `.x` are called RPC specification files, and they define the server procedures along with their arguments and results.

Define argument and return value

1-6 We define two structures, one for the arguments (a single `long`), and one for the results (a single `long`).

Define program, version, and procedure

7-11 We define an RPC program named `SQUARE_PROG` that consists of one version (`SQUARE_VERS`), and in that version is a single procedure named `SQUAREPROC`. The argument to this procedure is a `square_in` structure, and its return value is a `square_out` structure. We also assign this procedure a number of 1, we assign the version a value of 1, and we assign the program number a 32-bit hexadecimal value. (We say more about these program numbers in Figure 16.9.)

We compile this specification file using a program supplied with the Sun RPC package, `rpcgen`.

The next program we write is the client `main` function that calls our remote procedure. We show this in Figure 16.2.

sunrpc/square1/client.c

```
 1 #include    "unpipc.h"          /* our header */
 2 #include    "square.h"          /* generated by rpcgen */

 3 int
 4 main(int argc, char **argv)
 5 {
 6     CLIENT *cl;
 7     square_in in;
 8     square_out *outp;

 9     if (argc != 3)
10         err_quit("usage: client <hostname> <integer-value>");

11     cl = Clnt_create(argv[1], SQUARE_PROG, SQUARE_VERS, "tcp");

12     in.arg1 = atol(argv[2]);
13     if ( (outp = squareproc_1(&in, cl)) == NULL)
14         err_quit("%s", clnt_sperror(cl, argv[1]));

15     printf("result: %ld\n", outp->res1);
16     exit(0);
17 }
```

sunrpc/square1/client.c

Figure 16.2 Client `main` function that calls remote procedure.

Include header generated by `rpcgen`

2 We `#include` the `square.h` header that is generated by `rpcgen`.

Declare client handle

6 We declare a *client handle* named `cl`. Client handles are intended to look like standard I/O `FILE` pointers (hence the uppercase name of `CLIENT`).

Obtain client handle

11 We call `clnt_create`, which returns a client handle upon success.

```
#include <rpc/rpc.h>

CLIENT *clnt_create(const char *host, unsigned long prognum,
                    unsigned long versnum, const char *protocol);
```

Returns: nonnull client handle if OK, `NULL` on error

As with standard I/O `FILE` pointers, we don't care what the client handle points to. It is probably some structure of information that is maintained by the RPC runtime system. `clnt_create` allocates one of these structures and returns its pointer to us, and we then pass this pointer to the RPC runtime each time we call a remote procedure.

The first argument to `clnt_create` is either the hostname or IP address of the host running our server. The second argument is the program name, and the third argument is the version number, both from our `square.x` file (Figure 16.1). The final argument is our choice of protocol, and we normally specify either TCP or UDP.

Call remote procedure and print result

12-15 We call our procedure, and the first argument is a pointer to the input structure (`&in`), and the second argument is the client handle. (In most standard I/O calls, the `FILE` handle is the final argument. Similarly, the `CLIENT` handle is normally the final argument to the RPC functions.) The return value is a pointer to the result structure. Notice that we allocate room for the input structure, but the RPC runtime allocates the result structure.

In our `square.x` specification file, we named our procedure `SQUAREPROC`, but from the client we call `squareproc_1`. The convention is that the name in the `.x` file is converted to lowercase and an underscore is appended, followed by the version number.

On the server side, all we write is our server procedure, which we show in Figure 16.3. The `rpcgen` program automatically generates the server `main` function.

sunrpc/square1/server.c

```
1 #include    "unpipc.h"
2 #include    "square.h"

3 square_out *
4 squareproc_1_svc(square_in *inp, struct svc_req *rqstp)
5 {
6     static square_out out;

7     out.res1 = inp->arg1 * inp->arg1;
8     return (&out);
9 }
```

sunrpc/square1/server.c

Figure 16.3 Server procedure that is called using Sun RPC.

Procedure arguments

3-4 We first notice that the name of our server procedure has `_svc` appended following the version number. This allows two ANSI C function prototypes in the `square.h` header, one for the function called by the client in Figure 16.2 (which had the client handle as an argument) and one for the actual server function (which has different arguments).

When our server procedure is called, the first argument is a pointer to the input structure, and the second argument is a pointer to a structure passed by the RPC runtime that contains information about this invocation (which we ignore in this simple procedure).

Execute and return

6-8 We fetch the input argument and calculate its square. The result is stored in a structure whose address is the return value from this function. Since we are returning the address of a variable from the function, that variable *cannot* be an automatic variable. We declare it as `static`.

> Astute readers will note that this prevents our server function from being thread safe. We discuss this in Section 16.2 and show a thread-safe version there.

We now compile our client under Solaris and our server under BSD/OS, start the server, and run the client.

```
solaris % client bsdi 11
result: 121
solaris % client 209.75.135.35 22
result: 484
```

The first time we specify the server's hostname, and the second time its IP address. This demonstrates that the `clnt_create` function and the RPC runtime functions that it calls allow either a hostname or an IP address.

We now demonstrate some error returns from `clnt_create` when either the host does not exist, or the host exists but is not running our server.

```
solaris % client nosuchhost 11
nosuchhost: RPC: Unknown host          from the RPC runtime
clnt_create error                      from our wrapper function
solaris % client localhost 11
localhost: RPC: Program not registered
clnt_create error
```

We have written a client and server and shown their use without any explicit network programming at all. Our client just calls two functions (`clnt_create` and `squareproc_1`), and on the server side, we have just written the function `squareproc_1_svc`. All the details involving XTI under Solaris, sockets under BSD/OS, and network I/O are handled by the RPC runtime. This is the purpose of RPC: to allow the programming of distributed applications without requiring explicit knowledge of network programming.

Another important point in this example is that the two systems, a Sparc running Solaris and an Intel x86 running BSD/OS, have different *byte orders*. That is, the Sparc is big endian and the Intel is little endian (which we show in Section 3.4 of UNPv1). These byte ordering differences are also handled automatically by the runtime library, using a standard called *XDR* (external data representation), which we discuss in Section 16.8.

More steps are involved in building this client and server than in the other programs in this text. Here are the steps involved in building the client executable:

```
solaris % rpcgen -C square.x
solaris % cc -c client.c -o client.o
solaris % cc -c square_clnt.c -o square_clnt.o
solaris % cc -c square_xdr.c -o square_xdr.o
solaris % cc -o client client.o square_clnt.o square_xdr.o libunpipc.a -lnsl
```

The `-C` option to `rpcgen` tells it to generate ANSI C prototypes in the `square.h` header. `rpcgen` also generates a *client stub* (`square_clnt.c`) and a file named `square_xdr.c` that handles the XDR data conversions. Our library (with functions used in this book) is `libunpipc.a`, and `-lnsl` specifies the system library with the networking functions under Solaris (which includes the RPC and XDR runtime).

We see similar commands when we build the server, although `rpcgen` does not need to be run again. The file `square_svc.c` contains the server `main` function, and

`square_xdr.o`, the same file from earlier that contains the XDR functions, is also required by the server.

```
solaris % cc -c server.c -o server.o
solaris % cc -c square_svc.c -o square_svc.o
solaris % cc -o server server.o square_svc.o square_xdr.o libunpipc.a -lnsl
```

This generates a client and server that both run under Solaris.

When the client and server are being built for different systems (e.g., in our earlier example, we ran the client under Solaris and the server under BSD/OS), additional steps may be required. For example, some of the files must be either shared (e.g., NFS) or copied between the two systems, and files that are used by both the client and server (`square_xdr.o`) must be compiled on each system.

Figure 16.4 summarizes the files and steps required to build our client–server example. The three shaded boxes are the files that we must write. The dashed lines show the files that `#include square.h`.

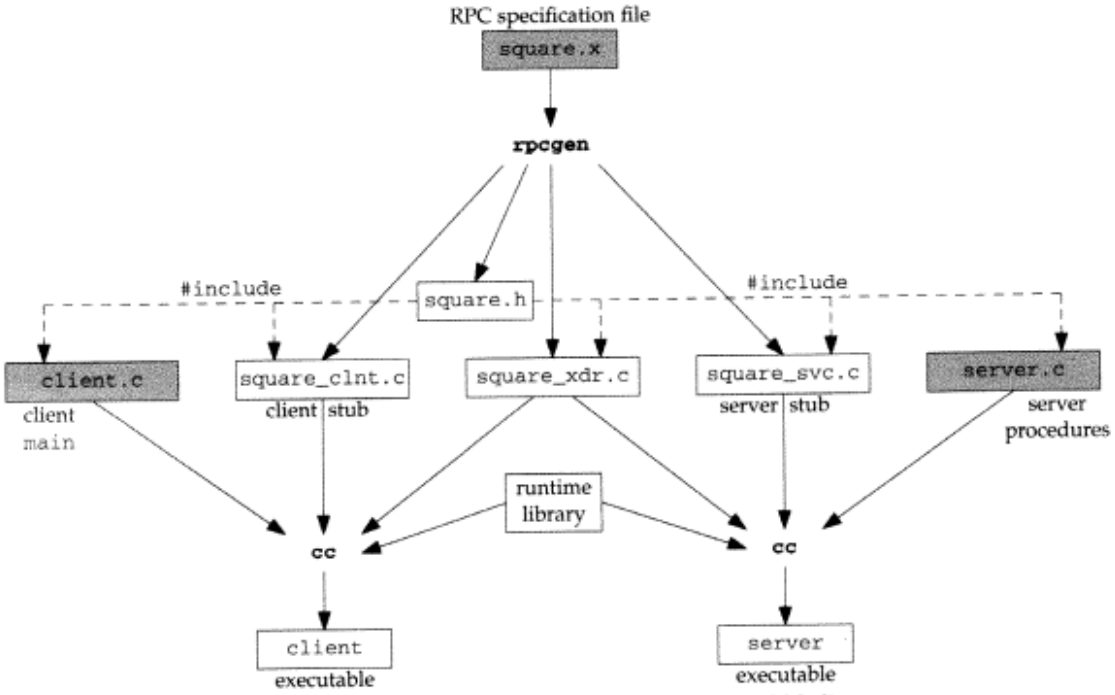

Figure 16.4 Summary of steps required to build an RPC client–server.

Figure 16.5 summarizes the steps that normally take place in a remote procedure call. The numbered steps are executed in order.

0. The sever is started and it registers itself with the port mapper on the server host. The client is then started, and it calls `clnt_create`, which contacts the port mapper on the server host to find the server's ephemeral port. The `clnt_create` function also establishes a TCP connection with the server (since we specified TCP

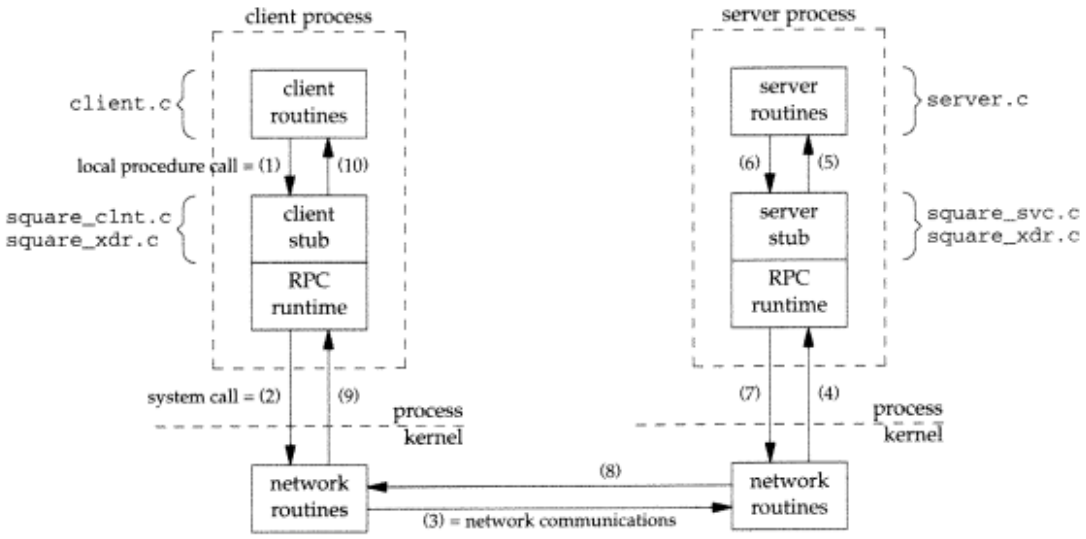

Figure 16.5 Steps involved in a remote procedure call.

as the protocol in Figure 16.2). We do not show these steps in the figure and save our detailed description for Section 16.3.

1. The client calls a local procedure, called the *client stub*. In Figure 16.2, this procedure was named `squareproc_1`, and the file containing the client stub was generated by `rpcgen` and called `square_clnt.c`. To the client, the client stub appears to be the actual server procedure that it wants to call. The purpose of the stub is to package up the arguments to the remote procedure, possibly put them into some standard format, and then build one or more network messages. The packaging of the client's arguments into a network message is termed *marshaling*. The client routines and the stub normally call functions in the RPC runtime library (e.g., `clnt_create` in our earlier example). When link editing under Solaris, these runtime functions are loaded from the `-lnsl` library, whereas under BSD/OS, they are in the standard C library.
2. These network messages are sent to the remote system by the client stub. This normally requires a system call into the local kernel (e.g., `write` or `sendto`).
3. The network messages are transferred to the remote system. The typical networking protocols used for this step are either TCP or UDP.
4. A *server stub* procedure is waiting on the remote system for the client's request. It unmarshals the arguments from the network messages.
5. The server stub executes a local procedure call to invoke the actual server function (our `squareproc_1_svc` procedure in Figure 16.3), passing it the arguments that it received in the network messages from the client.
6. When the server procedure is finished, it returns to the server stub, returning whatever its return values are.

7. The server stub converts the return values, if necessary, and marshals them into one or more network messages to send back to the client.
8. The messages are transferred back across the network to the client.
9. The client stub reads the network messages from the local kernel (e.g., `read` or `recvfrom`).
10. After possibly converting the return values, the client stub finally returns to the client function. This step appears to be a normal procedure return to the client.

History

Probably one of the earliest papers on RPC is [White 1975]. According to [Corbin 1991], White then moved to Xerox, and several RPC systems were developed there. One of these, Courier, was released as a product in 1981. The classic paper on RPC is [Birrell and Nelson 1984], which describes the RPC facility for the Cedar project running on single-user Dorado workstations at Xerox in the early 1980s. Xerox was implementing RPC on workstations before most people knew what workstations were! A Unix implementation of Courier was distributed for many years with the 4.x BSD releases, but today Courier is of historical interest only.

Sun released the first version of its RPC package in 1985. It was developed by Bob Lyon, who had left Xerox in 1983 to join Sun. Its official name is ONC/RPC: Open Network Computing Remote Procedure Call, but it is often called just "Sun RPC." Technically, it is similar to Courier. The original releases of Sun RPC were written using the sockets API and worked with either TCP or UDP. The publicly available source code release was called RPCSRC. In the early 1990s, this was rewritten to use TLI, the predecessor to XTI (described in Part 4 of UNPv1), and works with any networking protocol supported by the kernel. Publicly available source code implementations of both are available from `ftp://playground.sun.com/pub/rpc` with the sockets version named `rpcsrc` and the TLI version named `tirpcsrc` (called TI-RPC, where "TI" stands for "transport independent").

RFC 1831 [Srinivasan 1995a] provides an overview of Sun RPC and describes the format of the RPC messages that are sent across the network. RFC 1832 [Srinivasan 1995b] describes XDR, both the supported datatypes and their format "on the wire." RFC 1833 [Srinivasan 1995c] describes the binding protocols: RPCBIND and its predecessor, the port mapper.

Probably the most widespread application that uses Sun RPC is NFS, Sun's network filesystem. Normally, NFS is not built using the standard RPC tools, `rpcgen` and the RPC runtime library that we describe in this chapter. Instead, most of the library routines are hand-optimized and reside within the kernel for performance reasons. Nevertheless, most systems that support NFS also support Sun RPC.

In the mid-1980s, Apollo competed against Sun in the workstation market, and they designed their own RPC package to compete against Sun's, called NCA (Network Computing Architecture), and their implementation was called NCS (Network Computing System). NCA/RPC was the RPC protocol, NDR (Network Data Representation) was similar to Sun's XDR, and NIDL (Network Interface Definition Language) defined the

interfaces between the clients and servers (e.g., similar to our `.x` file in Figure 16.1). The runtime library was called NCK (Network Computing Kernel).

Apollo was acquired by Hewlett Packard in 1989, and NCA was developed into the Open Software Foundation's Distributed Computing Environment (DCE), of which RPC is a fundamental element from which most pieces are built. More information on DCE is available from `http://www.camb.opengroup.org/tech/dce`. An implementation of the DCE RPC package has been made publicly available at `ftp://gatekeeper.dec.com/pub/DEC/DCE`. This directory also contains a 171-page document describing the internals of the DCE RPC package. DCE is available for many platforms.

> Sun RPC is more widespread than DCE RPC, probably because of its freely available implementation and its packaging as part of the basic system with most versions of Unix. DCE RPC is normally available as an add-on (i.e., separate cost) feature. Widespread porting of the publicly available implementation has not occurred, although a Linux port is underway. In this text, we cover only Sun RPC. All three RPC packages—Courier, Sun RPC, and DCE RPC—are amazingly similar, because the basic RPC concepts are the same.
>
> Most Unix vendors provide additional, detailed documentation on Sun RPC. For example, the Sun documentation is available at `http://docs.sun.com`, and in the Developer Collection, Volume 1, is a 280-page "ONC+ Developer's Guide." The Digital Unix documentation at `http://www.unix.digital.com/faqs/publications/pub_page/V40D_DOCS.HTM` includes a 116-page manual titled "Programming with ONC RPC."
>
> RPC itself is a controversial topic. Eight postings on this topic are contained in `http://www.kohala.com/~rstevens/papers.others/rpc.comments.txt`.

In this chapter, we assume TI-RPC (the transport independent version of RPC mentioned earlier) for most examples, and we talk about TCP and UDP as the supported protocols, even though TI-RPC supports any protocols that are supported by the host.

16.2 Multithreading

Recall Figure 15.9, in which we showed the automatic thread management performed by a doors server, providing a concurrent server by default. We now show that Sun RPC provides an *iterative server* by default. We start with the example from the previous section and modify only the server procedure. Figure 16.6 shows the new function, which prints its thread ID, sleeps for 5 seconds, prints its thread ID again, and returns.

We start the server and then run the client three times:

```
solaris % client localhost 22 & client localhost 33 & \
client localhost 44 &
[3]     25179
[4]     25180
[5]     25181
solaris % result: 484                  about 5 seconds after the prompt is printed
result: 1936                           another 5 seconds later
result: 1089                           and another 5 seconds later
```

sunrpc/square2/server.c

```
 1 #include    "unpipc.h"
 2 #include    "square.h"

 3 square_out *
 4 squareproc_1_svc(square_in *inp, struct svc_req *rqstp)
 5 {
 6     static square_out out;

 7     printf("thread %ld started, arg = %ld\n",
 8            pr_thread_id(NULL), inp->arg1);
 9     sleep(5);
10     out.res1 = inp->arg1 * inp->arg1;
11     printf("thread %ld done\n", pr_thread_id(NULL));

12     return (&out);
13 }
```

sunrpc/square2/server.c

Figure 16.6 Server procedure that sleeps for 5 seconds.

Although we cannot tell from this output, a 5-second wait occurs between the printing of each result by the client. If we look at the server output, we see that the clients are handled iteratively: the first client's request is handled to completion, and then the second client's request is handled to completion, and finally the third client's request is handled to completion.

```
solaris % server
thread 1 started, arg = 22
thread 1 done
thread 1 started, arg = 44
thread 1 done
thread 1 started, arg = 33
thread 1 done
```

One thread handles all client requests: the server is not multithreaded by default.

Our doors servers in Chapter 15 all ran in the foreground when started from the shell, as in

```
solaris % server
```

That allowed us to place debugging calls to `printf` in our server procedures. But Sun RPC servers, by default, run as daemons, performing the steps as outlined in Section 12.4 of UNPv1. This requires calling `syslog` from the server procedure to print any diagnostic information. What we have done, however, is specify the C compiler flag `-DDEBUG` when we compile our server, which is the equivalent of placing the line

```
#define DEBUG
```

in the server stub (the `square_svc.c` file that is generated by `rpcgen`). This stops the server `main` function from making itself a daemon, and leaves it connected to the terminal on which it was started. That is why we can call `printf` from our server procedure.

The provision for a multithreaded server appeared with Solaris 2.4 and is enabled by a `-M` command-line option to `rpcgen`. This makes the server code generated by `rpcgen` thread safe. Another option, `-A`, has the server automatically create threads as

they are needed to process new client requests. We enable both options when we run `rpcgen`.

Both the client and server also require source code changes, which we should expect, given our use of `static` in Figure 16.3. The only change we make to our `square.x` file is to change the version from 1 to 2. Nothing changes in the declarations of the procedure's argument and result structures.

Figure 16.7 shows our new client program.

sunrpc/square3/client.c

```
 1 #include    "unpipc.h"
 2 #include    "square.h"

 3 int
 4 main(int argc, char **argv)
 5 {
 6     CLIENT *cl;
 7     square_in in;
 8     square_out out;

 9     if (argc != 3)
10         err_quit("usage: client <hostname> <integer-value>");

11     cl = Clnt_create(argv[1], SQUARE_PROG, SQUARE_VERS, "tcp");

12     in.arg1 = atol(argv[2]);
13     if (squareproc_2(&in, &out, cl) != RPC_SUCCESS)
14         err_quit("%s", clnt_sperror(cl, argv[1]));

15     printf("result: %ld\n", out.res1);
16     exit(0);
17 }
```

sunrpc/square3/client.c

Figure 16.7 Client `main` function for multithreaded server.

Declare variable to hold result

8 We declare a variable of type `square_out`, not a pointer to this type.

New argument for procedure call

12–14 A pointer to our `out` variable becomes the second argument to `squareproc_2`, and the client handle is the last argument. Instead of this function returning a pointer to the result (as in Figure 16.2), it now returns either `RPC_SUCCESS` or some other value if an error occurs. The `clnt_stat` enum in the `<rpc/clnt_stat.h>` header lists all the possible error returns.

Figure 16.8 shows our new server procedure. As with Figure 16.6, it prints its thread ID, sleeps for 5 seconds, prints another message, and returns.

New arguments and return value

3–12 The changes required for multithreading involve the function arguments and return value. Instead of returning a pointer to the result structure (as in Figure 16.3), a pointer to this structure is now the second argument to the function. The pointer to the `svc_req` structure moves to the third position. The return value is now `TRUE` upon success, or `FALSE` if an error is encountered.

sunrpc/square3/server.c

```
 1 #include    "unpipc.h"
 2 #include    "square.h"

 3 bool_t
 4 squareproc_2_svc(square_in *inp, square_out *outp, struct svc_req *rqstp)
 5 {
 6     printf("thread %ld started, arg = %ld\n",
 7            pr_thread_id(NULL), inp->arg1);
 8     sleep(5);
 9     outp->res1 = inp->arg1 * inp->arg1;
10     printf("thread %ld done\n", pr_thread_id(NULL));
11     return (TRUE);
12 }

13 int
14 square_prog_2_freeresult(SVCXPRT *transp, xdrproc_t xdr_result,
15                           caddr_t result)
16 {
17     xdr_free(xdr_result, result);
18     return (1);
19 }
```

sunrpc/square3/server.c

Figure 16.8 Multithreaded server procedure.

New function to free XDR memory

13-19 Another source code change we must make is to provide a function that frees any storage automatically allocated. This function is called from the server stub after the server procedure returns and after the result has been sent to the client. In our example, we just call the generic `xdr_free` routine. (We talk more about this function with Figure 16.19 and Exercise 16.10.) If our server procedure had allocated any storage necessary to hold the result (say a linked list), it would free that memory from this new function.

We build our client and server and again run three copies of the client at the same time:

```
solaris % client localhost 55 & client localhost 66 & \
client localhost 77 &
[3]     25427
[4]     25428
[5]     25429
solaris % result: 4356
result: 3025
result: 5929
```

This time we can tell that the three results are printed one right after the other. Looking at the server output, we see that three threads are used, and all run simultaneously.

```
solaris % server
thread 1 started, arg = 55
thread 4 started, arg = 77
```

```
thread 6 started, arg = 66
thread 6 done
thread 1 done
thread 4 done
```

One unfortunate side effect of the source code changes required for multithreading is that not all systems support this feature. For example, Digital Unix 4.0B and BSD/OS 3.1 both provide the older RPC system that does not support multithreading. That means if we want to compile and run a program on both types of systems, we need `#ifdefs` to handle the differences in the calling sequences at the client and server ends. Of course, a nonthreaded client on BSD/OS, say, can still call a multithreaded server procedure running on Solaris, but if we have an RPC client (or server) that we want to compile on both types of systems, we need to modify the source code to handle the differences.

16.3 Server Binding

In the description of Figure 16.5, we glossed over step 0: how the server registers itself with its local port mapper and how the client discovers the value of this port. We first note that any host running an RPC server must be running the *port mapper*. The port mapper is assigned TCP port 111 and UDP port 111, and these are the only fixed Internet port assignments for Sun RPC. RPC servers always bind an ephemeral port and then register their ephemeral port with the local port mapper. When a client starts, it must first contact the port mapper on the server's host, ask for the server's ephemeral port number, and then contact the server on that ephemeral port. The port mapper is providing a name service whose scope is confined to that system.

Some readers will claim that NFS also has a fixed port number assignment of 2049. Although many implementations use this port by default, and some older implementations still have this port number hardcoded into the client and server, most current implementations allow other ports to be used. Most NFS clients also contact the port mapper on the server host to obtain the port number.

With Solaris 2.x, Sun renamed the port mapper RPCBIND. The reason for this change is that the term "port" implied Internet ports, whereas the TI-RPC package can work with any networking protocol, not just TCP and UDP. We will use the traditional name of port mapper. Also, in our discussion that follows, we assume that TCP and UDP are the only protocols supported on the server host.

The steps performed by the server and client are as follows:

1. When the system goes into multiuser mode, the port mapper is started. The executable name is typically `portmap` or `rpcbind`.

2. When our server starts, its `main` function (which is part of the server stub that is generated by `rpcgen`) calls the library function `svc_create`. This function determines the networking protocols supported by the host and creates a transport endpoint (e.g., socket) for each protocol, binding an ephemeral port to the TCP and UDP endpoints. It then contacts the local port mapper to register the TCP and UDP ephemeral port numbers with the RPC program number and version number.

The port mapper is itself an RPC program and the server registers itself with the port mapper using RPC calls (albeit to a known port, 111). A description of the procedures supported by the port mapper is given in RFC 1833 [Srinivasan 1995c]. Three versions of this RPC program exist: version 2 is the historical port mapper that handles just TCP and UDP ports, and versions 3 and 4 are the newer RPCBIND protocols.

We can see all the RPC programs that are registered with the port mapper by executing the `rpcinfo` program. We can execute this program to verify that port number 111 is used by the port mapper itself:

```
solaris % rpcinfo -p
   program vers proto   port  service
    100000    4   tcp    111  rpcbind
    100000    3   tcp    111  rpcbind
    100000    2   tcp    111  rpcbind
    100000    4   udp    111  rpcbind
    100000    3   udp    111  rpcbind
    100000    2   udp    111  rpcbind
```

(We have omitted many additional lines of output.) We see that Solaris 2.6 supports all three versions of the protocol, all at port 111, using either TCP or UDP. The mapping from the RPC program number to the service name is normally found in the file `/etc/rpc`. Executing the same command under BSD/OS 3.1 shows that it supports only version 2 of this program.

```
bsdi % rpcinfo -p
   program vers proto   port
    100000    2   tcp    111  portmapper
    100000    2   udp    111  portmapper
```

Digital Unix 4.0B also supports just version 2:

```
alpha % rpcinfo -p
   program vers proto   port
    100000    2   tcp    111  portmapper
    100000    2   udp    111  portmapper
```

Our server process then goes to sleep, waiting for a client request to arrive. This could be a new TCP connection on its TCP port, or the arrival of a UDP datagram on its UDP port. If we execute `rpcinfo` after starting our server from Figure 16.3, we see

```
solaris % rpcinfo -p
   program vers proto   port  service
 824377344    1   udp  47972
 824377344    1   tcp  40849
```

where 824377344 equals `0x31230000`, the program number that we assigned in Figure 16.1. We also assigned a version number of 1 in that figure. Notice that a server is ready to accept clients using either TCP or UDP, and the client chooses which of these two protocols to use when it creates the client handle (the final argument to `clnt_create` in Figure 16.2).

3. The client starts and calls `clnt_create`. The arguments (Figure 16.2) are the server's hostname or IP address, the program number, version number, and a string specifying the protocol. An RPC request is sent to the server host's port mapper (normally using UDP as the protocol for this RPC message), asking for the information on the specified program, version, and protocol. Assuming success, the port number is saved in the client handle for all future RPC calls using this handle.

In Figure 16.1, we assigned a program number of `0x31230000` to our program. The 32-bit program numbers are divided into groups, as shown in Figure 16.9.

Program number	Description
0x00000000 - 0x1fffffff	defined by Sun
0x20000000 - 0x3fffffff	defined by user
0x40000000 - 0x5fffffff	transient (for customer-written applications)
0x60000000 - 0xffffffff	reserved

Figure 16.9 Program number ranges for Sun RPC.

The `rpcinfo` program shows the programs currently registered on your system. Another source of information on the RPC programs supported on a given system is normally the `.x` files in the directory `/usr/include/rpcsvc`.

`inetd` and RPC Servers

By default, servers created by `rpcgen` can be invoked by the `inetd` superserver. (Section 12.5 of UNPv1 covers `inetd` in detail.) Examining the server stub generated by `rpcgen` shows that when the server `main` starts, it checks whether standard input is a XTI endpoint and, if so, assumes it was started by `inetd`.

Backing up, after creating an RPC server that will be invoked by `inetd`, the `/etc/inetd.conf` configuration file needs to be updated with the server information: the RPC program name, the program numbers that are supported, which protocols to support, and the pathname of the server executable. As an example, here is one line (wrapped to fit on this page) from the Solaris configuration file:

```
rstatd/2-4  tli  rpc/datagram_v  wait  root
    /usr/lib/netsvc/rstat/rpc.rstatd rpc.rstatd
```

The first field is the program name (which will be mapped to its corresponding program number using the `/etc/rpc` file), and the supported versions are 2, 3, and 4. The next field specifies a XTI endpoint (as opposed to a socket endpoint), and the third field specifies that all visible datagram protocols are supported. Looking at the file `/etc/netconfig`, there are two of these protocols: UDP and `/dev/clts`. (Chapter 29 of UNPv1 describes this file and XTI addresses.) The fourth field, `wait`, tells `inetd` to wait for this server to terminate before monitoring the XTI endpoint for another client request. All RPC servers in `/etc/inetd.conf` specify the `wait` attribute.

The next field, `root`, specifies the user ID under which the program will run, and the last two fields are the pathname of the executable and the program name with any

arguments to be passed to the program (there are no command-line arguments for this program).

`inetd` will create the XTI endpoints and register them with the port mapper for the specified program and versions. We can verify this with `rpcinfo`:

```
solaris % rpcinfo | grep statd
100001    2    udp        0.0.0.0.128.11       rstatd     superuser
100001    3    udp        0.0.0.0.128.11       rstatd     superuser
100001    4    udp        0.0.0.0.128.11       rstatd     superuser
100001    2    ticlts     \000\000\020,        rstatd     superuser
100001    3    ticlts     \000\000\020,        rstatd     superuser
100001    4    ticlts     \000\000\020,        rstatd     superuser
```

The fourth field is the printable format for XTI addresses (which prints the individual bytes) and $128 \times 256 + 11$ equals 32779, which is the UDP ephemeral port number assigned to this XTI endpoint.

When a UDP datagram arrives for port 32779, `inetd` will detect that a datagram is ready to be read and it will `fork` and then `exec` the program `/usr/lib/netsvc/rstat/rpc.rstatd`. Between the `fork` and `exec`, the XTI endpoint for this server will be duplicated onto descriptors 0, 1, and 2, and all other `inetd` descriptors are closed (Figure 12.7 of UNPv1). `inetd` will also stop monitoring this XTI endpoint for additional client requests until this server (which will be a child of `inetd`) terminates—the `wait` attribute in the configuration file for this server.

Assuming this program was generated by `rpcgen`, it will detect that standard input is a XTI endpoint and initialize it accordingly as an RPC server endpoint. This is done by calling the RPC functions `svc_tli_create` and `svc_reg`, two functions that we do not cover. The second function does not register this server with the port mapper—that is done only once by `inetd` when it starts. The RPC server loop, a function named `svc_run`, will read the pending datagram and call the appropriate server procedure to handle the client's request.

Normally, servers invoked by `inetd` handle one client's request and terminate, allowing `inetd` to wait for the next client request. As an optimization, RPC servers generated by `rpcgen` wait around for a small amount of time (2 minutes is the default) in case another client request arrives. If so, this existing server that is already running will read the datagram and process the request. This avoids the overhead of a `fork` and an `exec` for multiple client requests that arrive in quick succession. After the small wait period, the server will terminate. This will generate `SIGCHLD` for `inetd`, causing it to start looking for arriving datagrams on the XTI endpoint again.

16.4 Authentication

By default, there is no information in an RPC request to identify the client. The server replies to the client's request without worrying about who the client is. This is called *null authentication* or `AUTH_NONE`.

The next level is called *Unix authentication* or `AUTH_SYS`. The client must tell the RPC runtime to include its identification (hostname, effective user ID, effective group ID, and supplementary group IDs) with each request. We modify our client–server

from Section 16.2 to include Unix authentication. Figure 16.10 shows the client.

sunrpc/square4/client.c

```
 1 #include    "unpipc.h"
 2 #include    "square.h"

 3 int
 4 main(int argc, char **argv)
 5 {
 6     CLIENT *cl;
 7     square_in in;
 8     square_out out;

 9     if (argc != 3)
10         err_quit("usage: client <hostname> <integer-value>");

11     cl = Clnt_create(argv[1], SQUARE_PROG, SQUARE_VERS, "tcp");

12     auth_destroy(cl->cl_auth);
13     cl->cl_auth = authsys_create_default();

14     in.arg1 = atol(argv[2]);
15     if (squareproc_2(&in, &out, cl) != RPC_SUCCESS)
16         err_quit("%s", clnt_sperror(cl, argv[1]));

17     printf("result: %ld\n", out.res1);
18     exit(0);
19 }
```

sunrpc/square4/client.c

Figure 16.10 Client that provides Unix authentication.

12-13 These two lines are new. We first call `auth_destroy` to destroy the previous authentication associated with this client handle, the null authentication that is created by default. The function `authsys_create_default` creates the appropriate Unix authentication structure, and we store that in the `cl_auth` member of the `CLIENT` structure. The remainder of the client has not changed from Figure 16.7.

Figure 16.11 shows our new server procedure, modified from Figure 16.8. We do not show the `square_prog_2_freeresult` function, which does not change.

6-8 We now use the pointer to the `svc_req` structure that is always passed as an argument to the server procedure.

```
struct svc_req {
  u_long              rq_prog;     /* program number */
  u_long              rq_vers;     /* version number */
  u_long              rq_proc;     /* procedure number */
  struct opaque_auth  rq_cred;     /* raw credentials */
  caddr_t             rq_clntcred; /* cooked credentials (read-only) */
  SVCXPRT            *rq_xprt;     /* transport handle */
};

struct opaque_auth {
  enum_t   oa_flavor;    /* flavor: AUTH_xxx constant */
  caddr_t  oa_base;      /* address of more auth stuff */
  u_int    oa_length;    /* not to exceed MAX_AUTH_BYTES */
};
```

sunrpc/square4/server.c

```
 1 #include    "unpipc.h"
 2 #include    "square.h"

 3 bool_t
 4 squareproc_2_svc(square_in *inp, square_out *outp, struct svc_req *rqstp)
 5 {
 6     printf("thread %ld started, arg = %ld, auth = %d\n",
 7            pr_thread_id(NULL), inp->arg1, rqstp->rq_cred.oa_flavor);
 8     if (rqstp->rq_cred.oa_flavor == AUTH_SYS) {
 9         struct authsys_parms *au;

10         au = (struct authsys_parms *) rqstp->rq_clntcred;
11         printf("AUTH_SYS: host %s, uid %ld, gid %ld\n",
12                au->aup_machname, (long) au->aup_uid, (long) au->aup_gid);
13     }
14     sleep(5);
15     outp->res1 = inp->arg1 * inp->arg1;
16     printf("thread %ld done\n", pr_thread_id(NULL));
17     return (TRUE);
18 }
```

sunrpc/square4/server.c

Figure 16.11 Server procedure that looks for Unix authentication.

The `rq_cred` member contains the raw authentication information, and its `oa_flavor` member is an integer that identifies the type of authentication. The term "raw" means that the RPC runtime has not processed the information pointed to by `oa_base`. But if the authentication type is one supported by the runtime, then the cooked credentials pointed to by `rq_clntcred` have been processed by the runtime into some structure appropriate for that type of authentication. We print the type of authentication and then check whether it equals `AUTH_SYS`.

9–12 For Unix authentication, the pointer to the cooked credentials (`rq_clntcred`) points to an `authsys_parms` structure containing the client's identity:

```
struct authsys_parms {
  u_long  aup_time;      /* credentials creation time */
  char   *aup_machname;  /* hostname where client is located */
  uid_t   aup_uid;       /* effective user ID */
  gid_t   aup_gid;       /* effective group ID */
  u_int   aup_len;       /* #elements in aup_gids[] */
  gid_t  *aup_gids;      /* supplementary group IDs */
};
```

We obtain the pointer to this structure and print the client's hostname, effective user ID, and effective group ID.

If we start our server and run the client once, we get the following output from the server:

```
solaris % server
thread 1 started, arg = 44, auth = 1
AUTH_SYS: host solaris.kohala.com, uid 765, gid 870
thread 1 done
```

Unix authentication is rarely used, because it is simple to defeat. We can easily build our own RPC packets containing Unix authentication information, setting the user ID and group IDs to any values we want, and send it to the server. The server has no way to *verify* that we are who we claim to be.

> Actually, NFS uses Unix authentication by default, but the requests are normally sent by the NFS client's kernel and usually with a reserved port (Section 2.7 of UNPv1). Some NFS servers are configured to respond to a client's request only if it arrives from a reserved port. If you are trusting the client host to mount your filesystems, you are trusting that client's kernel to identify its users correctly. If a reserved port is not required by the server, then hackers can write their own programs that send NFS requests to an NFS server, setting the Unix authentication IDs to any values desired. Even if a reserved port is required by the server, if you have your own system on which you have superuser privileges, and you can plug your system into the network, you can still send your own NFS requests to the server.

An RPC packet, either a request or a reply, actually contains two fields related to authentication: the *credentials* and the *verifier* (Figures 16.30 and 16.32). A common analogy is a picture ID (passport, driver's license, or whatever). The credentials are the printed information (name, address, date of birth, etc.), and the verifier is the picture. There are also different forms of credentials: a picture is better than just listing the height, weight, and sex, for example. If we had an ID card without any form of identifying information (library cards are often examples of this), then we would have credentials without any verifier, and anyone could use the card and claim to be the owner.

In the case of null authentication, both the credentials and the verifier are empty. With Unix authentication, the credentials contain the hostname and the user and group IDs, but the verifier is empty. Other forms of authentication are supported, and the credentials and verifiers contain other information:

`AUTH_SHORT` An alternate form of Unix authentication that is sent in the verifier field from the server back to the client in the RPC reply. It is a smaller amount of information than full Unix authentication, and the client can send this back to the server as the credentials in subsequent requests. The intent of this type of credential is to save network bandwidth and server CPU cycles.

`AUTH_DES` DES is an acronym for the *Data Encryption Standard*, and this form of authentication is based on secret key and public key cryptography. This scheme is also called *secure RPC*, and when used as the basis for NFS, this is called *secure NFS*.

`AUTH_KERB` This scheme is based on MIT's Kerberos system for authentication.

Chapter 19 of [Garfinkel and Spafford 1996] says more about the latter two forms of authentication, including their setup and use.

16.5 Timeout and Retransmission

We now look at the timeout and retransmission strategy used by Sun RPC. Two timeout values are used:

1. The *total timeout* is the total amount of time that a client waits for the server's reply. This value is used by both TCP and UDP.
2. The *retry timeout* is used only by UDP and is the amount of time between retransmissions of the client's request, waiting for the server's reply.

First, no need exists for a retry timeout with TCP because TCP is a reliable protocol. If the server host never receives the client's request, the client's TCP will time out and retransmit the request. When the server host receives the client's request, the server's TCP will acknowledge its receipt to the client's TCP. If the server's acknowledgment is lost, causing the client's TCP to retransmit the request, when the server TCP receives this duplicate data, it will be discarded and another acknowledgment sent by the server TCP. With a reliable protocol, the reliability (timeout, retransmission, handling of duplicate data or duplicate ACKs) is provided by the transport layer, and is not a concern of the RPC runtime. One request sent by the client RPC layer will be received as one request by the server RPC layer (or the client RPC layer will get an error indication if the request never gets acknowledged), regardless of what happens at the network and transport layers.

After we have created a client handle, we can call `clnt_control` to both query and set options that affect the handle. This is similar to calling `fcntl` for a descriptor, or calling `getsockopt` and `setsockopt` for a socket.

```
#include <rpc/rpc.h>

bool_t clnt_control(CLIENT *cl, unsigned int request, char *ptr);
```

Returns: TRUE if OK, FALSE on error

cl is the client handle, and what is pointed to by *ptr* depends on the *request*.

We modify our client from Figure 16.2 to call this function and print the two timeouts. Figure 16.12 shows our new client.

Protocol is a command-line argument

10-12 We now specify the protocol as another command-line argument and use this as the final argument to `clnt_create`.

Get total timeout

13-14 The first argument to `clnt_control` is the client handle, the second is the request, and the third is normally a pointer to a buffer. Our first request is `CLGET_TIMEOUT`, which returns the total timeout in the `timeval` structure whose address is the third argument. This request is valid for all protocols.

Try to get retry timeout

15-16 Our next request is `CLGET_RETRY_TIMEOUT` for the retry timeout, but this is valid only for UDP. Therefore, if the return value is `FALSE`, we print nothing.

We also modify our server from Figure 16.6 to sleep for 1000 seconds instead of 5 seconds, to guarantee that the client's request times out. We start the server on our host

sunrpc/square5/client.c

```
 1 #include    "unpipc.h"
 2 #include    "square.h"

 3 int
 4 main(int argc, char **argv)
 5 {
 6     CLIENT *cl;
 7     square_in in;
 8     square_out *outp;
 9     struct timeval tv;

10     if (argc != 4)
11         err_quit("usage: client <hostname> <integer-value> <protocol>");

12     cl = Clnt_create(argv[1], SQUARE_PROG, SQUARE_VERS, argv[3]);

13     Clnt_control(cl, CLGET_TIMEOUT, (char *) &tv);
14     printf("timeout = %ld sec, %ld usec\n", tv.tv_sec, tv.tv_usec);
15     if (clnt_control(cl, CLGET_RETRY_TIMEOUT, (char *) &tv) == TRUE)
16         printf("retry timeout = %ld sec, %ld usec\n", tv.tv_sec, tv.tv_usec);

17     in.arg1 = atol(argv[2]);
18     if ( (outp = squareproc_1(&in, cl)) == NULL)
19         err_quit("%s", clnt_sperror(cl, argv[1]));

20     printf("result: %ld\n", outp->res1);
21     exit(0);
22 }
```

sunrpc/square5/client.c

Figure 16.12 Client that queries and prints the two RPC timeout values.

`bsdi` and run the client twice, once specifying TCP and once specifying UDP, but the results are not what we expect:

```
solaris % date ; client bsdi 44 tcp ; date
Wed Apr 22 14:46:57 MST 1998
timeout = 30 sec, 0 usec              this says 30 seconds
bsdi: RPC: Timed out
Wed Apr 22 14:47:22 MST 1998          but this is 25 seconds later

solaris % date ; client bsdi 55 udp ; date
Wed Apr 22 14:48:05 MST 1998
timeout = -1 sec, -1 usec             bizarre
retry timeout = 15 sec, 0 usec        this turns out to be correct
bsdi: RPC: Timed out
Wed Apr 22 14:48:31 MST 1998          about 25 seconds later
```

In the TCP case, the total timeout is returned by `clnt_control` as 30 seconds, but our measurement shows a timeout of 25 seconds. With UDP, the total timeout is returned as –1.

To see what is happening here, look at the client stub, the function `squareproc_1` in the file `square_clnt.c` that is generated by `rpcgen`. This function calls a library function named `clnt_call`, and the final argument is a `timeval` structure named

TIMEOUT that is declared in this file and is initialized to 25 seconds. This argument to clnt_call overrides the default of 30 seconds for TCP and the –1 values for UDP. This argument is used until the client explicitly sets the total timeout by calling clnt_control with a request of CLSET_TIMEOUT. If we want to change the total timeout, we should call clnt_control and should not modify the structure in the client stub.

> The only way to verify the UDP retry timeout is to watch the packets using tcpdump. This shows that the first datagram is sent as soon as the client starts, and the next datagram is about 15 seconds later.

TCP Connection Management

If we watch the TCP client–server that we just described using tcpdump, we see TCP's three-way handshake, followed by the client sending its request, and the server acknowledging this request. About 25 seconds later, the client sends a FIN, which is caused by the client process terminating, and the remaining three segments of the TCP connection termination sequence follow. Section 2.5 of UNPv1 describes these segments in more detail.

We want to show the following characteristics of Sun RPC's usage of TCP connections: a new TCP connection is established by the client's call to clnt_create, and this connection is used by all procedure calls associated with the specified program and version. A client's TCP connection is terminated either explicitly by calling clnt_destroy or implicitly by the termination of the client process.

```
#include <rpc/rpc.h>

void clnt_destroy(CLIENT *cl);
```

We start with our client from Figure 16.2 and modify it to call the server procedure twice, then call clnt_destroy, and then pause. Figure 16.13 shows the new client.

Running this program yields the expected output:

```
solaris % client kalae 5
result: 25
result: 100
                                    program just waits until we kill it
```

But the verification of our earlier statements is shown only by the tcpdump output. This shows that one TCP connection is created (by the call to clnt_create) and is used for both client requests. The connection is then terminated by the call to clnt_destroy, even though our client process does not terminate.

Transaction ID

Another part of the timeout and retransmission strategy is the use of a *transaction ID* or *XID* to identify the client requests and server replies. When a client issues an RPC call, the RPC runtime assigns a 32-bit integer XID to the call, and this value is sent in the

sunrpc/square9/client.c

```
1 #include    "unpipc.h"          /* our header */
2 #include    "square.h"          /* generated by rpcgen */

3 int
4 main(int argc, char **argv)
5 {
6     CLIENT *cl;
7     square_in in;
8     square_out *outp;

9     if (argc != 3)
10        err_quit("usage: client <hostname> <integer-value>");

11    cl = Clnt_create(argv[1], SQUARE_PROG, SQUARE_VERS, "tcp");

12    in.arg1 = atol(argv[2]);
13    if ( (outp = squareproc_1(&in, cl)) == NULL)
14        err_quit("%s", clnt_sperror(cl, argv[1]));
15    printf("result: %ld\n", outp->res1);

16    in.arg1 *= 2;
17    if ( (outp = squareproc_1(&in, cl)) == NULL)
18        err_quit("%s", clnt_sperror(cl, argv[1]));
19    printf("result: %ld\n", outp->res1);

20    clnt_destroy(cl);

21    pause();

22    exit(0);
23 }
```

sunrpc/square9/client.c

Figure 16.13 Client program to examine TCP connection usage.

RPC message. The server must return this XID with its reply. The XID does not change when the RPC runtime retransmits a request. The XID serves two purposes:

1. The client verifies that the XID of the reply equals the XID that was sent with the request; otherwise the client ignores this reply. If TCP is being used, the client should rarely receive a reply with the incorrect XID, but with UDP, and the possibility of retransmitted requests and a lossy network, the receipt of a reply with the incorrect XID is a definite possibility.
2. The server is allowed to maintain a *cache* of the replies that it sends, and one of the items that it uses to determine whether a request is a duplicate is the XID. We describe this shortly.

The TI-RPC package uses the following algorithm for choosing an XID for a new request, where the ^ operator is C's bitwise exclusive OR:

```
struct timeval  now;

gettimeofday(&now, NULL);
xid = getpid() ^ now.tv_sec ^ now.tv_usec;
```

Server Duplicate Request Cache

To enable the RPC runtime to maintain a duplicate request cache, the server must call `svc_dg_enablecache`. Once this cache is enabled, there is no way to turn it off (other than termination of the server process).

```
#include <rpc/rpc.h>

int svc_dg_enablecache(SVCXPRT *xprt, unsigned long size);
```

Returns: 1 if OK, 0 on error

xprt is the transport handle, and this pointer is member of the `svc_req` structure (Section 16.4). The address of this structure is an argument to the server procedure. *size* is the number of cache entries for which memory should be allocated.

When this cache is enabled, the server maintains a FIFO (first-in, first-out) cache of all the replies that it sends. Each reply is uniquely identified by the following:

- program number,
- version number,
- procedure number,
- XID, and
- client address (IP address and UDP port).

Each time the RPC runtime in the server receives a client request, it first searches the cache to see whether it already has a reply for this request. If so, the cached reply is returned to the client instead of calling the server procedure again.

The purpose of the duplicate request cache is to avoid calling a server procedure multiple times when duplicate requests are received, probably because the server procedure is not *idempotent*. A duplicate request can be received because the reply was lost or because the client retransmission passes the reply in the network. Notice that this duplicate request cache applies only to datagram protocols such as UDP, because if TCP is being used, a duplicate request never makes it to the application; it is handled completely by TCP (see Exercise 16.6).

16.6 Call Semantics

In Figure 15.29, we showed a doors client that retransmitted its request to the server when the client's call to `door_call` was interrupted by a caught signal. But we then showed that this caused the server procedure to be called twice, not once. We then categorized server procedures into those that are *idempotent* (can be called any number of times without harm), and those that are not idempotent (such as subtracting money from a bank account).

Procedure calls can be placed into one of three categories:

1. *Exactly once* means the procedure was executed once, period. This type of operation is hard to achieve, owing to the possibility of server crashes.

2. *At most once* means the procedure was not executed at all or it was executed once. If a normal return is made to the caller, we know the procedure was executed once. But if an error return is made, we're not certain whether the procedure was executed once or not at all.
3. *At least once* means the procedure was executed at least once, but perhaps more. This is OK for idempotent procedures—the client keeps transmitting its request until it receives a valid response. But if the client has to send its request more than once to receive a valid response, a possibility exists that the procedure was executed more than once.

With a local procedure call, if the procedure returns, we know that it was executed exactly once, but if the process crashes after the procedure has been called, we don't know whether it was executed once or not at all. We must consider various scenarios with remote procedure calls:

- If TCP is being used and a reply is received, we know that the remote procedure was called exactly once. But if a reply is not received (say the server crashes), we don't know whether the server procedure executed to completion before the host crashed, or whether the server procedure had not yet been called (at-most-once semantics). Providing exactly-once semantics in the face of server crashes and extended network outages requires a transaction processing system, something that is beyond the capability of an RPC package.
- If UDP is being used without a server cache and a reply is received, we know that the server procedure was called at least once, but possibly more than once (at-least-once semantics).
- If UDP is being used with a server cache and a reply is received, we know that the server procedure was called exactly once. But if a reply is not received, we have at-most-once semantics, similar to the TCP scenario.

Given these three choices:

1. TCP,
2. UDP with a server cache, or
3. UDP without a server cache,

our recommendations are:

- Always use TCP unless the overhead of the TCP connections is excessive for the application.
- Use a transaction processing system for nonidempotent procedures that are important to do correctly (i.e., bank accounts, airline reservations, and the like).
- For a nonidempotent procedure, using TCP is preferable to UDP with a server cache. TCP was designed to be reliable from the beginning, and adding this to a UDP application is rarely the same as just using TCP (e.g., Section 20.5 of UNPv1).

- Using UDP without a server cache for an idempotent procedure is OK.
- Using UDP without a server cache for a nonidempotent procedure is dangerous.

We cover additional advantages of TCP in the next section.

16.7 Premature Termination of Client or Server

We now consider what happens when either the client or the server terminates prematurely and TCP is being used as the transport protocol. Since UDP is connectionless, when a process with an open UDP endpoint terminates, nothing is sent to the peer. All that will happen in the UDP scenario when one end crashes is that the peer will time out, possibly retransmit, and eventually give up, as discussed in the previous section. But when a process with an open TCP connection terminates, that connection is terminated, sending a FIN to the peer (pp. 36–37 of UNPv1), and we want to see what the RPC runtime does when it receives this unexpected FIN from its peer.

Premature Termination of Server

We first terminate the server prematurely while it is processing a client's request. The only change we make to our client is to remove the `"tcp"` argument from the call to `clnt_call` in Figure 16.2 and require the transport protocol to be a command-line argument, as in Figure 16.12. In our server procedure, we add a call to the `abort` function. This terminates the server process, causing the server's TCP to send a FIN to the client, which we can verify with `tcpdump`.

We first run our Solaris client to our BSD/OS server:

```
solaris % client bsdi 22 tcp
bsdi: RPC: Unable to receive; An event requires attention
```

When the server's FIN is received by the client, the RPC runtime is waiting for the server's reply. It detects the unexpected reply and returns an error from our call to `squareproc_1`. The error (`RPC_CANTRECV`) is saved by the runtime in the client handle, and the call to `clnt_sperror` (from our `Clnt_create` wrapper function) prints this as "Unable to receive." The remainder of the error message, "An event requires attention," corresponds to the XTI error saved by the runtime, and is also printed by `clnt_sperror`. About 30 different `RPC_`*xxx* errors can be returned by a client's call of a remote procedure, and they are listed in the `<rpc/clnt_stat.h>` header.

If we swap the hosts for the client and server, we see the same scenario, with the same error returned by the RPC runtime (`RPC_CANTRECV`), but a different message at the end.

```
bsdi % client solaris 11 tcp
solaris: RPC: Unable to receive; errno = Connection reset by peer
```

The Solaris server that we aborted above was not compiled as a multithreaded server, and when we called `abort`, the entire process was terminated. Things change if we are running a multithreaded server and only the thread servicing the client's call

terminates. To force this scenario, we replace the call to `abort` with a call to `pthread_exit`, as we did with our doors example in Figure 15.25. We run our client under BSD/OS and our multithreaded server under Solaris.

```
bsdi % client solaris 33 tcp
solaris: RPC: Timed out
```

When the server thread terminates, the TCP connection to the client is *not* closed; it remains open in the server process. Therefore, no FIN is sent to the client, so the client just times out. We would see the same error if the server host crashed after the client's request was sent to the server and acknowledged by the server's TCP.

Premature Termination of Client

When an RPC client terminates while it has an RPC procedure call in progress using TCP, the client's TCP will send a FIN to the server when the client process terminates. Our question is whether the server's RPC runtime detects this condition and possibly notifies the server procedure. (Recall from Section 15.11 that a doors server thread is canceled when the client prematurely terminates.)

To generate this condition, our client calls `alarm(3)` right before calling the server procedure, and our server procedure calls `sleep(6)`. (This is what we did with our doors example in Figures 15.30 and 15.31. Since the client does not catch `SIGALRM`, the process is terminated by the kernel about 3 seconds before the server's reply is sent.) We run our client under BSD/OS and our server under Solaris.

```
bsdi % client solaris 44 tcp
Alarm call
```

This is what we expect at the client, but nothing different happens at the server. The server procedure completes its 6-second sleep and returns. If we watch what happens with `tcpdump` we see the following:

- When the client terminates (about 3 seconds after starting), the client TCP sends a FIN to the server, which the server TCP acknowledges. The TCP term for this is a *half-close* (Section 18.5 of TCPv1).
- About 6 seconds after the client and server started, the server sends its reply, which its TCP sends to the client. (Sending data across a TCP connection after receiving a FIN is OK, as we describe on pp. 130–132 of UNPv1, because TCP connections are full-duplex.) The client TCP responds with an RST (reset), because the client process has terminated. This will be recognized by the server on its next read or write on the connection, but nothing happens at this time.

We summarize the points made in this section.

- RPC clients and servers using UDP never know whether the other end terminates prematurely. They may time out when no response is received, but they cannot tell the type of error: premature process termination, crashing of the peer host, network unreachability, and so on.

- An RPC client or server using TCP has a better chance of detecting problems at the peer, because premature termination of the peer process automatically causes the peer TCP to close its end of the connection. But this does not help if the peer is a threaded RPC server, because termination of the peer thread does not close the connection. Also this does not help detect a crashing of the peer host, because when that happens, the peer TCP does not close its open connections. A timeout is still required to handle all these scenarios.

16.8 XDR: External Data Representation

When we used doors in the previous chapter to call a procedure in one process from another process, both processes were on the same host, so we had no data conversion problems. But with RPC between different hosts, the various hosts can use different data formats. First, the sizes of the fundamental C datatypes can be different (e.g., a `long` on some systems occupies 32 bits, whereas on others it occupies 64 bits), and second, the actual bit ordering can differ (e.g., big-endian versus little-endian byte ordering, which we talked about on pp. 66–69 and pp. 137–140 of UNPv1). We have already encountered this with Figure 16.3 when we ran our server on a little-endian x86 and our client on a big-endian Sparc, yet we were able to exchange a long integer correctly between the two hosts.

Sun RPC uses XDR, the *External Data Representation* standard, to describe and encode the data (RFC 1832 [Srinivasan 1995b]). XDR is both a language for *describing* the data and a set of rules for *encoding* the data. XDR uses *implicit typing*, which means the sender and receiver must both know the types and ordering of the data: for example, two 32-bit integer values followed by one single precision floating point value, followed by a character string.

> As a comparison, in the OSI world, ASN.1 (Abstract Syntax Notation one) is the normal way to describe the data, and BER (Basic Encoding Rules) is a common way to encode the data. This scheme also uses *explicit typing*, which means each data value is preceded by some value (a "specifier") describing the datatype that follows. In our example, the stream of bytes would contain the following fields, in order: a specifier that the next value is an integer, the integer value, a specifier that the next value is an integer, the integer value, a specifier that the next value is a floating point value, the floating point value, a specifier that the next value is a character string, the character string.

The XDR representation of all datatypes requires a multiple of 4 bytes, and these bytes are always transmitted in the big-endian byte order. Signed integer values are stored using two's complement notation, and floating point values are stored using the IEEE format. Variable-length fields always contain up to 3 bytes of padding at the end, so that the next item is on a 4-byte boundary. For example, a 5-character ASCII string would be transmitted as 12 bytes:

- a 4-byte integer count containing the value 5,
- the 5-byte string, and
- 3 bytes of 0 for padding.

When describing XDR and the datatypes that it supports, we have three items to consider:

1. How do we declare a variable of each type in our RPC specification file (our `.x` file) for `rpcgen`? Our only example so far (Figure 16.1) uses only a long integer.
2. Which C datatype does `rpcgen` convert this to in the `.h` header that it generates?
3. What is the actual format of the data that is transmitted?

Figure 16.14 answers the first two questions. To generate this table, an RPC specification file was created using all the supported XDR datatypes. The file was run through `rpcgen` and the resulting C header examined.

We now describe the table entries in more detail, referencing each by the number in the first column (1–15).

1. A `const` declaration is turned into a C `#define`.
2. A `typedef` declaration is turned into a C `typedef`.
3. These are the five signed integer datatypes. The first four are transmitted as 32-bit values by XDR, and the last one is transmitted as a 64-bit value by XDR.

   64-bit integers are known to many C compilers as type `long long int` or just `long long`. Not all compilers and operating systems support these. Since the generated `.h` file declares the C variable of type `longlong_t`, some header needs to define

   ```
   typedef  long long  longlong_t;
   ```

   An XDR `long` occupies 32 bits, but a C `long` on a 64-bit Unix system holds 64 bits (e.g., the LP64 model described on p. 27 of UNPv1). Indeed, these decade-old XDR names are unfortunate in today's world. Better names would have been something like `int8_t`, `int16_t`, `int32_t`, `int64_t`, and so on.

4. These are the five unsigned integer datatypes. The first four are transmitted as 32-bit values by XDR, and the last one is transmitted as a 64-bit value by XDR.
5. These are the three floating point datatypes. The first is transmitted as a 32-bit value, the second as a 64-bit value, and the third as a 128-bit value.

   Quadruple-precision floating point numbers are known in C as type `long double`. Not all compilers and operating systems support these. (Your compiler may allow `long double`, but treat it as a `double`.) Since the generated `.h` file declares the C variable of type `quadruple`, some header needs to define

   ```
   typedef  long double  quadruple;
   ```

   Under Solaris 2.6, for example, we must include the line

   ```
   %#include <floatingpoint.h>
   ```

   at the beginning of the `.x` file, because this header includes the required definition. The percent sign at the beginning of the line tells `rpcgen` to place the remainder of the line in the `.h` file.

	RPC specification file (.x)	C header file (.h)
1	`const` *name* `=` *value*`;`	`#define` *name value*
2	`typedef` *declaration*`;`	`typedef` *declaration*`;`

3	`char` *var*`;`
	`short` *var*`;`
	`int` *var*`;`
	`long` *var*`;`
	`hyper` *var*`;`
		`char` *var*`;`
		`short` *var*`;`
		`int` *var*`;`
		`long` *var*`;`
		`longlong_t` *var*`;`

4	`unsigned char` *var*`;`
	`unsigned short` *var*`;`
	`unsigned int` *var*`;`
	`unsigned long` *var*`;`
	`unsigned hyper` *var*`;`
		`u_char` *var*`;`
		`u_short` *var*`;`
		`u_int` *var*`;`
		`u_long` *var*`;`
		`u_longlong_t` *var*`;`

5	`float` *var*`;`
	`double` *var*`;`
	`quadruple` *var*`;`
		`float` *var*`;`
		`double` *var*`;`
		`quadruple` *var*`;`

6	`bool` *var*`;`	`bool_t` *var*`;`
7	`enum` *var* `{` *name* `=` *const*`, ... };`	`enum` *var* `{` *name* `=` *const*`, ... };` `typedef enum` *var var*`;`
8	`opaque` *var*`[`*n*`];`	`char` *var*`[`*n*`];`

9	`opaque` *var*`<`*m*`>;`	`struct {`
		`  u_int  var_len;`
		`  char  *var_val;`
		`} var;`

10	`string` *var*`<`*m*`>;`	`char *var;`
11	*datatype var*`[`*n*`];`	*datatype var*`[`*n*`];`

12	*datatype var*`<`*m*`>;`	`struct {`
		`  u_int  var_len;`
		`  datatype  *var_val;`
		`} var;`

13	`struct` *var* `{` *members* `... };`	`struct` *var* `{` *members* `... };` `typedef struct` *var var*`;`

14	`union` *var* `switch (int` *disc*`) {`
	`  case` *discvalueA*`:  armdeclA;`
	`  case` *discvalueB*`:  armdeclB;`
	`  ...`
	`  default:  defaultdecl;`
	`};`
		`struct` *var* `{`
		`  int disc;`
		`  union {`
		`  armdeclA;`
		`  armdeclB;`
		`  ...`
		`  defaultdecl;`
		`  } var_u;`
		`};`
		`typedef struct` *var var*`;`

15	*datatype* `*`*name*`;`	*datatype* `*`*name*`;`

Figure 16.14 Summary of datatypes supported by XDR and `rpcgen`.

6. The *boolean* datatype is equivalent to a signed integer. The RPC headers also `#define` the constant `TRUE` to be 1 and the constant `FALSE` to be 0.
7. An *enumeration* is equivalent to a signed integer and is the same as C's `enum` datatype. `rpcgen` also generates a `typedef` for the specified variable name.
8. *Fixed-length opaque data* is a specified number of bytes (n) that are transmitted as 8-bit values, uninterpreted by the runtime library.
9. *Variable-length opaque data* is also a sequence of uninterpreted bytes that are transmitted as 8-bit values, but the actual number of bytes is transmitted as an unsigned integer and precedes the data. When sending this type of data (e.g., when filling in the arguments prior to an RPC call), set the length before making the call. When this type of data is received, the length must be examined to determine how much data follows.

   The maximum length m can be omitted in the declaration. But if the length is specified at compile time, the runtime library will check that the actual length (what we show as the *var*`_len` member of the structure) does not exceed the value of m.
10. A *string* is a sequence of ASCII characters. In memory, a string is stored as a normal null-terminated C character string, but when a string is transmitted, it is preceded by an unsigned integer that specifies the actual number of characters that follows (not including the terminating null). When sending this type of data, the runtime determines the number of characters by calling `strlen`. When this type of data is received, it is stored as a null-terminated C character string.

    The maximum length m can be omitted in the declaration. But if the length is specified at compile time, the runtime library will check that the actual length does not exceed the value of m.
11. A *fixed-length array* of any datatype is transmitted as a sequence of n elements of that datatype.
12. A *variable-length array* of any datatype is transmitted as an unsigned integer that specifies the actual number of elements in the array, followed by the array elements.

    The maximum number of elements m can be omitted in the declaration. But if this maximum is specified at compile time, the runtime library will check that the actual length does not exceed the value of m.
13. A *structure* is transmitted by transmitting each member in turn. `rpcgen` also generates a `typedef` for the specified variable name.
14. A *discriminated union* is composed of an integer discriminant followed by a set of datatypes (called *arms*) based on the value of the discriminant. In Figure 16.14, we show that the discriminant must be an `int`, but it can also be an `unsigned int`, an `enum`, or a `bool` (all of which are transmitted as a 32-bit integer value). When a discriminated union is transmitted, the 32-bit value of

the discriminant is transmitted first, followed only by the arm value corresponding to the value of the discriminant. The `default` declaration is often `void`, which means that nothing is transmitted following the 32-bit value of the discriminant. We show an example of this shortly.

15. *Optional data* is a special type of `union` that we describe with an example in Figure 16.24. The XDR declaration looks like a C pointer declaration, and that is what the generated `.h` file contains.

Figure 16.16 summarizes the encoding used by XDR for its various datatypes.

Example: Using XDR without RPC

We now show an example of XDR but without RPC. That is, we will use XDR to encode a structure of binary data into a machine-independent representation that can be processed on other systems. This technique can be used to write files in a machine-independent format or to send data to another computer across a network in a machine-independent format. Figure 16.15 shows our RPC specification file, `data.x`, which is really just an XDR specification file, since we do not declare any RPC procedures.

> The filename suffix of `.x` comes from the term "XDR specification file." The RPC specification (RFC 1831) says that the RPC language, sometimes called RPCL, is identical to the XDR language (which is defined in RFC 1832), except for the addition of a program definition (which describes the program, versions, and procedures).

sunrpc/xdr1/data.x

```
 1 enum result_t {
 2     RESULT_INT = 1, RESULT_DOUBLE = 2
 3 };

 4 union union_arg switch (result_t result) {
 5 case RESULT_INT:
 6     int     intval;
 7 case RESULT_DOUBLE:
 8     double  doubleval;
 9 default:
10     void;
11 };

12 struct data {
13     short   short_arg;
14     long    long_arg;

15     string  vstring_arg < 128 >;    /* variable-length string */
16     opaque  fopaque_arg[3];     /* fixed-length opaque */
17     opaque  vopaque_arg <>;     /* variable-length opaque */
18     short   fshort_arg[4];      /* fixed-length array */
19     long    vlong_arg <>;       /* variable-length array */
20     union_arg uarg;
21 };
```

sunrpc/xdr1/data.x

Figure 16.15 XDR specification file.

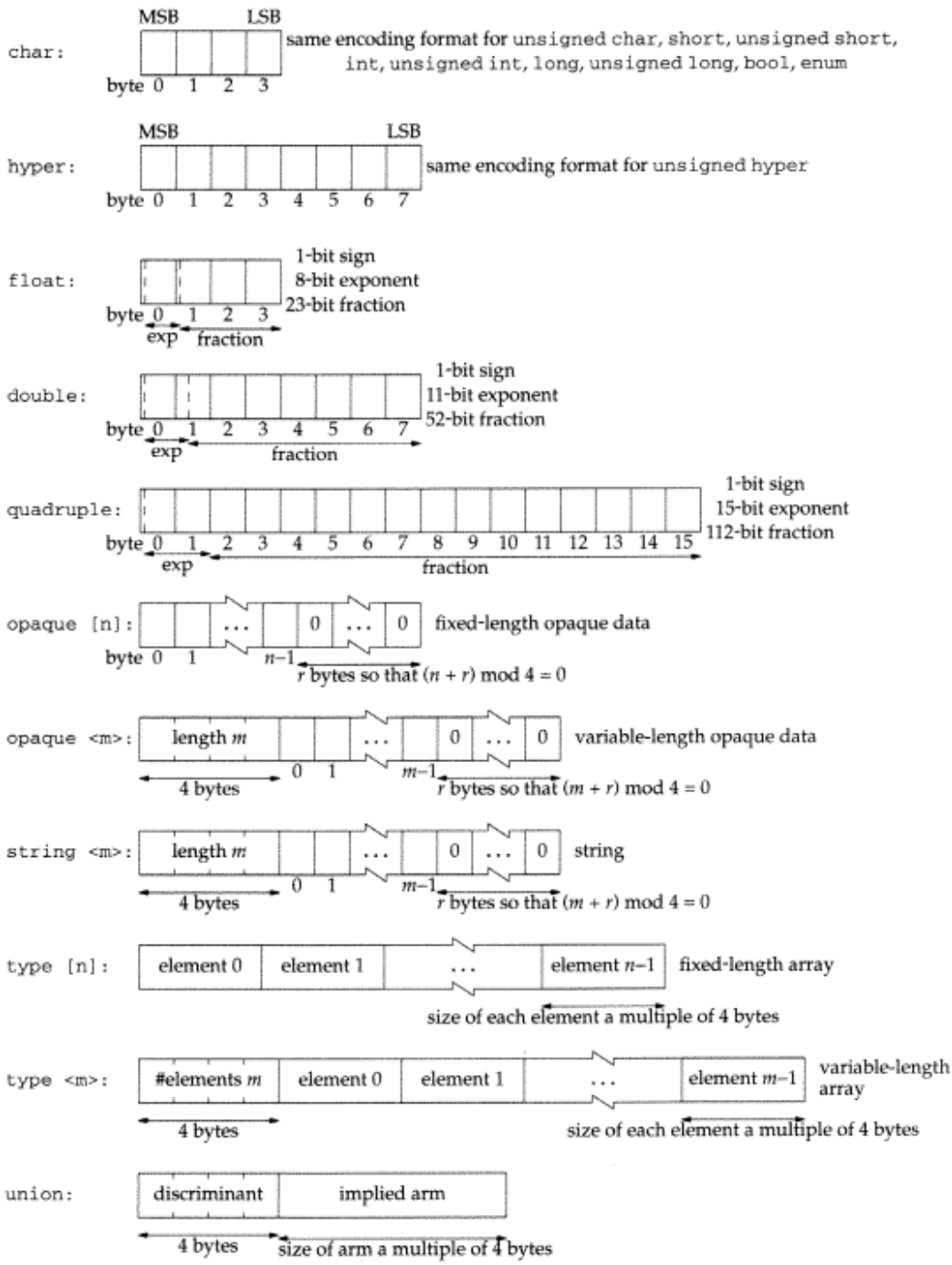

Figure 16.16 Encoding used by XDR for its various datatypes.

Declare enumeration and discriminated union

1-11 We declare an enumeration with two values, followed by a discriminated union that uses this enumeration as the discriminant. If the discriminant value is `RESULT_INT`, then an integer value is transmitted after the discriminant value. If the discriminant value is `RESULT_DOUBLE`, then a double precision floating point value is transmitted after the discriminant value; otherwise, nothing is transmitted after the discriminant value.

Declare structure

12-21 We declare a structure containing numerous XDR datatypes.

Since we do not declare any RPC procedures, if we look at all the files generated by `rpcgen` in Figure 16.4, we see that the client stub and server stub are not generated by `rpcgen`. But it still generates the `data.h` header and the `data_xdr.c` file containing the XDR functions to encode or decode the data items that we declared in our `data.x` file.

Figure 16.17 shows the `data.h` header that is generated. The contents of this header are what we expect, given the conversions shown in Figure 16.14.

In the file `data_xdr.c`, a function is defined named `xdr_data` that we can call to encode or decode the contents of the `data` structure that we define. (The function name suffix of `_data` comes from the name of our structure in Figure 16.15.) The first program that we write is called `write.c`, and it sets the values of all the variables in the `data` structure, calls the `xdr_data` function to encode all the fields into XDR format, and then writes the result to standard output.

Figure 16.18 shows this program.

Set structure members to some nonzero value

12-32 We first set all the members of the `data` structure to some nonzero value. In the case of variable-length fields, we must set the count and that number of values. For the discriminated union, we set the discriminant to `RESULT_INT` and the integer value to 123.

Allocate suitably aligned buffer

33 We call `malloc` to allocate room for the buffer that the XDR routines will store into, since it must be aligned on a 4-byte boundary, and just allocating a `char` array does not guarantee this alignment.

Create XDR memory stream

34 The runtime function `xdrmem_create` initializes the buffer pointed to by `buff` for XDR to use as a memory stream. We allocate a variable of type `XDR` named `xhandle` and pass the address of this variable as the first argument. The XDR runtime maintains the information in this variable (buffer pointer, current position in the buffer, and so on). The final argument is `XDR_ENCODE`, which tells XDR that we will be going from host format (our `out` structure) into XDR format.

sunrpc/xdr1/data.h

```
 1 /*
 2  * Please do not edit this file.  It was generated using rpcgen.
 3  */

 4 #ifndef _DATA_H_RPCGEN
 5 #define _DATA_H_RPCGEN

 6 enum result_t {
 7     RESULT_INT = 1,
 8     RESULT_DOUBLE = 2
 9 };
10 typedef enum result_t result_t;

11 struct union_arg {
12     result_t result;
13     union {
14         int     intval;
15         double  doubleval;
16     } union_arg_u;
17 };
18 typedef struct union_arg union_arg;

19 struct data {
20     short   short_arg;
21     long    long_arg;
22     char   *vstring_arg;
23     char    fopaque_arg[3];
24     struct {
25         u_int   vopaque_arg_len;
26         char   *vopaque_arg_val;
27     } vopaque_arg;
28     short   fshort_arg[4];
29     struct {
30         u_int   vlong_arg_len;
31         long   *vlong_arg_val;
32     } vlong_arg;
33     union_arg uarg;
34 };
35 typedef struct data data;

36         /* the xdr functions */
37 extern bool_t xdr_result_t(XDR *, result_t *);
38 extern bool_t xdr_union_arg(XDR *, union_arg *);
39 extern bool_t xdr_data(XDR *, data *);

40 #endif  /* !_DATA_H_RPCGEN */
```

sunrpc/xdr1/data.h

Figure 16.17 Header generated by `rpcgen` from Figure 16.15.

sunrpc/xdr1/write.c

```
 1 #include    "unpipc.h"
 2 #include    "data.h"

 3 int
 4 main(int argc, char **argv)
 5 {
 6     XDR     xhandle;
 7     data    out;                /* the structure whose values we store */
 8     char   *buff;               /* the result of the XDR encoding */
 9     char    vop[2];
10     long    vlong[3];
11     u_int   size;

12     out.short_arg = 1;
13     out.long_arg = 2;
14     out.vstring_arg = "hello, world";   /* pointer assignment */

15     out.fopaque_arg[0] = 99;    /* fixed-length opaque */
16     out.fopaque_arg[1] = 88;
17     out.fopaque_arg[2] = 77;

18     vop[0] = 33;                /* variable-length opaque */
19     vop[1] = 44;
20     out.vopaque_arg.vopaque_arg_len = 2;
21     out.vopaque_arg.vopaque_arg_val = vop;

22     out.fshort_arg[0] = 9999;   /* fixed-length array */
23     out.fshort_arg[1] = 8888;
24     out.fshort_arg[2] = 7777;
25     out.fshort_arg[3] = 6666;

26     vlong[0] = 123456;          /* variable-length array */
27     vlong[1] = 234567;
28     vlong[2] = 345678;
29     out.vlong_arg.vlong_arg_len = 3;
30     out.vlong_arg.vlong_arg_val = vlong;

31     out.uarg.result = RESULT_INT;   /* discriminated union */
32     out.uarg.union_arg_u.intval = 123;

33     buff = Malloc(BUFFSIZE);    /* must be aligned on 4-byte boundary */
34     xdrmem_create(&xhandle, buff, BUFFSIZE, XDR_ENCODE);

35     if (xdr_data(&xhandle, &out) != TRUE)
36         err_quit("xdr_data error");
37     size = xdr_getpos(&xhandle);
38     Write(STDOUT_FILENO, buff, size);

39     exit(0);
40 }
```

sunrpc/xdr1/write.c

Figure 16.18 Initialize the `data` structure and write it in XDR format.

Encode the structure

35-36 We call the `xdr_data` function, which was generated by `rpcgen` in the file `data_xdr.c`, and it encodes the `out` structure into XDR format. A return value of `TRUE` indicates success.

Obtain size of encoded data and `write`

37-38 The function `xdr_getpos` returns the current position of the XDR runtime in the output buffer (i.e., the byte offset of the next byte to store into), and we use this as the size of our `write`.

Figure 16.19 shows our `read` program, which reads the file that was written by the previous program, printing the values of all the members of the `data` structure.

Allocate suitably aligned buffer

11-13 We call `malloc` to allocate a buffer that is suitably aligned and read the file that was generated by the previous program into the buffer.

Create XDR memory stream, initialize buffer, and decode

14-17 We initialize an XDR memory stream, this time specifying `XDR_DECODE` to indicate that we want to convert from XDR format into host format. We initialize our `in` structure to 0 and call `xdr_data` to decode the buffer `buff` into our structure `in`. We must initialize the XDR destination to 0 (the `in` structure), because some of the XDR routines (notably `xdr_string`) require this. `xdr_data` is the same function that we called from Figure 16.18; what has changed is the final argument to `xdrmem_create`: in the previous program, we specified `XDR_ENCODE`, but in this program, we specify `XDR_DECODE`. This value is saved in the XDR handle (`xhandle`) by `xdrmem_create` and then used by the XDR runtime to determine whether to encode or decode the data.

Print structure values

18-42 We print all the members of our `data` structure.

Free any XDR-allocated memory

43 We call `xdr_free` to free the dynamic memory that the XDR runtime might have allocated (see also Exercise 16.10).

We now run our `write` program on a Sparc, redirecting standard output to a file named `data`:

```
solaris % write > data
solaris % ls -l data
-rw-rw-r--   1 rstevens other1        76 Apr 23 12:32 data
```

We see that the file size is 76 bytes, and that corresponds to Figure 16.20, which details the storage of the data (nineteen 4-byte values).

sunrpc/xdr1/read.c

```
1 #include    "unpipc.h"
2 #include    "data.h"

3 int
4 main(int argc, char **argv)
5 {
6     XDR     xhandle;
7     int     i;
8     char   *buff;
9     data    in;
10    ssize_t n;

11    buff = Malloc(BUFFSIZE);    /* must be aligned on 4-byte boundary */
12    n = Read(STDIN_FILENO, buff, BUFFSIZE);
13    printf("read %ld bytes\n", (long) n);

14    xdrmem_create(&xhandle, buff, n, XDR_DECODE);
15    memset(&in, 0, sizeof(in));
16    if (xdr_data(&xhandle, &in) != TRUE)
17        err_quit("xdr_data error");

18    printf("short_arg = %d, long_arg = %ld, vstring_arg = '%s'\n",
19           in.short_arg, in.long_arg, in.vstring_arg);

20    printf("fopaque[] = %d, %d, %d\n",
21           in.fopaque_arg[0], in.fopaque_arg[1], in.fopaque_arg[2]);

22    printf("vopaque<> =");
23    for (i = 0; i < in.vopaque_arg.vopaque_arg_len; i++)
24        printf(" %d", in.vopaque_arg.vopaque_arg_val[i]);
25    printf("\n");

26    printf("fshort_arg[] = %d, %d, %d, %d\n", in.fshort_arg[0],
27           in.fshort_arg[1], in.fshort_arg[2], in.fshort_arg[3]);

28    printf("vlong<> =");
29    for (i = 0; i < in.vlong_arg.vlong_arg_len; i++)
30        printf(" %ld", in.vlong_arg.vlong_arg_val[i]);
31    printf("\n");

32    switch (in.uarg.result) {
33    case RESULT_INT:
34        printf("uarg (int) = %d\n", in.uarg.union_arg_u.intval);
35        break;
36    case RESULT_DOUBLE:
37        printf("uarg (double) = %g\n", in.uarg.union_arg_u.doubleval);
38        break;
39    default:
40        printf("uarg (void)\n");
41        break;
42    }

43    xdr_free(xdr_data, (char *) &in);

44    exit(0);
45 }
```

sunrpc/xdr1/read.c

Figure 16.19 Read the `data` structure in XDR format and print the values.

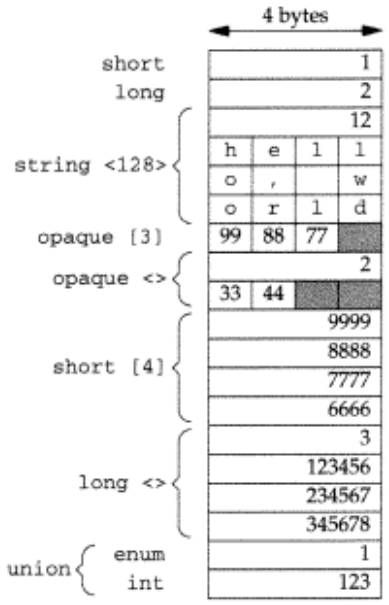

Figure 16.20 Format of the XDR stream written by Figure 16.18.

If we read this binary data file under BSD/OS or under Digital Unix, the results are what we expect:

```
bsdi % read < data
read 76 bytes
short_arg = 1, long_arg = 2, vstring_arg = 'hello, world'
fopaque[] = 99, 88, 77
vopaque<> = 33 44
fshort_arg[] = 9999, 8888, 7777, 6666
vlong<> = 123456 234567 345678
uarg (int) = 123

alpha % read < data
read 76 bytes
short_arg = 1, long_arg = 2, vstring_arg = 'hello, world'
fopaque[] = 99, 88, 77
vopaque<> = 33 44
fshort_arg[] = 9999, 8888, 7777, 6666
vlong<> = 123456 234567 345678
uarg (int) = 123
```

Example: Calculating the Buffer Size

In our previous example, we allocated a buffer of length `BUFFSIZE` (which is defined to be 8192 in our `unpipc.h` header, Figure C.1), and that was adequate. Unfortunately, no simple way exists to calculate the total size required by the XDR encoding of a given

structure. Just calculating the `sizeof` the structure is wrong, because each member is encoded separately by XDR. What we must do is go through the structure, member by member, adding the size that will be used by the XDR encoding of each member. For example, Figure 16.21 shows a simple structure with three members.

sunrpc/xdr1/example.x

```
1 const   MAXC = 4;

2 struct example {
3     short   a;
4     double  b;
5     short   c[MAXC];
6 };
```

sunrpc/xdr1/example.x

Figure 16.21 XDR specification of a simple structure.

The program shown in Figure 16.22 calculates the number of bytes that XDR requires to encode this structure to be 28 bytes.

sunrpc/xdr1/example.c

```
 1 #include    "unpipc.h"
 2 #include    "example.h"

 3 int
 4 main(int argc, char **argv)
 5 {
 6     int     size;
 7     example foo;

 8     size = RNDUP(sizeof(foo.a)) + RNDUP(sizeof(foo.b)) +
 9         RNDUP(sizeof(foo.c[0])) * MAXC;
10     printf("size = %d\n", size);
11     exit(0);
12 }
```

sunrpc/xdr1/example.c

Figure 16.22 Program to calculate the number of bytes that XDR encoding requires.

8-9 The macro `RNDUP` is defined in the `<rpc/xdr.h>` header and rounds its argument up to the next multiple of `BYTES_PER_XDR_UNIT` (4). For a fixed-length array, we calculate the size of each element and multiply this by the number of elements.

The problem with this technique is variable-length datatypes. If we declare `string d<10>`, then the maximum number of bytes required is `RNDUP(sizeof(int))` (for the length) plus `RNDUP(sizeof(char)*10)` (for the characters). But we cannot calculate a size for a variable-length declaration without a maximum, such as `float e<>`. The easiest solution is to allocate a buffer that should be larger than needed, and check for failure of the XDR routines (Exercise 16.5).

Example: Optional Data

There are three ways to specify optional data in an XDR specification file, all of which we show in Figure 16.23.

sunrpc/xdr1/opt1.x

```
 1 union optlong switch (bool flag) {
 2 case TRUE:
 3     long    val;
 4 case FALSE:
 5     void;
 6 };

 7 struct args {
 8     optlong arg1;               /* union with boolean discriminant */
 9     long    arg2 < 1 >;         /* variable-length array with one element */
10     long   *arg3;               /* pointer */
11 };
```

sunrpc/xdr1/opt1.x

Figure 16.23 XDR specification file showing three ways to specify optional data.

Declare union with boolean discriminant

1-8 We define a `union` with `TRUE` and `FALSE` arms and a structure member of this type. When the discriminant `flag` is `TRUE`, a `long` value follows; otherwise, nothing follows. When encoded by the XDR runtime, this will be encoded as either

- a 4-byte flag of 1 (`TRUE`) followed by a 4-byte value, or
- a 4-byte flag of 0 (`FALSE`).

Declare variable-length array

9 When we specify a variable-length array with a maximum of one element, it will be coded as either

- a 4-byte length of 1 followed by a 4-byte value, or
- a 4-byte length of 0.

Declare XDR pointer

10 A new way to specify optional data is shown for `arg3` (which corresponds to the last line in Figure 16.14). This argument will be coded as either

- a 4-byte value of 1 followed by a 4-byte value, or
- a 4-byte value of 0

depending on the value of the corresponding C pointer when the data is encoded. If the pointer is nonnull, the first encoding is used (8 bytes), else the second encoding is used (4 bytes of 0). This is a handy way of encoding optional data when the data is referenced in our code by a pointer.

One implementation detail that makes the first two declarations generate identical encodings is that the value of `TRUE` is 1, which is also the length of the variable-length array when one element is present.

Figure 16.24 shows the `.h` file that is generated by `rpcgen` for this specification file.

14-21 Even though all three arguments will be encoded the same by the XDR runtime, the way we set and fetch their values in C is different for each one.

sunrpc/xdr1/opt1.h

```
 7 struct optlong {
 8     int     flag;
 9     union {
10         long    val;
11     } optlong_u;
12 };
13 typedef struct optlong optlong;

14 struct args {
15     optlong arg1;
16     struct {
17         u_int   arg2_len;
18         long   *arg2_val;
19     } arg2;
20     long   *arg3;
21 };
22 typedef struct args args;
```

sunrpc/xdr1/opt1.h

Figure 16.24 C header generated by `rpcgen` for Figure 16.23.

sunrpc/xdr1/opt1z.c

```
 1 #include    "unpipc.h"
 2 #include    "opt1.h"

 3 int
 4 main(int argc, char **argv)
 5 {
 6     int     i;
 7     XDR     xhandle;
 8     char   *buff;
 9     long   *lptr;
10     args    out;
11     size_t  size;

12     out.arg1.flag = FALSE;
13     out.arg2.arg2_len = 0;
14     out.arg3 = NULL;

15     buff = Malloc(BUFFSIZE);    /* must be aligned on 4-byte boundary */
16     xdrmem_create(&xhandle, buff, BUFFSIZE, XDR_ENCODE);

17     if (xdr_args(&xhandle, &out) != TRUE)
18         err_quit("xdr_args error");
19     size = xdr_getpos(&xhandle);

20     lptr = (long *) buff;
21     for (i = 0; i < size; i += 4)
22         printf("%ld\n", (long) ntohl(*lptr++));

23     exit(0);
24 }
```

sunrpc/xdr1/opt1z.c

Figure 16.25 None of the three arguments will be encoded.

Figure 16.25 is a simple program that sets the values of the three arguments so that none of the `long` values are encoded.

Set values

12-14 We set the discriminant of the `union` for the first argument to `FALSE`, the length of the variable-length array to 0, and the pointer corresponding to the third argument to `NULL`.

Allocate suitably aligned buffer and encode

15-19 We allocate a buffer and encode our `out` structure into an XDR memory stream.

Print XDR buffer

20-22 We print the buffer, one 4-byte value at a time, using the `ntohl` function (host-to-network long integer) to convert from the XDR big-endian byte order to the host's byte order. This shows exactly what has been encoded into the buffer by the XDR runtime:

```
solaris % opt1z
0
0
0
```

As we expect, each argument is encoded as 4 bytes of 0 indicating that no value follows.

Figure 16.26 is a modification of the previous program that assigns values to all three arguments, encodes them into an XDR memory stream, and prints the stream.

Set values

12-18 To assign a value to the `union`, we set the discriminant to `TRUE` and set the value. To assign a value to the variable-length array, we set the array length to 1, and its associated pointer points to the value. To assign a value to the third argument, we set the pointer to the address of the value.

When we run this program, it prints the expected six 4-byte values:

```
solaris % opt1
1                       discriminant value of TRUE
5
1                       variable-length array length
9876
1                       flag for nonnull pointer variable
123
```

Example: Linked List Processing

Given the capability to encode optional data from the previous example, we can extend XDR's pointer notation and use it to encode and decode linked lists containing a variable number of elements. Our example is a linked list of name–value pairs, and Figure 16.27 shows the XDR specification file.

1-5 Our `mylist` structure contains one name–value pair and a pointer to the next structure. The last structure in the list will have a null `next` pointer.

sunrpc/xdr1/opt1.c

```
 1 #include    "unpipc.h"
 2 #include    "opt1.h"

 3 int
 4 main(int argc, char **argv)
 5 {
 6     int     i;
 7     XDR     xhandle;
 8     char   *buff;
 9     long    lval2, lval3, *lptr;
10     args    out;
11     size_t  size;

12     out.arg1.flag = TRUE;
13     out.arg1.optlong_u.val = 5;

14     lval2 = 9876;
15     out.arg2.arg2_len = 1;
16     out.arg2.arg2_val = &lval2;

17     lval3 = 123;
18     out.arg3 = &lval3;

19     buff = Malloc(BUFFSIZE);    /* must be aligned on 4-byte boundary */
20     xdrmem_create(&xhandle, buff, BUFFSIZE, XDR_ENCODE);

21     if (xdr_args(&xhandle, &out) != TRUE)
22         err_quit("xdr_args error");
23     size = xdr_getpos(&xhandle);

24     lptr = (long *) buff;
25     for (i = 0; i < size; i += 4)
26         printf("%ld\n", (long) ntohl(*lptr++));

27     exit(0);
28 }
```

sunrpc/xdr1/opt1.c

Figure 16.26 Assign values to all three arguments from Figure 16.23.

sunrpc/xdr1/opt2.x

```
1 struct mylist {
2     string  name <>;
3     long    value;
4     mylist *next;
5 };

6 struct args {
7     mylist *list;
8 };
```

sunrpc/xdr1/opt2.x

Figure 16.27 XDR specification for linked list of name–value pairs.

Figure 16.28 shows the `.h` file generated by `rpcgen` from Figure 16.27.

Figure 16.29 is our program that initializes a linked list containing three name–value pairs and then calls the XDR runtime to encode it.

sunrpc/xdr1/opt2.h

```
 7 struct mylist {
 8     char   *name;
 9     long    value;
10     struct mylist *next;
11 };
12 typedef struct mylist mylist;

13 struct args {
14     mylist *list;
15 };
16 typedef struct args args;
```

sunrpc/xdr1/opt2.h

Figure 16.28 C declarations corresponding to Figure 16.27.

sunrpc/xdr1/opt2.c

```
 1 #include    "unpipc.h"
 2 #include    "opt2.h"

 3 int
 4 main(int argc, char **argv)
 5 {
 6     int     i;
 7     XDR     xhandle;
 8     long   *lptr;
 9     args    out;                /* the structure that we fill */
10     char   *buff;               /* the XDR encoded result */
11     mylist  nameval[4];         /* up to 4 list entries */
12     size_t  size;

13     out.list = &nameval[2];     /* [2] -> [1] -> [0] */
14     nameval[2].name = "name1";
15     nameval[2].value = 0x1111;
16     nameval[2].next = &nameval[1];
17     nameval[1].name = "namee2";
18     nameval[1].value = 0x2222;
19     nameval[1].next = &nameval[0];
20     nameval[0].name = "nameee3";
21     nameval[0].value = 0x3333;
22     nameval[0].next = NULL;

23     buff = Malloc(BUFFSIZE);    /* must be aligned on 4-byte boundary */
24     xdrmem_create(&xhandle, buff, BUFFSIZE, XDR_ENCODE);

25     if (xdr_args(&xhandle, &out) != TRUE)
26         err_quit("xdr_args error");
27     size = xdr_getpos(&xhandle);

28     lptr = (long *) buff;
29     for (i = 0; i < size; i += 4)
30         printf("%8lx\n", (long) ntohl(*lptr++));

31     exit(0);
32 }
```

sunrpc/xdr1/opt2.c

Figure 16.29 Initialize linked list, encode it, and print result.

Initialize linked list

11-22 We allocate room for four list entries but initialize only three. The first entry is `nameval[2]`, then `nameval[1]`, and then `nameval[0]`. The head of the linked list (`out.list`) is set to `&nameval[2]`. Our reason for initializing the list in this order is just to show that the XDR runtime follows the pointers, and the order of the linked list entries that are encoded has nothing to do with which array entries are being used. We have also initialized the values to hexadecimal values, because we will print the long integer values in hex, because this makes it easier to see the ASCII values in each byte.

The output shows that each list entry is preceded by a 4-byte value of 1 (which we can consider as either a length of 1 for a variable-length array, or as the boolean value `TRUE`), and the fourth entry consists of just a 4-byte value of 0, indicating the end of the list.

```
solaris % opt2
       1                 one element follows
       5                 string length
6e616d65                 n a m e
31000000                 1, 3 bytes of pad
    1111                 corresponding value
       1                 one element follows
       6                 string length
6e616d65                 n a m e
65320000                 e 2, 2 bytes of pad
    2222                 corresponding value
       1                 one element follows
       7                 string length
6e616d65                 n a m e
65653300                 e e 3, 1 byte of pad
    3333                 corresponding value
       0                 no element follows: end-of-list
```

If XDR decodes a linked list of this form, it will dynamically allocate memory for the list entries and pointers, and link the pointers together, allowing us to traverse the list easily in C.

16.9 RPC Packet Formats

Figure 16.30 shows the format of an RPC request when encapsulated in a TCP segment.

Since TCP is a byte stream and provides no message boundaries, some method of delineating the messages must be provided by the application. Sun RPC defines a *record* as either a request or reply, and each record is composed of one or more *fragments*. Each fragment starts with a 4-byte value: the high-order bit is the final-fragment flag, and the low-order 31 bits is the count. If the final-fragment bit is 0, then additional fragments make up the record.

This 4-byte value is transmitted in the big-endian byte order, the same as all 4-byte XDR integers, but this field is not in standard XDR format because XDR does not transmit bit fields.

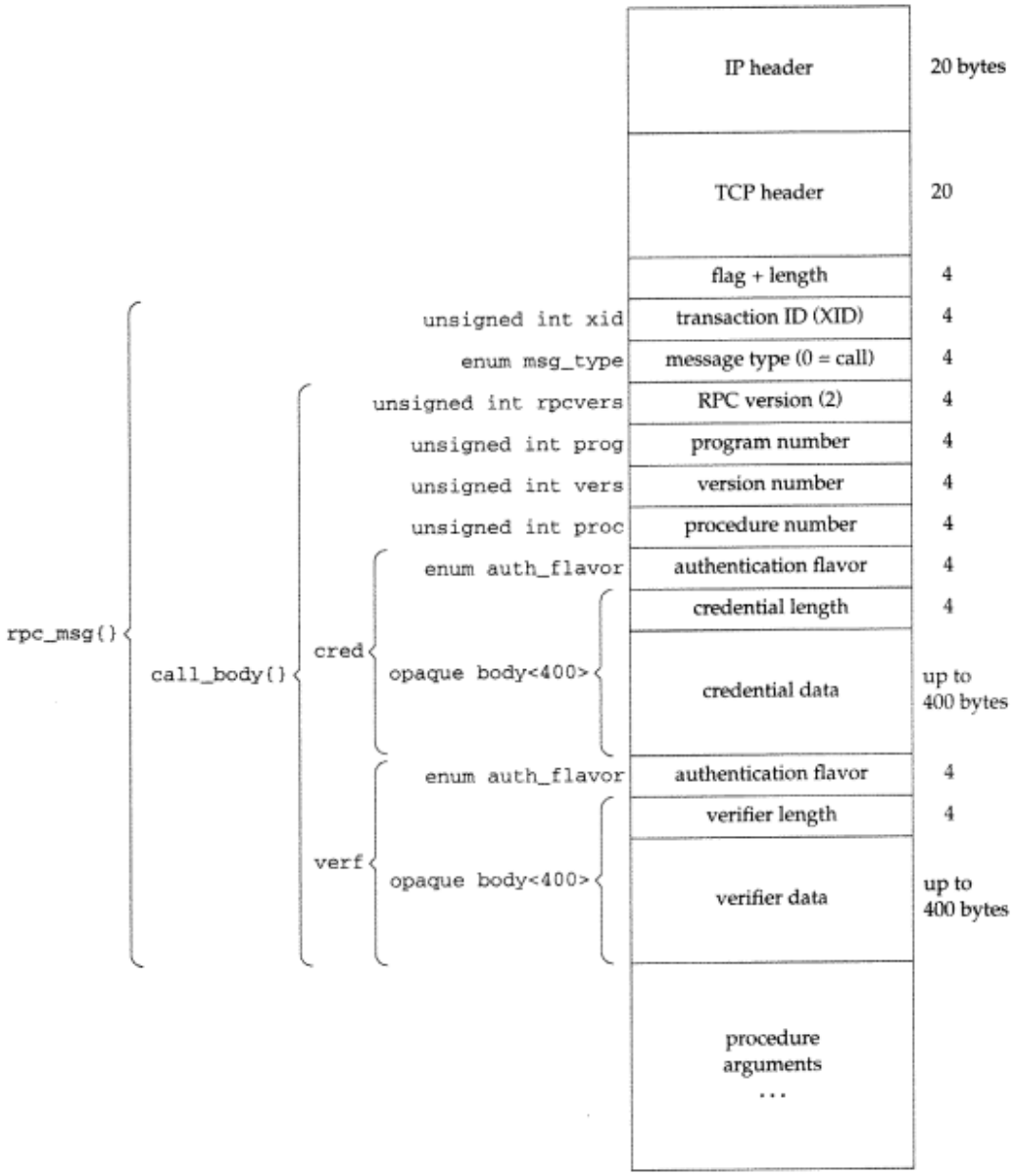

Figure 16.30 RPC request encapsulated in a TCP segment.

If UDP is being used instead of TCP, the first field following the UDP header is the XID, as we show in Figure 16.32.

With TCP, virtually no limit exists to the size of the RPC request and reply, because any number of fragments can be used and each fragment has a 31-bit length field. But with UDP, the

request and reply must each fit in a single UDP datagram, and the maximum amount of data in this datagram is 65507 bytes (assuming IPv4). Many implementations prior to the TI-RPC package further limit the size of either the request or reply to around 8192 bytes, so if more than about 8000 bytes is needed for either the request or reply, TCP should be used.

We now show the actual XDR specification of an RPC request, taken from RFC 1831. The names that we show in Figure 16.30 were taken from this specification.

```
enum auth_flavor {
  AUTH_NONE  = 0,
  AUTH_SYS   = 1,
  AUTH_SHORT = 2
    /* and more to be defined */
};

struct opaque_auth {
   auth_flavor  flavor;
   opaque       body<400>;
};

enum msg_type {
   CALL  = 0,
   REPLY = 1
};

struct call_body {
   unsigned int  rpcvers;    /* RPC version: must be 2 */
   unsigned int  prog;       /* program number */
   unsigned int  vers;       /* version number */
   unsigned int  proc;       /* procedure number */
   opaque_auth   cred;       /* caller's credentials */
   opaque_auth   verf;       /* caller's verifier */
     /* procedure-specific parameters start here */
};

struct rpc_msg {
   unsigned int  xid;
   union switch (msg_type  mtype) {
   case CALL:
      call_body   cbody;
   case REPLY:
      reply_body  rbody;
   } body;
};
```

The contents of the variable-length opaque data containing the credentials and verifier depend on the flavor of authentication. For null authentication (the default), the length of the opaque data should be 0. For Unix authentication, the opaque data contains the following information:

```
struct authsys_parms {
  unsigned int  stamp;
  string        machinename<255>;
  unsigned int  uid;
  unsigned int  gid;
  unsigned int  gids<16>;
};
```

When the credential flavor is AUTH_SYS, the verifier flavor should be AUTH_NONE.

The format of an RPC reply is more complicated than that of a request, because errors can occur in the request. Figure 16.31 shows the possibilities.

Figure 16.32 shows the format of a successful RPC reply, this time showing the UDP encapsulation.

We now show the actual XDR specification of an RPC reply, taken from RFC 1831.

```
enum reply_stat {
  MSG_ACCEPTED = 0,
  MSG_DENIED   = 1
};

enum accept_stat {
  SUCCESS       = 0, /* RPC executed successfully */
  PROG_UNAVAIL  = 1, /* program # unavailable */
  PROG_MISMATCH = 2, /* version # unavailable */
  PROC_UNAVAIL  = 3, /* procedure # unavailable */
  GARBAGE_ARGS  = 4, /* cannot decode arguments */
  SYSTEM_ERR    = 5  /* memory allocation failure, etc. */
};

struct accepted_reply {
  opaque_auth  verf;
  union switch (accept_stat  stat) {
  case SUCCESS:
    opaque  results[0];   /* procedure-specific results start here */
  case PROG_MISMATCH:
    struct {
      unsigned int  low;  /* lowest version # supported */
      unsigned int  high; /* highest version # supported */
    } mismatch_info;
  default:  /* PROG_UNAVAIL, PROC_UNAVAIL, GARBAGE_ARGS, SYSTEM_ERR */
    void;
  } reply_data;
};

union reply_body switch (reply_stat  stat) {
  case MSG_ACCEPTED:
    accepted_reply  areply;
  case MSG_DENIED:
    rejected_reply  rreply;
} reply;
```

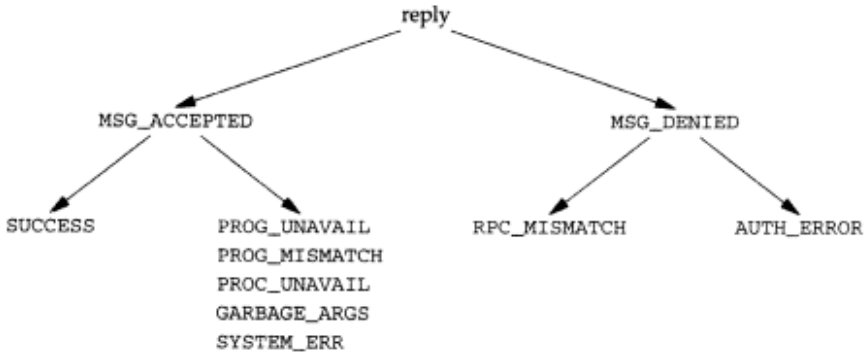

Figure 16.31 Possible RPC replies.

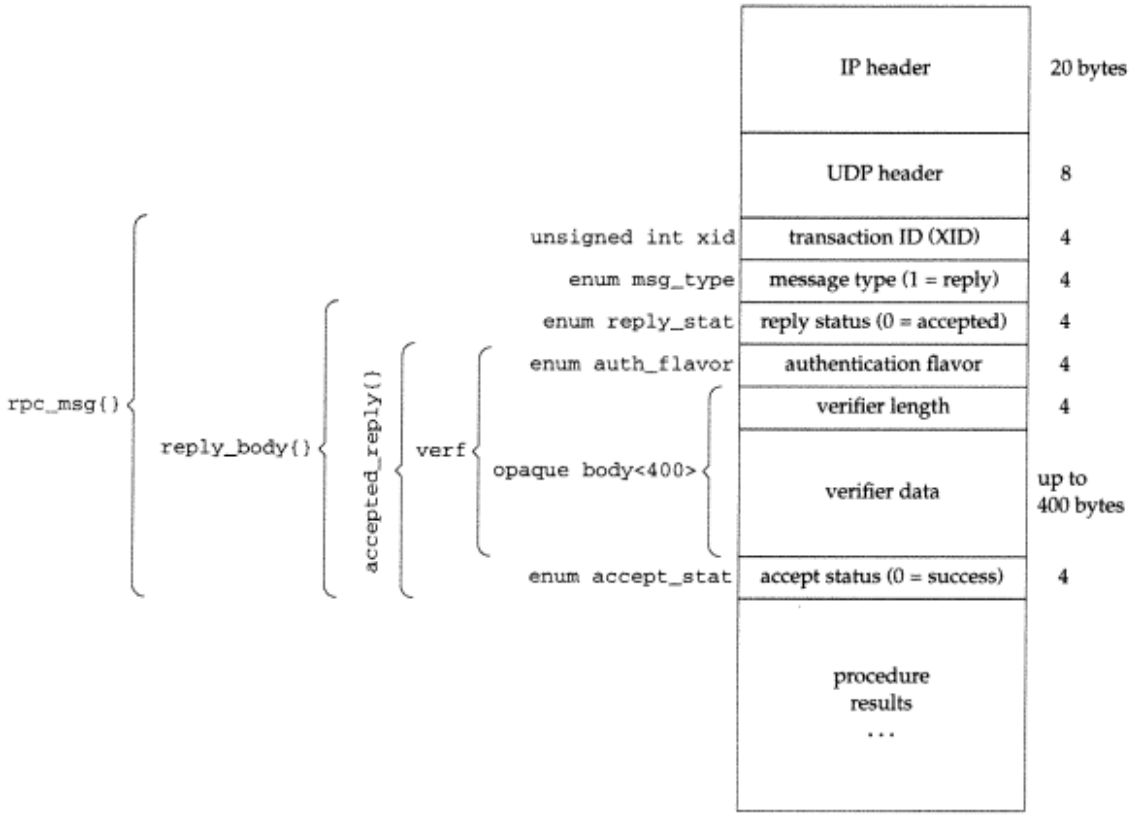

Figure 16.32 Successful RPC reply encapsulated as a UDP datagram.

The call can be rejected by the server if the RPC version number is wrong or if an authentication error occurs.

```
enum reject_stat {
   RPC_MISMATCH = 0, /* RPC version number not 2 */
   AUTH_ERROR   = 1  /* authentication error */
};

enum auth_stat {
  AUTH_OK            = 0,  /* success */
    /* following are failures at server end */
  AUTH_BADCRED       = 1,  /* bad credential (seal broken) */
  AUTH_REJECTEDCRED  = 2,  /* client must begin new session */
  AUTH_BADVERF       = 3,  /* bad verifier (seal broken) */
  AUTH_REJECTEDVERF  = 4,  /* verifier expired or replayed */
  AUTH_TOOWEAK       = 5,  /* rejected for security reasons */
    /* following are failures at client end */
  AUTH_INVALIDRESP   = 6,  /* bogus response verifier */
  AUTH_FAILED        = 7   /* reason unknown */
};

union rejected_reply switch (reject_stat  stat) {
  case RPC_MISMATCH:
    struct {
      unsigned int  low;  /* lowest RPC version # supported */
      unsigned int  high; /* highest RPC version # supported */
    } mismatch_info;
  case AUTH_ERROR:
    auth_stat  stat;
};
```

20 bytes

8

4

4

4

4

4

up to 400 bytes

4

16.10 Summary

Sun RPC allows us to code distributed applications with the client running on one host and the server on another host. We first define the server procedures that the client can call and then write an RPC specification file that describes the arguments and return values for each of these procedures. We then write the client `main` function that calls the server procedures, and the server procedures themselves. The client code appears to just call the server procedures, but underneath the covers, network communication is taking place, hidden by the various RPC runtime routines.

The `rpcgen` program is a fundamental part of building applications using RPC. It reads our specification file, and generates the client stub and the server stub, as well as generating functions that call the required XDR runtime routines that will handle all the data conversions. The XDR runtime is also a fundamental part of this process. XDR defines a standard way of exchanging various data formats between different systems that may have different-sized integers, different byte orders, different floating point formats, and the like. As we showed, we can use XDR by itself, independent of the RPC package, just for exchanging data in a standard format using any form of communications to actually transfer the data (programs written using sockets or XTI, floppy disks, CD-ROMs, or whatever).

Sun RPC provides its own form of naming, using 32-bit program numbers, 32-bit version numbers, and 32-bit procedure numbers. Each host that runs an RPC server must run a program named the port mapper (now called RPCBIND). RPC servers bind ephemeral TCP and UDP ports and then register with the port mapper to associate these ephemeral ports with the programs and versions provided by the server. When an RPC client starts, it contacts the port mapper on the server's host to obtain the desired port number, and then contacts the server itself, normally using either TCP or UDP.

By default, no authentication is provided by RPC clients, and RPC servers handle any client request that they receive. This is the same as if we were to write our own client–server using either sockets or XTI. Sun RPC provides three additional forms of authentication: Unix authentication (providing the client's hostname, user ID, and group IDs), DES authentication (based on secret key and public key cryptography), and Kerberos authentication.

Understanding the timeout and retransmission strategy of the underlying RPC package is essential to using RPC (or any form of network programming). When a reliable transport layer such as TCP is used, only a total timeout is needed by the RPC client, as any lost or duplicated packets are handled completely by the transport layer. When an unreliable transport such as UDP is used, however, the RPC package has a retry timeout in addition to a total timeout. A transaction ID is used by the RPC client to verify that a received reply is the one desired.

Any procedure call can be classified as having exactly-once semantics, at-most-once semantics, or at-least-once semantics. With local procedure calls, we normally ignore this issue, but with RPC, we must be aware of the differences, as well as understanding the difference between an idempotent procedure (one that can be called any number of times without harm) and one that is not idempotent (and must be called only once).

Sun RPC is a large package, and we have just scratched the surface. Nevertheless, given the basics that have been covered in this chapter, complete applications can be written. Using `rpcgen` hides many of the details and simplifies the coding. The Sun manuals refer to various levels of RPC coding—the simplified interface, top level, intermediate level, expert level, and bottom level—but these categorizations are meaningless. The number of functions provided by the RPC runtime is 164, with the division as follows:

- 11 `auth_` functions (authentication),
- 26 `clnt_` functions (client side),
- 5 `pmap_` functions (port mapper access),
- 24 `rpc_` functions (general),
- 44 `svc_` functions (server side), and
- 54 `xdr` functions (XDR conversions).

This compares to around 25 functions each for the sockets and XTI APIs, and less than 10 functions each for the doors API and the Posix and System V message queue APIs, semaphore APIs, and shared memory APIs. Fifteen functions deal with Posix threads, 10 functions with Posix condition variables, 11 functions with Posix read–write locks, and one function with `fcntl` record locking.

Exercises

16.1 When we start one of our servers, it registers itself with the port mapper. But if we terminate it, say with our terminal interrupt key, what happens to this registration? What happens if a client request arrives at some time later for this server?

16.2 We have a client–server using RPC with UDP, and it has no server reply cache. The client sends a request to the server but the server takes 20 seconds before sending its reply. The client times out after 15 seconds, causing the server procedure to be called a second time. What happens to the server's second reply?

16.3 The XDR `string` datatype is always encoded as a length followed by the characters. What changes if we want a fixed-length string and write, say, `char c[10]` instead of `string s<10>`?

16.4 Change the maximum size of the `string` in Figure 16.15 from 128 and 10, and run the `write` program. What happens? Now remove the maximum length specifier from the `string` declaration, that is, write `string vstring_arg<>` and compare the `data_xdr.c` file to one that is generated with a maximum length. What changes?

16.5 Change the third argument to `xdrmem_create` in Figure 16.18 (the buffer size) to 50 and see what happens.

16.6 In Section 16.5, we described the duplicate request cache that can be enabled when UDP is being used. We could say that TCP maintains its own duplicate request cache. What are we referring to, and how big is this TCP duplicate request cache? (*Hint*: How does TCP detect the receipt of duplicate data?)

16.7 Given the five elements that uniquely identify each entry in the server's duplicate request cache, in what order should these five values be compared, to require the fewest number of comparisons, when comparing a new request to a cache entry?

16.8 When watching the actual packets for our client–server from Section 16.5 using TCP, the size of the request segment is 48 bytes and the size of the reply segment is 32 bytes (ignoring the IPv4 and TCP headers). Account for these sizes (e.g., Figures 16.30 and 16.32). What will the sizes be if we use UDP instead of TCP?

16.9 Can an RPC client on a system that does not support threads call a server procedure that has been compiled to support threads? What about the differences in the arguments that we described in Section 16.2?

16.10 In our `read` program in Figure 16.19, we allocate room for the buffer into which the file is read, and that buffer contains the pointer `vstring_arg`. But where is the string stored that is pointed to by `vstring_arg`? Modify the program to verify your assumption.

16.11 Sun RPC defines the *null procedure* as the one with a procedure number of 0 (which is why we always started our procedure numbering with 1, as in Figure 16.1). Furthermore, every server stub generated by `rpcgen` automatically defines this procedure (which you can easily verify by looking at any of the server stubs generated by the examples in this chapter). The null procedure takes no arguments and returns nothing, and is often used for verifying that a given server is running, or to measure the round-trip time to the server. But if we look at the client stub, no stub is generated for this procedure. Look up the manual page for the `clnt_call` function and use it to call the null procedure for any of the servers shown in this chapter.

16.12 Why does no entry exist for a message size of 65536 for Sun RPC using UDP in Figure A.2? Why do no entries exist for message sizes of 16384 and 32768 for Sun RPC using UDP in Figure A.4?

16.13 Verify that omitting the call to `xdr_free` in Figure 16.19 introduces a *memory leak*. Add the statement

```
for ( ; ; ) {
```

immediately before calling `xdrmem_create`, and put the ending brace immediately before the call to `xdr_free`. Run the program and watch its memory size using `ps`. Then move the ending brace to follow the call to `xdr_free` and run the program again, watching its memory size.

Epilogue

This text has described in detail four different techniques for interprocess communication (IPC):

1. message passing (pipes, FIFOs, Posix and System V message queues),
2. synchronization (mutexes, condition variables, read–write locks, file and record locks, Posix and System V semaphores),
3. shared memory (anonymous, named Posix, named System V), and
4. procedure calls (Solaris doors, Sun RPC).

Message passing and procedure calls are often used by themselves, that is, they normally provide their own synchronization. Shared memory, on the other hand, usually requires some form of application-provided synchronization to work correctly. The synchronization techniques are sometimes used by themselves; that is, without the other forms of IPC.

After covering 16 chapters of details, the obvious question is: which form of IPC should be used to solve some particular problem? Unfortunately, there is no silver bullet regarding IPC. The vast number of different types of IPC provided by Unix indicates that no one solution solves all (or even most) problems. All that you can do is become familiar with the facilities provided by each form of IPC and then compare the features with the needs of your specific application.

We first list four items that must be considered, in case they are important for your application.

1. *Networked* versus *nonnetworked*. We assume that this decision has already been made and that IPC is being used between processes or threads on a single host.

If the application might be distributed across multiple hosts, consider using sockets instead of IPC, to simplify the later move to a networked application.

2. *Portability* (recall Figure 1.5). Almost all Unix systems support Posix pipes, Posix FIFOs, and Posix record locking. As of 1998, most Unix systems support System V IPC (messages, semaphores, and shared memory), whereas only a few support Posix IPC (messages, semaphores, and shared memory). More implementations of Posix IPC should appear, but it is (unfortunately) an option with Unix 98. Many Unix systems support Posix threads (which include mutexes and condition variables) or should support them in the near future. Some systems that support Posix threads do not support the process-shared attributes of mutexes and condition variables. The read–write locks required by Unix 98 should be adopted by Posix, and many versions of Unix already support some type of read–write lock. Memory-mapped I/O is widespread, and most Unix systems also provide anonymous memory mapping (either `/dev/zero` or `MAP_ANON`). Sun RPC should be available on almost all Unix systems, whereas doors are a Solaris-only feature (for now).
3. *Performance*. If this is a critical item in your design, run the programs developed in Appendix A on your own systems. Better yet, modify these programs to simulate the environment of your particular application and measure their performance in this environment.
4. *Realtime scheduling*. If you need this feature and your system supports the Posix realtime scheduling option, consider the Posix functions for message passing and synchronization (message queues, semaphores, mutexes, and condition variables). For example, when someone posts to a Posix semaphore on which multiple threads are blocked, the thread that is unblocked is chosen in a manner appropriate to the scheduling policies and parameters of the blocked threads. System V semaphores, on the other hand, make no such guarantee.

To help understand some of the features and limitations of the various types of IPC, we summarize some of the major differences:

- Pipes and FIFOs are byte streams with no message boundaries. Posix messages and System V messages have record boundaries that are maintained from the sender to the receiver. (With regard to the Internet protocols described in UNPv1, TCP is a byte stream, but UDP provides messages with record boundaries.)
- Posix message queues can send a signal to a process or initiate a new thread when a message is placed onto an empty queue. No similar form of notification is provided for System V message queues. Neither type of message queue can be used directly with either `select` or `poll` (Chapter 6 of UNPv1), although we provided workarounds in Figure 5.14 and Section 6.9.
- The bytes of data in a pipe or FIFO are first-in, first-out. Posix messages and System V messages have a priority that is assigned by the sender. When reading a Posix message queue, the highest priority message is always returned first.

When reading a System V message queue, the reader can ask for any priority message that it wants.

- When a message is placed onto a Posix or System V message queue, or written to a pipe or FIFO, one copy is delivered to exactly one thread. No peeking capability exists (similar to the sockets `MSG_PEEK` flag; Section 13.7 of UNPv1), and these messages cannot be broadcast or multicast to multiple recipients (as is possible with sockets and XTI using the UDP protocol; Chapters 18 and 19 of UNPv1).
- Mutexes, condition variables, and read–write locks are all unnamed: they are memory-based. They can be shared easily between the different threads within a single process. They can be shared between different processes only if they are stored in memory that is shared between the different processes. Posix semaphores, on the other hand, come in two flavors: named and memory-based. Named semaphores can always be shared between different processes (since they are identified by Posix IPC names), and memory-based semaphores can be shared between different processes if the semaphore is stored in memory that is shared between the different processes. System V semaphores are also named, using the `key_t` datatype, which is often obtained from the pathname of a file. These semaphores can be shared easily between different processes.
- `fcntl` record locks are automatically released by the kernel if the process holding the lock terminates without releasing the lock. System V semaphores have this feature as an option. Mutexes, condition variables, read–write locks, and Posix semaphores do not have this feature.
- Each `fcntl` lock is associated with some range of bytes (what we called a "record") in the file referenced by the descriptor. Read–write locks are not associated with any type of record.
- Posix shared memory and System V shared memory both have kernel persistence. They remain in existence until explicitly deleted, even if they are not currently being used by some process.
- The size of a Posix shared memory object can be extended while the object is being used. The size of a System V shared memory segment is fixed when it is created.
- The kernel limits for the three types of System V IPC often require tuning by the system administrator, because their default values are usually inadequate for real-world applications (Section 3.8). The kernel limits for the three types of Posix IPC usually require no tuning at all.
- Information about System V IPC objects (current size, owner ID, last-modification time, etc.) is available with a command of `IPC_STAT` with the three `XXXctl` functions, and with the `ipcs` command. No standard way exists to obtain this information about Posix IPC objects. If the implementation uses files in the filesystem for these objects, then the information is available with the `stat` function or with the `ls` command, if we know the mapping from the Posix

IPC name to the pathname. But if the implementation does not use files, this information may not be available.

- Of the various synchronization techniques—mutexes, condition variables, read–write locks, record locks, and Posix and System V semaphores—the only functions that can be called from a signal handler (Figure 5.10) are `sem_post` and `fcntl`.
- Of the various message passing techniques—pipes, FIFOs, and Posix and System V message queues—the only functions that can be called from a signal handler are `read` and `write` (for pipes and FIFOs).
- Of all the message passing techniques, only doors accurately provide the client's identity to the server (Section 15.5). In Section 5.4, we mentioned two other types of message passing that also identify the client: BSD/OS provides this identity when a Unix domain socket is used (Section 14.8 of UNPv1), and SVR4 passes the sender's identity across a pipe when a descriptor is passed across the pipe (Section 15.3.1 of APUE).

Appendix A

Performance Measurements

A.1 Introduction

In the text, we have covered six types of message passing:

- pipes,
- FIFOs,
- Posix message queues,
- System V message queues,
- doors, and
- Sun RPC,

and five types of synchronization:

- mutexes and condition variables,
- read–write locks,
- `fcntl` record locking,
- Posix semaphores, and
- System V semaphores.

We now develop some simple programs to measure the performance of these types of IPC, so we can make intelligent decisions about when to use a particular form of IPC.

When comparing the different forms of message passing, we are interested in two measurements.

1. The *bandwidth* is the speed at which we can move data through the IPC channel. To measure this, we send lots of data (millions of bytes) from one process to another. We also measure this for different sizes of the I/O operation (`writes` and `reads` for pipes and FIFOs, for example), expecting to find that the bandwidth increases as the amount of data per I/O operation increases.

2. The *latency* is how long a small IPC message takes to go from one process to another and back. We measure this as the time for a 1-byte message to go from one process to another, and back (the round-trip time).

In the real world, the bandwidth tells us how long bulk data takes to be sent across an IPC channel, but IPC is also used for small control messages, and the time required by the system to handle these small messages is provided by latency. Both numbers are important.

To measure the various forms of synchronization, we modify our program that increments a counter in shared memory, with either multiple threads or multiple processes incrementing the counter. Since the increment is a simple operation, the time required is dominated by the time of the synchronization primitives.

> The simple programs used in this Appendix to measure the various forms of IPC are loosely based on the `lmbench` suite of benchmarks that is described in [McVoy and Staelin 1996]. This is a sophisticated set of benchmarks that measure many characteristics of a Unix system (context switch time, I/O throughput, etc.) and not just IPC. The source code is publicly available: `http://www.bitmover.com/lmbench`.
>
> The numbers shown in this Appendix are provided to let us compare the techniques described in this book. An ulterior motive is to show how simple measuring these values is. Before making choices among the various techniques, you should measure these performance numbers on your own systems. Unfortunately, as easy as the numbers are to measure, when anomalies are detected, explaining these is often very hard, without access to the source code for the kernel or libraries in question.

A.2 Results

We now summarize all the results from this Appendix, for easy reference when going through the various programs that we show.

The two systems used for all the measurements are a SparcStation 4/110 running Solaris 2.6 and a Digital Alpha (DEC 3000 model 300, Pelican) running Digital Unix 4.0B. The following lines were added to the Solaris `/etc/system` file:

```
set msgsys:msginfo_msgmax = 16384
set msgsys:msginfo_msgmnb = 32768
set msgsys:msginfo_msgseg = 4096
```

This allows 16384-byte messages on a System V message queue (Figure A.2). The same changes were accomplished with Digital Unix by specifying the following lines as input to the Digital Unix `sysconfig` program:

```
ipc:
        msg-max = 16384
        msg-mnb = 32768
```

Message Passing Bandwidth Results

Figure A.2 lists the bandwidth results measured on a Sparc running Solaris 2.6, and Figure A.3 graphs these values. Figure A.4 lists the bandwidth results measured on an Alpha running Digital Unix 4.0B, and Figure A.5 graphs these values.

As we might expect, the bandwidth normally increases as the size of the message increases. Since many implementations of System V message queues have small kernel limits (Section 3.8), the largest message is 16384 bytes, and even for messages of this size, kernel defaults had to be increased. The decrease in bandwidth above 4096 bytes for Solaris is probably caused by the configuration of the internal message queue limits. For comparison with UNPv1, we also show the values for a TCP socket and a Unix domain socket. These two values were measured using programs in the `lmbench` package using only 65536-byte messages. For the TCP socket, the two processes were both on the same host.

Message Passing Latency Results

Figure A.1 lists the latency results measured under Solaris 2.6 and Digital Unix 4.0B.

	Latency (microseconds)								
	Pipe	Posix message queue	System V message queue	Doors	Sun RPC TCP	Sun RPC UDP	TCP socket	UDP socket	Unix domain socket
Solaris 2.6	324	584	260	121	1891	1677	798	755	465
DUnix 4.0B	574	995	625		1648	1373	848	639	289

Figure A.1 Latency to exchange a 1-byte message using various forms of IPC.

In Section A.4, we show the programs that measured the first six values, and the remaining three are from the `lmbench` suite. For the TCP and UDP measurements, the two processes were on the same host.

Message size	Bandwidth (MBytes/sec)							
	Pipe	Posix message queue	System V message queue	Doors	Sun RPC TCP	Sun RPC UDP	TCP socket	Unix domain socket
1024	6.3	3.7	4.9	6.3	0.5	0.5		
2048	8.7	5.3	6.3	10.0	0.9	1.0		
4096	9.8	8.4	6.6	12.6	1.6	2.8		
8192	12.7	10.2	5.8	14.4	2.4	2.8		
16384	13.1	11.6	6.1	16.8	3.2	3.4		
32768	13.2	13.4		11.4	3.5	4.3		
65536	13.7	14.4		12.2	3.7		13.2	11.3

Figure A.2 Bandwidth for various types of message passing (Solaris 2.6).

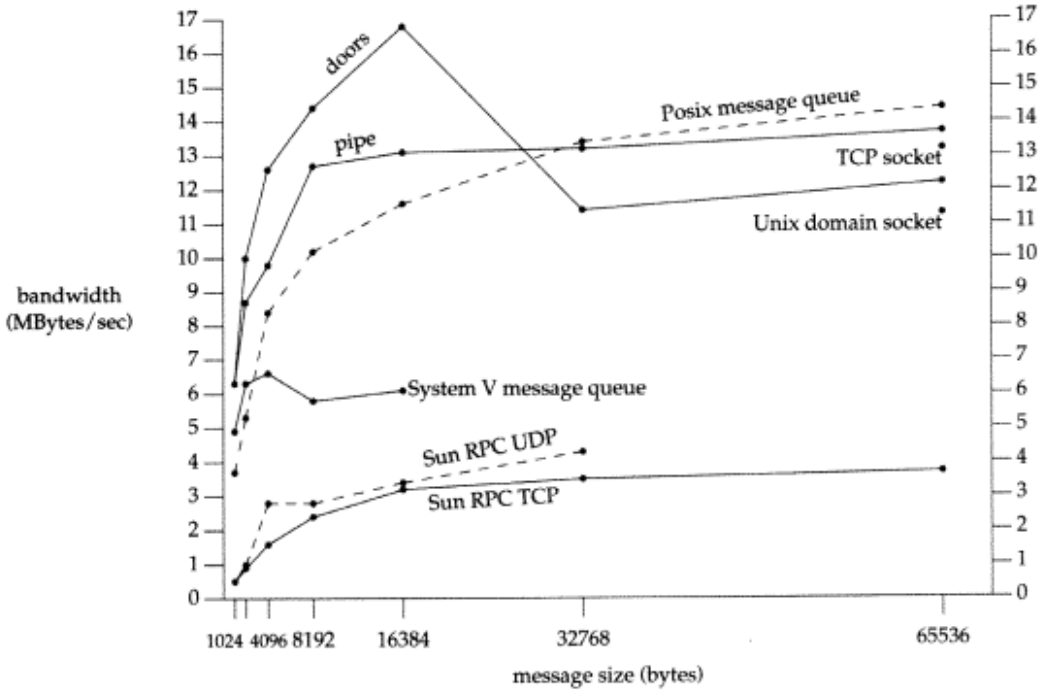

Figure A.3 Bandwidth for various types of message passing (Solaris 2.6).

Message size	Bandwidth (MBytes/sec)						
	Pipe	Posix message queue	System V message queue	Sun RPC TCP	Sun RPC UDP	TCP socket	Unix domain socket
1024	9.9	1.8	12.7	0.6	0.6		
2048	15.2	3.5	15.0	0.8	1.0		
4096	17.1	5.9	21.1	1.3	1.8		
8192	16.5	8.6	17.1	1.8	2.5		
16384	17.3	11.7	17.3	2.3			
32768	15.9	14.0		2.6			
65536	14.2	9.4		2.8		4.6	18.0

Figure A.4 Bandwidth for various types of message passing (Digital Unix 4.0B).

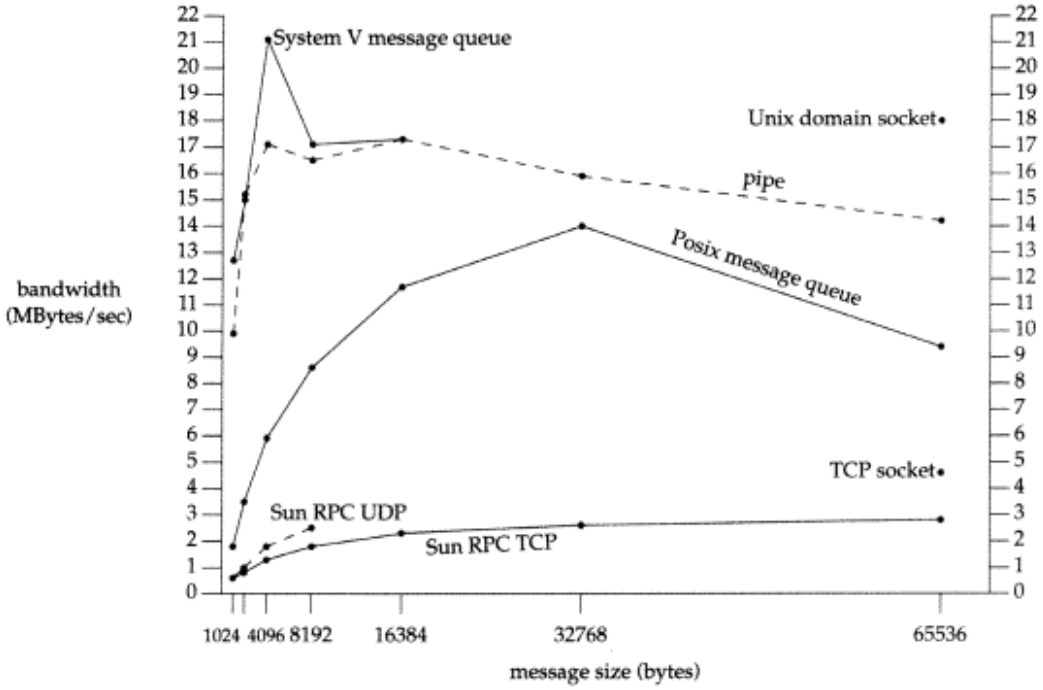

Figure A.5 Bandwidth for various types of message passing (Digital Unix 4.0B).

Thread Synchronization Results

Figure A.6 lists the time required by one or more threads to increment a counter that is in shared memory using various forms of synchronization under Solaris 2.6, and Figure A.7 graphs these values. Each thread increments the counter 1,000,000 times, and the number of threads incrementing the counter varied from one to five. Figure A.8 lists these values under Digital Unix 4.0B, and Figure A.9 graphs these values.

The reason for increasing the number of threads is to verify that the code using the synchronization technique is correct and to see whether the time starts increasing nonlinearly as the number of threads increases. We can measure `fcntl` record locking only for a single thread, because this form of synchronization works between processes and not between multiple threads within a single process.

Under Digital Unix, the times become very large for the two types of Posix semaphores with more than one thread, indicating some type of anomaly. We do not graph these values.

> One possible reason for these larger-than-expected numbers is that this program is a pathological synchronization test. That is, the threads do nothing but synchronization, and the lock is held essentially all the time. Since the threads are created with process contention scope, by default, each time a thread loses its timeslice, it probably holds the lock, so the new thread that is switched to probably blocks immediately.

Process Synchronization Results

Figures A.6 and A.7 and Figures A.8 and A.9 showed the measurements of the various synchronization techniques when used to synchronize the *threads* within a single process. Figures A.10 and A.11 show the performance of these techniques under Solaris 2.6 when the counter is shared between different *processes*. Figures A.12 and A.13 show the process synchronization results under Digital Unix 4.0B. The results are similar to the threaded numbers, although the two forms of Posix semaphores are now similar for Solaris. We plot only the first value for `fcntl` record locking, since the remaining values are so large. As we noted in Section 7.2, Digital Unix 4.0B does not support the `PTHREAD_PROCESS_SHARED` feature, so we cannot measure the mutex values between different processes. We again see some type of anomaly for Posix semaphores under Digital Unix when multiple processes are involved.

# threads	Time required to increment a counter in shared memory (seconds)						
	Posix mutex	Read–write lock	Posix memory semaphore	Posix named semaphore	System V semaphore	System V semaphore with UNDO	`fcntl` record locking
1	0.7	2.0	4.5	15.4	16.3	21.1	89.4
2	1.5	5.4	9.0	31.1	31.5	37.5	
3	2.2	7.5	14.4	46.5	48.3	57.7	
4	2.9	13.7	18.2	62.5	65.8	75.8	
5	3.7	19.7	22.8	76.8	81.8	90.0	

Figure A.6 Time required to increment a counter in shared memory (Solaris 2.6).

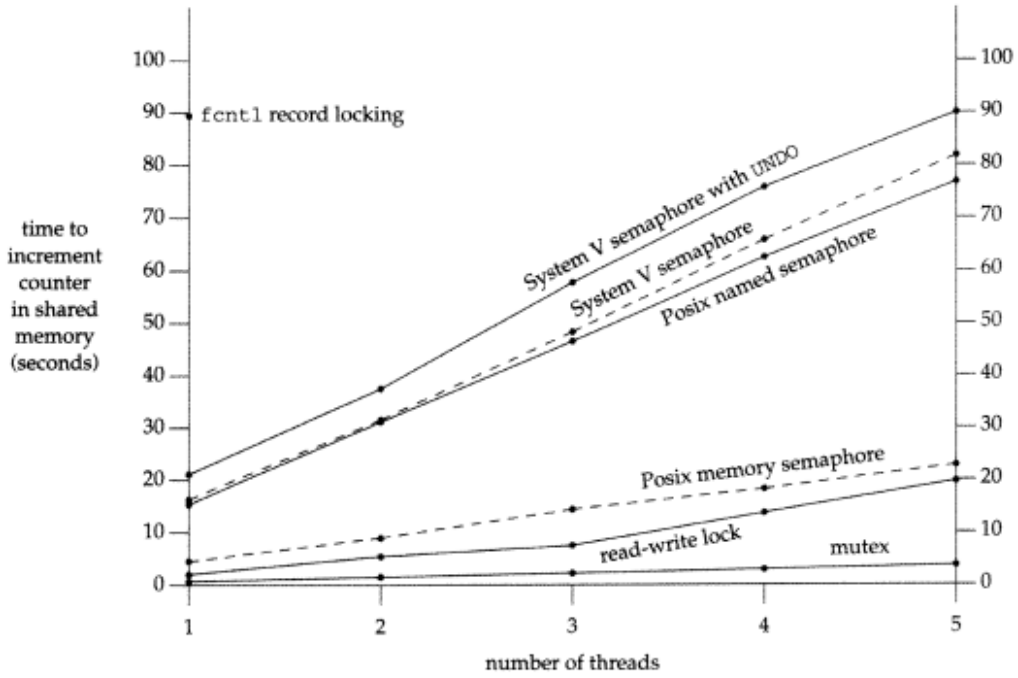

Figure A.7 Time required to increment a counter in shared memory (Solaris 2.6).

# threads	Time required to increment a counter in shared memory (seconds)						
	Posix mutex	Read–write lock	Posix memory semaphore	Posix named semaphore	System V semaphore	System V semaphore with UNDO	`fcntl` record locking
1	2.9	12.9	13.2	14.2	26.6	46.6	96.4
2	11.4	40.8	742.5	771.6	54.9	93.9	
3	28.4	73.2	1080.5	1074.7	84.5	141.9	
4	49.3	95.0	1534.1	1502.2	109.9	188.4	
5	67.3	126.3	1923.3	1764.1	137.3	233.6	

Figure A.8 Time required to increment a counter in shared memory (Digital Unix 4.0B).

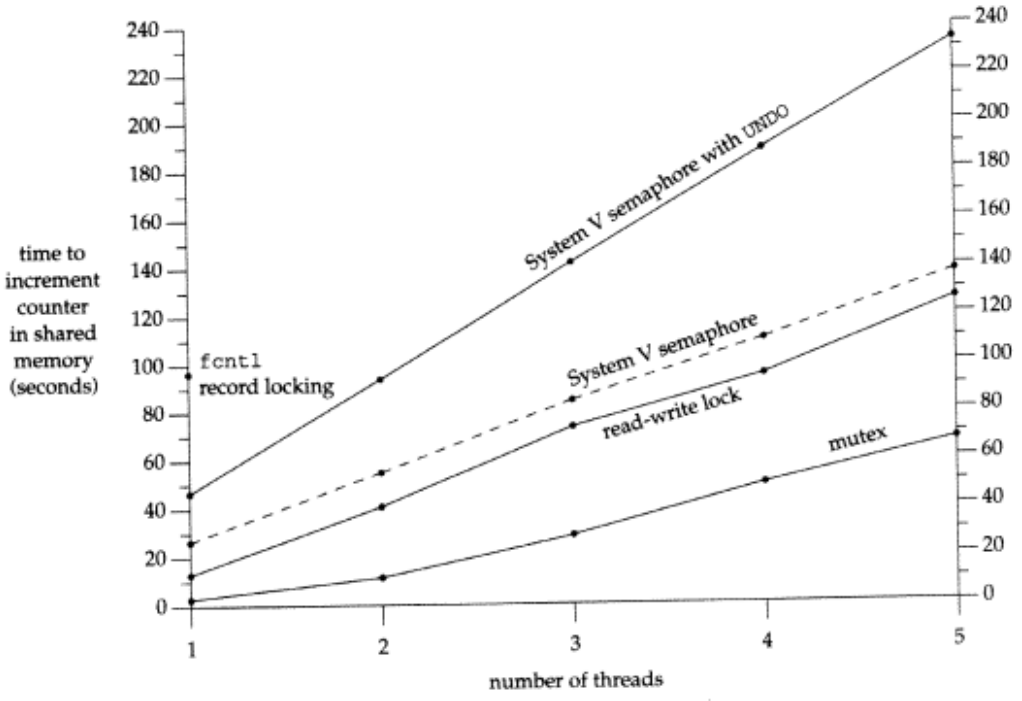

Figure A.9 Time required to increment a counter in shared memory (Digital Unix 4.0B).

# processes	Time required to increment a counter in shared memory (seconds)						
	Posix mutex	Read–write lock	Posix memory semaphore	Posix named semaphore	System V semaphore	System V semaphore with UNDO	fcntl record locking
1	0.8	1.9	13.6	14.3	17.3	22.1	90.7
2	1.6	3.9	29.2	29.2	34.9	41.6	244.5
3	2.3	6.4	41.6	42.9	54.0	60.1	376.4
4	3.1	12.2	57.3	58.8	72.4	81.9	558.0
5	4.0	20.4	70.4	73.5	87.8	102.6	764.0

Figure A.10 Time required to increment a counter in shared memory (Solaris 2.6).

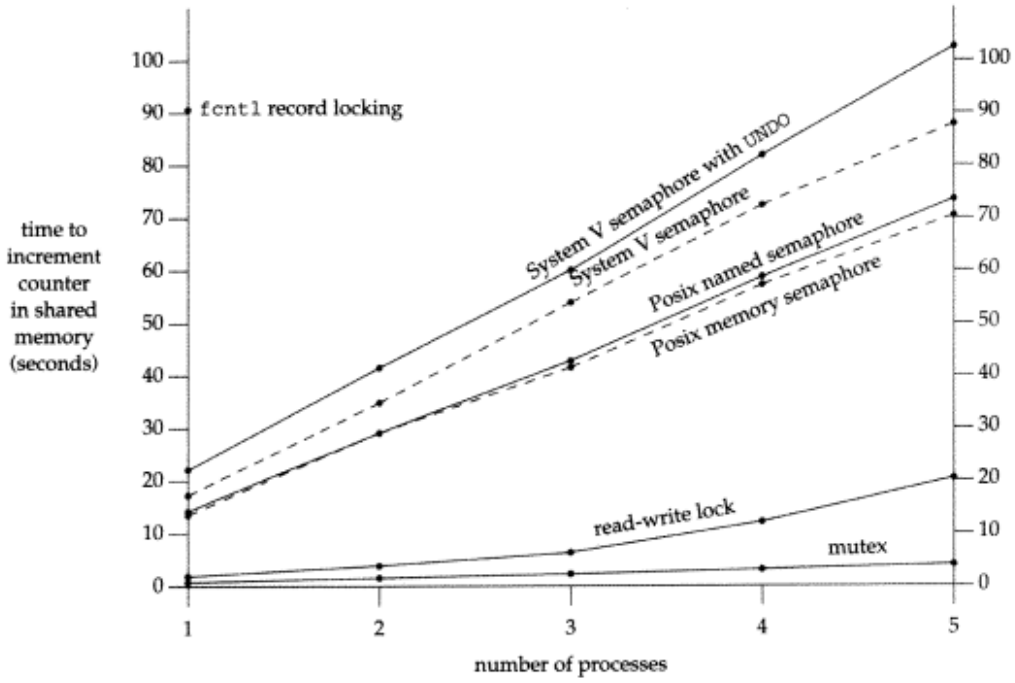

Figure A.11 Time required to increment a counter in shared memory (Solaris 2.6).

# processes	Time required to increment a counter in shared memory (seconds)				
	Posix memory semaphore	Posix named semaphore	System V semaphore	System V semaphore with UNDO	fcntl record locking
1	12.8	12.5	30.1	49.0	98.1
2	664.8	659.2	58.6	95.7	477.1
3	1236.1	1269.8	96.4	146.2	1785.2
4	1772.9	1804.1	120.3	197.0	2582.8
5	2179.9	2196.8	147.7	250.9	3419.2

Figure A.12 Time required to increment a counter in shared memory (Digital Unix 4.0B).

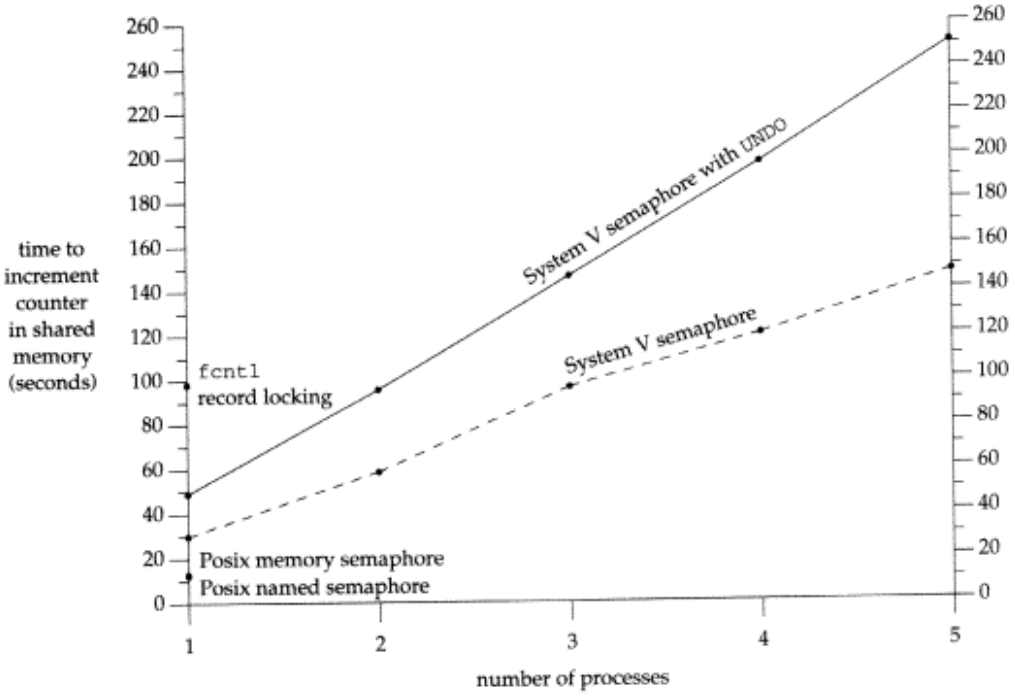

Figure A.13 Time required to increment a counter in shared memory (Digital Unix 4.0B).

A.3 Message Passing Bandwidth Programs

This section shows the three programs that measure the bandwidth of pipes, Posix message queues, and System V message queues. We showed the results of these programs in Figures A.2 and A.3.

Pipe Bandwidth Program

Figure A.14 shows an overview of the program that we are about to describe.

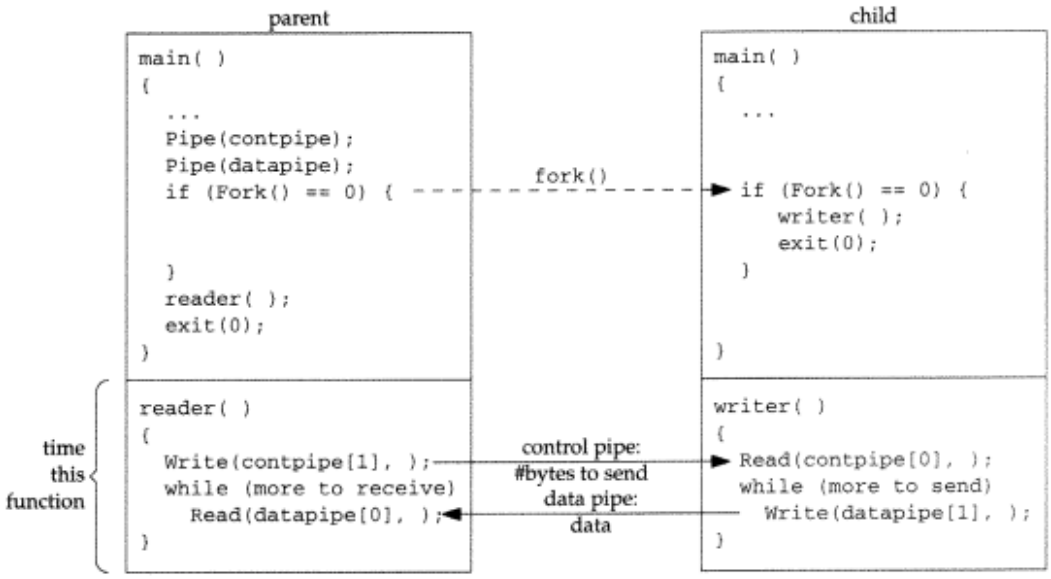

Figure A.14 Overview of program to measure the bandwidth of a pipe.

Figure A.15 shows the first half of our `bw_pipe` program, which measures the bandwidth of a pipe.

Command-line arguments

11-15 The command-line arguments specify the number of loops to perform (typically five in the measurements that follow), the number of megabytes to transfer (an argument of 10 causes $10 \times 1024 \times 1024$ bytes to be transferred), and the number of bytes for each `write` and `read` (which varies between 1024 and 65536 in the measurements that we showed).

Allocate buffer and touch it

16-17 `valloc` is a version of `malloc` that allocates the requested amount of memory starting on a page boundary. Our function `touch` (Figure A.17) stores 1 byte of data in each page of the buffer, forcing the kernel to page-in each page comprising the buffer. We do so before any timing is done.

bench/bw_pipe.c

```
 1 #include    "unpipc.h"
 2 void    reader(int, int, int);
 3 void    writer(int, int);
 4 void   *buf;
 5 int     totalnbytes, xfersize;
 6 int
 7 main(int argc, char **argv)
 8 {
 9     int     i, nloop, contpipe[2], datapipe[2];
10     pid_t   childpid;
11     if (argc != 4)
12         err_quit("usage: bw_pipe <#loops> <#mbytes> <#bytes/write>");
13     nloop = atoi(argv[1]);
14     totalnbytes = atoi(argv[2]) * 1024 * 1024;
15     xfersize = atoi(argv[3]);
16     buf = Valloc(xfersize);
17     Touch(buf, xfersize);
18     Pipe(contpipe);
19     Pipe(datapipe);
20     if ( (childpid = Fork()) == 0) {
21         writer(contpipe[0], datapipe[1]);   /* child */
22         exit(0);
23     }
24         /* parent */
25     Start_time();
26     for (i = 0; i < nloop; i++)
27         reader(contpipe[1], datapipe[0], totalnbytes);
28     printf("bandwidth: %.3f MB/sec\n",
29            totalnbytes / Stop_time() * nloop);
30     kill(childpid, SIGTERM);
31     exit(0);
32 }
```

bench/bw_pipe.c

Figure A.15 `main` function to measure the bandwidth of a pipe.

`valloc` is not part of Posix.1 and is listed as a "legacy" interface by Unix 98: it was required by an earlier version of the X/Open specification but is now optional. Our `Valloc` wrapper function calls `malloc` if `valloc` is not supported.

Create two pipes

18-19 Two pipes are created: `contpipe[0]` and `contpipe[1]` are used to synchronize the two processes at the beginning of each transfer, and `datapipe[0]` and `datapipe[1]` are used for the actual data transfer.

`fork` to create child

20-31 A child process is created, and the child (a return value of 0) calls the `writer` function while the parent calls the `reader` function. The `reader` function in the parent is

called `nloop` times. Our `start_time` function is called immediately before the loop begins, and our `stop_time` function is called as soon as the loop terminates. These two functions are shown in Figure A.17. The bandwidth that is printed is the total number of bytes transferred each time around the loop, divided by the time needed to transfer the data (`stop_time` returns this as the number of microseconds since `start_time` was called), times the number of loops. The child is then killed with the `SIGTERM` signal, and the program terminates.

The second half of the program is shown in Figure A.16, and contains the two functions `writer` and `reader`.

bench/bw_pipe.c

```
33 void
34 writer(int contfd, int datafd)
35 {
36      int     ntowrite;

37      for ( ; ; ) {
38          Read(contfd, &ntowrite, sizeof(ntowrite));

39          while (ntowrite > 0) {
40              Write(datafd, buf, xfersize);
41              ntowrite -= xfersize;
42          }
43      }
44 }

45 void
46 reader(int contfd, int datafd, int nbytes)
47 {
48      ssize_t n;

49      Write(contfd, &nbytes, sizeof(nbytes));

50      while ((nbytes > 0) &&
51             ((n = Read(datafd, buf, xfersize)) > 0)) {
52          nbytes -= n;
53      }
54 }
```

bench/bw_pipe.c

Figure A.16 `writer` and `reader` functions to measure bandwidth of a pipe.

`writer` function

33–44 This function is an infinite loop that is called by the child. It waits for the parent to say that it is ready to receive the data, by reading an integer on the control pipe that specifies the number of bytes to write to the data pipe. When this notification is received, the child writes the data across the pipe to the parent, `xfersize` bytes per `write`.

`reader` function

45–54 This function is called by the parent in a loop. Each time the function is called, it writes an integer to the control pipe telling the child how many bytes to write to the pipe. The function then calls `read` in a loop, until all the data has been received.

Our `start_time`, `stop_time`, and `touch` functions are shown in Figure A.17.

lib/timing.c

```
 1 #include    "unpipc.h"
 2 static struct timeval tv_start, tv_stop;
 3 int
 4 start_time(void)
 5 {
 6     return (gettimeofday(&tv_start, NULL));
 7 }
 8 double
 9 stop_time(void)
10 {
11     double  clockus;
12     if (gettimeofday(&tv_stop, NULL) == -1)
13         return (0.0);
14     tv_sub(&tv_stop, &tv_start);
15     clockus = tv_stop.tv_sec * 1000000.0 + tv_stop.tv_usec;
16     return (clockus);
17 }
18 int
19 touch(void *vptr, int nbytes)
20 {
21     char   *cptr;
22     static int pagesize = 0;
23     if (pagesize == 0) {
24         errno = 0;
25 #ifdef  _SC_PAGESIZE
26         if ( (pagesize = sysconf(_SC_PAGESIZE)) == -1)
27             return (-1);
28 #else
29         pagesize = getpagesize();    /* BSD */
30 #endif
31     }
32     cptr = vptr;
33     while (nbytes > 0) {
34         *cptr = 1;
35         cptr += pagesize;
36         nbytes -= pagesize;
37     }
38     return (0);
39 }
```

lib/timing.c

Figure A.17 Timing functions: `start_time`, `stop_time`, and `touch`.

The `tv_sub` function is shown in Figure A.18; it subtracts two `timeval` structures, storing the result in the first structure.

lib/tv_sub.c

```
 1 #include    "unpipc.h"

 2 void
 3 tv_sub(struct timeval *out, struct timeval *in)
 4 {
 5     if ((out->tv_usec -= in->tv_usec) < 0) {    /* out -= in */
 6         --out->tv_sec;
 7         out->tv_usec += 1000000;
 8     }
 9     out->tv_sec -= in->tv_sec;
10 }
```

lib/tv_sub.c

Figure A.18 `tv_sub` function: subtract two `timeval` structures.

On a Sparc running Solaris 2.6, if we run our program five times in a row, we get

```
solaris % bw_pipe 5 10 65536
bandwidth: 13.722 MB/sec
solaris % bw_pipe 5 10 65536
bandwidth: 13.781 MB/sec
solaris % bw_pipe 5 10 65536
bandwidth: 13.685 MB/sec
solaris % bw_pipe 5 10 65536
bandwidth: 13.665 MB/sec
solaris % bw_pipe 5 10 65536
bandwidth: 13.584 MB/sec
```

Each time we specify five loops, 10,485,760 bytes per loop, and 65536 bytes per `write` and `read`. The average of these five runs is the 13.7 MBytes/sec value shown in Figure A.2.

Posix Message Queue Bandwidth Program

Figure A.19 is our `main` program that measures the bandwidth of a Posix message queue. Figure A.20 shows the `writer` and `reader` functions. This program is similar to our previous program that measures the bandwidth of a pipe.

> Note that our program must specify the maximum number of messages that can exist on the queue, when we create the queue, and we specify this as four. The capacity of the IPC channel can affect the performance, because the writing process can send this many messages before its call to `mq_send` blocks, forcing a context switch to the reading process. Therefore, the performance of this program depends on this magic number. Changing this number from four to eight under Solaris 2.6 had no effect on the numbers in Figure A.2, but this same change under Digital Unix 4.0B decreased the performance by 12%. We would have guessed the performance would increase with a larger number of messages, because this could halve the number of context switches. But if a memory-mapped file is used, this doubles the size of that file and the amount of memory that is `mmap`ed.

bench/bw_pxmsg.c

```
 1 #include    "unpipc.h"
 2 #define NAME    "bw_pxmsg"

 3 void    reader(int, mqd_t, int);
 4 void    writer(int, mqd_t);

 5 void   *buf;
 6 int     totalnbytes, xfersize;

 7 int
 8 main(int argc, char **argv)
 9 {
10     int     i, nloop, contpipe[2];
11     mqd_t   mq;
12     pid_t   childpid;
13     struct mq_attr attr;

14     if (argc != 4)
15         err_quit("usage: bw_pxmsg <#loops> <#mbytes> <#bytes/write>");
16     nloop = atoi(argv[1]);
17     totalnbytes = atoi(argv[2]) * 1024 * 1024;
18     xfersize = atoi(argv[3]);

19     buf = Valloc(xfersize);
20     Touch(buf, xfersize);

21     Pipe(contpipe);
22     mq_unlink(Px_ipc_name(NAME));   /* error OK */
23     attr.mq_maxmsg = 4;
24     attr.mq_msgsize = xfersize;
25     mq = Mq_open(Px_ipc_name(NAME), O_RDWR | O_CREAT, FILE_MODE, &attr);

26     if ( (childpid = Fork()) == 0) {
27         writer(contpipe[0], mq);    /* child */
28         exit(0);
29     }
30         /* parent */
31     Start_time();
32     for (i = 0; i < nloop; i++)
33         reader(contpipe[1], mq, totalnbytes);
34     printf("bandwidth: %.3f MB/sec\n",
35            totalnbytes / Stop_time() * nloop);

36     kill(childpid, SIGTERM);
37     Mq_close(mq);
38     Mq_unlink(Px_ipc_name(NAME));
39     exit(0);
40 }
```

bench/bw_pxmsg.c

Figure A.19 `main` function to measure bandwidth of a Posix message queue.

bench/bw_pxmsg.c

```
41 void
42 writer(int contfd, mqd_t mqsend)
43 {
44     int     ntowrite;

45     for ( ; ; ) {
46         Read(contfd, &ntowrite, sizeof(ntowrite));

47         while (ntowrite > 0) {
48             Mq_send(mqsend, buf, xfersize, 0);
49             ntowrite -= xfersize;
50         }
51     }
52 }

53 void
54 reader(int contfd, mqd_t mqrecv, int nbytes)
55 {
56     ssize_t n;

57     Write(contfd, &nbytes, sizeof(nbytes));

58     while ((nbytes > 0) &&
59            ((n = Mq_receive(mqrecv, buf, xfersize, NULL)) > 0)) {
60         nbytes -= n;
61     }
62 }
```

bench/bw_pxmsg.c

Figure A.20 `writer` and `reader` functions to measure bandwidth of a Posix message queue.

System V Message Queue Bandwidth Program

Figure A.21 is our `main` program that measures the bandwidth of a System V message queue, and Figure A.22 shows the `writer` and `reader` functions.

bench/bw_svmsg.c

```
 1 #include    "unpipc.h"

 2 void    reader(int, int, int);
 3 void    writer(int, int);

 4 struct msgbuf *buf;
 5 int     totalnbytes, xfersize;

 6 int
 7 main(int argc, char **argv)
 8 {
 9     int     i, nloop, contpipe[2], msqid;
10     pid_t   childpid;

11     if (argc != 4)
12         err_quit("usage: bw_svmsg <#loops> <#mbytes> <#bytes/write>");
13     nloop = atoi(argv[1]);
14     totalnbytes = atoi(argv[2]) * 1024 * 1024;
15     xfersize = atoi(argv[3]);
```

```
16      buf = Valloc(xfersize);
17      Touch(buf, xfersize);
18      buf->mtype = 1;

19      Pipe(contpipe);
20      msqid = Msgget(IPC_PRIVATE, IPC_CREAT | SVMSG_MODE);

21      if ( (childpid = Fork()) == 0) {
22          writer(contpipe[0], msqid);      /* child */
23          exit(0);
24      }
25      Start_time();
26      for (i = 0; i < nloop; i++)
27          reader(contpipe[1], msqid, totalnbytes);
28      printf("bandwidth: %.3f MB/sec\n",
29             totalnbytes / Stop_time() * nloop);

30      kill(childpid, SIGTERM);
31      Msgctl(msqid, IPC_RMID, NULL);
32      exit(0);
33  }
```

bench/bw_svmsg.c

Figure A.21 `main` function to measure bandwidth of a System V message queue.

bench/bw_svmsg.c

```
34  void
35  writer(int contfd, int msqid)
36  {
37      int     ntowrite;

38      for ( ; ; ) {
39          Read(contfd, &ntowrite, sizeof(ntowrite));

40          while (ntowrite > 0) {
41              Msgsnd(msqid, buf, xfersize - sizeof(long), 0);
42              ntowrite -= xfersize;
43          }
44      }
45  }

46  void
47  reader(int contfd, int msqid, int nbytes)
48  {
49      ssize_t n;

50      Write(contfd, &nbytes, sizeof(nbytes));

51      while ((nbytes > 0) &&
52             ((n = Msgrcv(msqid, buf, xfersize - sizeof(long), 0, 0)) > 0)) {
53          nbytes -= n + sizeof(long);
54      }
55  }
```

bench/bw_svmsg.c

Figure A.22 `writer` and `reader` functions to measure bandwidth of a System V message queue.

Doors Bandwidth Program

Our program to measure the bandwidth of the doors API is more complicated than the previous ones in this section, because we must `fork` before creating the door. Our parent creates the door and then notifies the child that the door can be opened by writing to a pipe.

Another change is that unlike Figure A.14, the `reader` function is not receiving the data. Instead, the data is being received by a function named `server` that is the server procedure for the door. Figure A.23 shows an overview of the program.

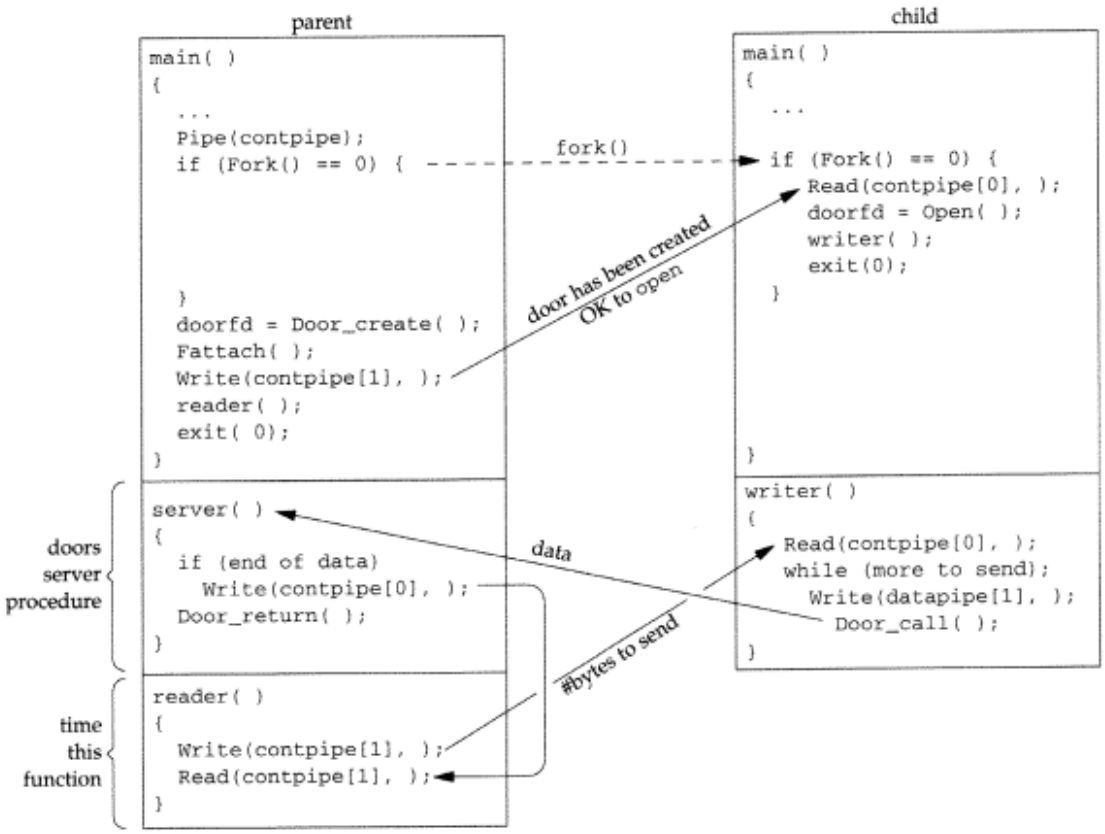

Figure A.23 Overview of program to measure the bandwidth of the doors API.

Since doors are supported only under Solaris, we simplify the program by assuming a full-duplex pipe (Section 4.4).

Another change from the previous programs is the fundamental difference between message passing, and procedure calling. In our Posix message queue program, for example, the writer just writes messages to a queue in a loop, and this is asynchronous. At some point, the queue will fill, or the writing process will lose its time slice of the processor, and the reader runs and reads the messages. If, for example, the queue held

eight messages and the writer wrote eight messages each time it ran, and the reader read all eight messages each time it ran, to send N messages would involve N/4 context switches (N/8 from the writer to the reader, and another N/8 from the reader to the writer). But the doors API is synchronous: the caller blocks each time it calls `door_call` and cannot resume until the server procedure returns. To exchange N messages now involves N×2 context switches. We will encounter the same problem when we measure the bandwidth of RPC calls. Despite the increased number of context switches, note from Figure A.3 that doors provide the fastest IPC bandwidth up through a message size of around 25000 bytes.

Figure A.24 shows the `main` function of our program. The `writer`, `server`, and `reader` functions are shown in Figure A.25.

Sun RPC Bandwidth Program

Since procedure calls in Sun RPC are synchronous, we have the same limitation that we mentioned with our doors program. It is also easier with RPC to generate two programs, a client and a server, because that is what `rpcgen` generates. Figure A.26 shows the RPC specification file. We declare a single procedure that takes a variable-length of opaque data as input and returns nothing.

Figure A.27 shows our client program, and Figure A.28 shows our server procedure. We specify the protocol (TCP or UDP) as a command-line argument for the client, allowing us to measure both protocols.

bench/bw_door.c

```
 1 #include    "unpipc.h"
 2 void    reader(int, int);
 3 void    writer(int);
 4 void    server(void *, char *, size_t, door_desc_t *, size_t);
 5 void   *buf;
 6 int     totalnbytes, xfersize, contpipe[2];
 7 int
 8 main(int argc, char **argv)
 9 {
10     int     i, nloop, doorfd;
11     char    c;
12     pid_t   childpid;
13     ssize_t n;
14     if (argc != 5)
15         err_quit("usage: bw_door <pathname> <#loops> <#mbytes> <#bytes/write>");
16     nloop = atoi(argv[2]);
17     totalnbytes = atoi(argv[3]) * 1024 * 1024;
18     xfersize = atoi(argv[4]);
19     buf = Valloc(xfersize);
20     Touch(buf, xfersize);
21     unlink(argv[1]);
22     Close(Open(argv[1], O_CREAT | O_EXCL | O_RDWR, FILE_MODE));
23     Pipe(contpipe);             /* assumes full-duplex SVR4 pipe */
24     if ( (childpid = Fork()) == 0) {
25             /* child = client = writer */
26         if ( (n = Read(contpipe[0], &c, 1)) != 1)
27             err_quit("child: pipe read returned %d", n);
28         doorfd = Open(argv[1], O_RDWR);
29         writer(doorfd);
30         exit(0);
31     }
32         /* parent = server = reader */
33     doorfd = Door_create(server, NULL, 0);
34     Fattach(doorfd, argv[1]);
35     Write(contpipe[1], &c, 1);  /* tell child door is ready */
36     Start_time();
37     for (i = 0; i < nloop; i++)
38         reader(doorfd, totalnbytes);
39     printf("bandwidth: %.3f MB/sec\n",
40            totalnbytes / Stop_time() * nloop);
41     kill(childpid, SIGTERM);
42     unlink(argv[1]);
43     exit(0);
44 }
```

bench/bw_door.c

Figure A.24 `main` function to measure the bandwidth of the doors API.

bench/bw_door.c

```
45 void
46 writer(int doorfd)
47 {
48     int     ntowrite;
49     door_arg_t arg;

50     arg.desc_ptr = NULL;          /* no descriptors to pass */
51     arg.desc_num = 0;
52     arg.rbuf = NULL;              /* no return values expected */
53     arg.rsize = 0;

54     for ( ; ; ) {
55         Read(contpipe[0], &ntowrite, sizeof(ntowrite));

56         while (ntowrite > 0) {
57             arg.data_ptr = buf;
58             arg.data_size = xfersize;
59             Door_call(doorfd, &arg);
60             ntowrite -= xfersize;
61         }
62     }
63 }

64 static int ntoread, nread;

65 void
66 server(void *cookie, char *argp, size_t arg_size,
67        door_desc_t *dp, size_t n_descriptors)
68 {
69     char    c;

70     nread += arg_size;
71     if (nread >= ntoread)
72         Write(contpipe[0], &c, 1);  /* tell reader() we are all done */

73     Door_return(NULL, 0, NULL, 0);
74 }

75 void
76 reader(int doorfd, int nbytes)
77 {
78     char    c;
79     ssize_t n;

80     ntoread = nbytes;             /* globals for server() procedure */
81     nread = 0;

82     Write(contpipe[1], &nbytes, sizeof(nbytes));

83     if ( (n = Read(contpipe[1], &c, 1)) != 1)
84         err_quit("reader: pipe read returned %d", n);
85 }
```

bench/bw_door.c

Figure A.25 `writer`, `server`, and `reader` functions for doors API bandwidth measurement.

bench/bw_sunrpc.x

```
1 %#define    DEBUG        /* so server runs in foreground */
2 struct data_in {
3   opaque    data<>;      /* variable-length opaque data */
4 };
5 program BW_SUNRPC_PROG {
6     version BW_SUNRPC_VERS {
7         void    BW_SUNRPC(data_in) = 1;
8     } = 1;
9 } = 0x31230001;
```

bench/bw_sunrpc.x

Figure A.26 RPC specification file for our bandwidth measurements of Sun RPC.

bench/bw_sunrpc_client.c

```
1 #include    "unpipc.h"
2 #include    "bw_sunrpc.h"
3 void   *buf;
4 int     totalnbytes, xfersize;
5 int
6 main(int argc, char **argv)
7 {
8     int     i, nloop, ntowrite;
9     CLIENT *cl;
10    data_in in;
11    if (argc != 6)
12        err_quit("usage: bw_sunrpc_client <hostname> <#loops>"
13                 " <#mbytes> <#bytes/write> <protocol>");
14    nloop = atoi(argv[2]);
15    totalnbytes = atoi(argv[3]) * 1024 * 1024;
16    xfersize = atoi(argv[4]);
17    buf = Valloc(xfersize);
18    Touch(buf, xfersize);
19    cl = Clnt_create(argv[1], BW_SUNRPC_PROG, BW_SUNRPC_VERS, argv[5]);
20    Start_time();
21    for (i = 0; i < nloop; i++) {
22        ntowrite = totalnbytes;
23        while (ntowrite > 0) {
24            in.data.data_len = xfersize;
25            in.data.data_val = buf;
26            if (bw_sunrpc_1(&in, cl) == NULL)
27                err_quit("%s", clnt_sperror(cl, argv[1]));
28            ntowrite -= xfersize;
29        }
30    }
31    printf("bandwidth: %.3f MB/sec\n",
32           totalnbytes / Stop_time() * nloop);
33    exit(0);
34 }
```

bench/bw_sunrpc_client.c

Figure A.27 RPC client program for bandwidth measurement.

bench/bw_sunrpc_server.c

```
 1 #include    "unpipc.h"
 2 #include    "bw_sunrpc.h"

 3 #ifndef RPCGEN_ANSIC
 4 #define bw_sunrpc_1_svc bw_sunrpc_1
 5 #endif

 6 void *
 7 bw_sunrpc_1_svc(data_in * inp, struct svc_req *rqstp)
 8 {
 9     static int nbytes;

10     nbytes = inp->data.data_len;
11     return (&nbytes);           /* must be nonnull, but xdr_void() will ignore */
12 }
```

bench/bw_sunrpc_server.c

Figure A.28 RPC server procedure for bandwidth measurement.

A.4 Message Passing Latency Programs

We now show the three programs that measure the latency of pipes, Posix message queues, and System V message queues. The performance numbers were shown in Figure A.1.

Pipe Latency Program

The program to measure the latency of a pipe is shown in Figure A.29.

`doit` function

2-9 This function runs in the parent and its clock time is measured. It writes 1 byte to a pipe (that is read by the child) and reads 1 byte from another pipe (that is written to by the child). This is what we described as the latency: how long it takes to send a small message and receive a small message in reply.

Create pipes

19-20 Two pipes are created and `fork` creates a child, leading to the arrangement shown in Figure 4.6 (but without the unused ends of each pipe closed, which is OK). Two pipes are needed for this test, since pipes are half-duplex, and we want two-way communication between the parent and child.

Child echoes 1-byte message

22-27 The child is an infinite loop that reads a 1-byte message and sends it back.

Measure parent

29-34 The parent first calls the `doit` function to send a 1-byte message to the child and read its 1-byte reply. This makes certain that both processes are running. The `doit` function is then called in a loop and the clock time is measured.

bench/lat_pipe.c

```
 1 #include    "unpipc.h"
 2 void
 3 doit(int readfd, int writefd)
 4 {
 5     char    c;
 6     Write(writefd, &c, 1);
 7     if (Read(readfd, &c, 1) != 1)
 8         err_quit("read error");
 9 }
10 int
11 main(int argc, char **argv)
12 {
13     int     i, nloop, pipe1[2], pipe2[2];
14     char    c;
15     pid_t   childpid;
16     if (argc != 2)
17         err_quit("usage: lat_pipe <#loops>");
18     nloop = atoi(argv[1]);
19     Pipe(pipe1);
20     Pipe(pipe2);
21     if ( (childpid = Fork()) == 0) {
22         for ( ; ; ) {               /* child */
23             if (Read(pipe1[0], &c, 1) != 1)
24                 err_quit("read error");
25             Write(pipe2[1], &c, 1);
26         }
27         exit(0);
28     }
29         /* parent */
30     doit(pipe2[0], pipe1[1]);
31     Start_time();
32     for (i = 0; i < nloop; i++)
33         doit(pipe2[0], pipe1[1]);
34     printf("latency: %.3f usec\n", Stop_time() / nloop);
35     Kill(childpid, SIGTERM);
36     exit(0);
37 }
```

bench/lat_pipe.c

Figure A.29 Program to measure the latency of a pipe.

On a Sparc running Solaris 2.6, if we run the program five times in a row, we get

```
solaris % lat_pipe 10000
latency: 278.633 usec
solaris % lat_pipe 10000
latency: 397.810 usec
solaris % lat_pipe 10000
latency: 392.567 usec
solaris % lat_pipe 10000
latency: 266.572 usec
solaris % lat_pipe 10000
latency: 284.559 usec
```

The average of these five runs is 324 microseconds, which we show in Figure A.1. These times include two context switches (parent-to-child, then child-to-parent), four system calls (`write` by parent, `read` by child, `write` by child, and `read` by parent), and the pipe overhead for 1 byte of data in each direction.

Posix Message Queue Latency Program

Our program to measure the latency of a Posix message queue is shown in Figure A.30.

25–28 Two message queues are created: one is used from the parent to the child, and the other from the child to the parent. Although Posix messages have a priority, allowing us to assign different priorities for the messages in the two different directions, `mq_receive` always returns the next message on the queue. Therefore, we cannot use just one queue for this test.

System V Message Queue Latency Program

Figure A.31 shows our program that measures the latency of a System V message queue.

Only one message queue is created, and it contains messages in both directions: parent-to-child and child-to-parent. The former have a type field of 1, and the latter have a type field of 2. The fourth argument to `msgrcv` in `doit` is 2, to read only messages of this type, and the fourth argument to `msgrcv` in the child is 1, to read only messages of this type.

> In Sections 9.3 and 11.3, we mentioned that many kernel-defined structures cannot be statically initialized because Posix.1 and Unix 98 guarantee only that certain members are present in the structure. These standards do not guarantee the order of these members, and the structures might contain other, nonstandard, members too. But in this program, we statically initialize the `msgbuf` structures, because System V message queues guarantee that this structure contains a `long` message type field followed by the actual data.

bench/lat_pxmsg.c

```
1 #include    "unpipc.h"
2 #define NAME1   "lat_pxmsg1"
3 #define NAME2   "lat_pxmsg2"
4 #define MAXMSG     4            /* room for 4096 bytes on queue */
5 #define MSGSIZE 1024
```

```
 6 void
 7 doit(mqd_t mqsend, mqd_t mqrecv)
 8 {
 9     char    buff[MSGSIZE];

10     Mq_send(mqsend, buff, 1, 0);
11     if (Mq_receive(mqrecv, buff, MSGSIZE, NULL) != 1)
12         err_quit("mq_receive error");
13 }

14 int
15 main(int argc, char **argv)
16 {
17     int     i, nloop;
18     mqd_t   mq1, mq2;
19     char    buff[MSGSIZE];
20     pid_t   childpid;
21     struct mq_attr attr;

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

25     attr.mq_maxmsg = MAXMSG;
26     attr.mq_msgsize = MSGSIZE;
27     mq1 = Mq_open(Px_ipc_name(NAME1), O_RDWR | O_CREAT, FILE_MODE, &attr);
28     mq2 = Mq_open(Px_ipc_name(NAME2), O_RDWR | O_CREAT, FILE_MODE, &attr);

29     if ( (childpid = Fork()) == 0) {
30         for ( ; ; ) {               /* child */
31             if (Mq_receive(mq1, buff, MSGSIZE, NULL) != 1)
32                 err_quit("mq_receive error");
33             Mq_send(mq2, buff, 1, 0);
34         }
35         exit(0);
36     }
37         /* parent */
38     doit(mq1, mq2);

39     Start_time();
40     for (i = 0; i < nloop; i++)
41         doit(mq1, mq2);
42     printf("latency: %.3f usec\n", Stop_time() / nloop);

43     Kill(childpid, SIGTERM);
44     Mq_close(mq1);
45     Mq_close(mq2);
46     Mq_unlink(Px_ipc_name(NAME1));
47     Mq_unlink(Px_ipc_name(NAME2));
48     exit(0);
49 }
```

bench/lat_pxmsg.c

Figure A.30 Program to measure the latency of a Posix message queue.

bench/lat_svmsg.c

```
 1 #include    "unpipc.h"
 2 struct msgbuf   p2child = { 1, { 0 } };      /* type = 1 */
 3 struct msgbuf   child2p = { 2, { 0 } };      /* type = 2 */
 4 struct msgbuf   inbuf;

 5 void
 6 doit(int msgid)
 7 {
 8     Msgsnd(msgid, &p2child, 0, 0);
 9     if (Msgrcv(msgid, &inbuf, sizeof(inbuf.mtext), 2, 0) != 0)
10         err_quit("msgrcv error");
11 }

12 int
13 main(int argc, char **argv)
14 {
15     int     i, nloop, msgid;
16     pid_t   childpid;

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

20     msgid = Msgget(IPC_PRIVATE, IPC_CREAT | SVMSG_MODE);

21     if ( (childpid = Fork()) == 0) {
22         for ( ; ; ) {               /* child */
23             if (Msgrcv(msgid, &inbuf, sizeof(inbuf.mtext), 1, 0) != 0)
24                 err_quit("msgrcv error");
25             Msgsnd(msgid, &child2p, 0, 0);
26         }
27         exit(0);
28     }
29         /* parent */
30     doit(msgid);

31     Start_time();
32     for (i = 0; i < nloop; i++)
33         doit(msgid);
34     printf("latency: %.3f usec\n", Stop_time() / nloop);

35     Kill(childpid, SIGTERM);
36     Msgctl(msgid, IPC_RMID, NULL);
37     exit(0);
38 }
```

bench/lat_svmsg.c

Figure A.31 Program to measure the latency of a System V message queue.

Doors Latency Program

Our program to measure the latency of the doors API is shown in Figure A.32. The child creates the door and associates the function `server` with the door. The parent then opens the door and invokes `door_call` in a loop. One byte of data is passed as an argument, and nothing is returned.

bench/lat_door.c

```
 1 #include    "unpipc.h"
 2 void
 3 server(void *cookie, char *argp, size_t arg_size,
 4        door_desc_t *dp, size_t n_descriptors)
 5 {
 6     char    c;
 7     Door_return(&c, sizeof(char), NULL, 0);
 8 }
 9 int
10 main(int argc, char **argv)
11 {
12     int     i, nloop, doorfd, contpipe[2];
13     char    c;
14     pid_t   childpid;
15     door_arg_t arg;
16     if (argc != 3)
17         err_quit("usage: lat_door <pathname> <#loops>");
18     nloop = atoi(argv[2]);
19     unlink(argv[1]);
20     Close(Open(argv[1], O_CREAT | O_EXCL | O_RDWR, FILE_MODE));
21     Pipe(contpipe);
22     if ( (childpid = Fork()) == 0) {
23         doorfd = Door_create(server, NULL, 0);
24         Fattach(doorfd, argv[1]);
25         Write(contpipe[1], &c, 1);
26         for ( ; ; )                 /* child = server */
27             pause();
28         exit(0);
29     }
30     arg.data_ptr = &c;          /* parent = client */
31     arg.data_size = sizeof(char);
32     arg.desc_ptr = NULL;
33     arg.desc_num = 0;
34     arg.rbuf = &c;
35     arg.rsize = sizeof(char);
36     if (Read(contpipe[0], &c, 1) != 1)  /* wait for child to create */
37         err_quit("pipe read error");
38     doorfd = Open(argv[1], O_RDWR);
39     Door_call(doorfd, &arg);    /* once to start everything */
40     Start_time();
41     for (i = 0; i < nloop; i++)
42         Door_call(doorfd, &arg);
43     printf("latency: %.3f usec\n", Stop_time() / nloop);
44     Kill(childpid, SIGTERM);
45     unlink(argv[1]);
46     exit(0);
47 }
```

bench/lat_door.c

Figure A.32 Program to measure the latency of the doors API.

Sun RPC Latency Program

To measure the latency of the Sun RPC API, we write two programs, a client and a server (similar to what we did when we measured the bandwidth). We use the same RPC specification file (Figure A.26), but our client calls the null procedure this time. Recall from Exercise 16.11 that this procedure takes no arguments and returns nothing, which is what we want to measure the latency. Figure A.33 shows the client. As in the solution to Exercise 16.11, we must call `clnt_call` directly to call the null procedure; a stub function is not provided in the client stub.

bench/lat_sunrpc_client.c

```
 1 #include    "unpipc.h"
 2 #include    "lat_sunrpc.h"

 3 int
 4 main(int argc, char **argv)
 5 {
 6     int     i, nloop;
 7     CLIENT *cl;
 8     struct timeval tv;

 9     if (argc != 4)
10         err_quit("usage: lat_sunrpc_client <hostname> <#loops> <protocol>");
11     nloop = atoi(argv[2]);

12     cl = Clnt_create(argv[1], BW_SUNRPC_PROG, BW_SUNRPC_VERS, argv[3]);

13     tv.tv_sec = 10;
14     tv.tv_usec = 0;
15     Start_time();
16     for (i = 0; i < nloop; i++) {
17         if (clnt_call(cl, NULLPROC, xdr_void, NULL,
18                       xdr_void, NULL, tv) != RPC_SUCCESS)
19             err_quit("%s", clnt_sperror(cl, argv[1]));
20     }
21     printf("latency: %.3f usec\n", Stop_time() / nloop);
22     exit(0);
23 }
```

bench/lat_sunrpc_client.c

Figure A.33 Sun RPC client for latency measurement.

We compile our server with the server function from Figure A.28, but that function is never called. Since we used `rpcgen` to build the client and server, we need to define at least one server procedure, but we never call it. The reason we used `rpcgen` is that it automatically generates the server `main` with the null procedure, which we need.

A.5 Thread Synchronization Programs

To measure the time required by the various synchronization techniques, we create some number of threads (one to five for the measurements shown in Figures A.6 and A.8) and each thread increments a counter in shared memory a large number of times, using the different forms of synchronization to coordinate access to the shared counter.

Posix Mutex Program

Figure A.34 shows the global variables and the `main` function for our program to measure Posix mutexes.

bench/incr_pxmutex1.c

```
 1 #include    "unpipc.h"
 2 #define MAXNTHREADS 100
 3 int     nloop;
 4 struct {
 5     pthread_mutex_t mutex;
 6     long    counter;
 7 } shared = {
 8     PTHREAD_MUTEX_INITIALIZER
 9 };
10 void   *incr(void *);
11 int
12 main(int argc, char **argv)
13 {
14     int     i, nthreads;
15     pthread_t tid[MAXNTHREADS];
16     if (argc != 3)
17         err_quit("usage: incr_pxmutex1 <#loops> <#threads>");
18     nloop = atoi(argv[1]);
19     nthreads = min(atoi(argv[2]), MAXNTHREADS);
20         /* lock the mutex */
21     Pthread_mutex_lock(&shared.mutex);
22         /* create all the threads */
23     Set_concurrency(nthreads);
24     for (i = 0; i < nthreads; i++) {
25         Pthread_create(&tid[i], NULL, incr, NULL);
26     }
27         /* start the timer and unlock the mutex */
28     Start_time();
29     Pthread_mutex_unlock(&shared.mutex);
30         /* wait for all the threads */
31     for (i = 0; i < nthreads; i++) {
32         Pthread_join(tid[i], NULL);
33     }
34     printf("microseconds: %.0f usec\n", Stop_time());
35     if (shared.counter != nloop * nthreads)
36         printf("error: counter = %ld\n", shared.counter);
37     exit(0);
38 }
```

bench/incr_pxmutex1.c

Figure A.34 Global variables and `main` function to measure Posix mutex synchronization.

Shared data

4-9 The shared data between the threads consists of the mutex itself and the counter. The mutex is statically initialized.

Lock mutex and create threads

20-26 The main thread locks the mutex before the threads are created, so that no thread can obtain the mutex until all the threads have been created and the mutex is released by the main thread. Our `set_concurrency` function is called and the threads are created. Each thread executes the `incr` function, which we show next.

Start timer and release the mutex

27-36 Once all the threads are created, the timer is started and the mutex is released. The main thread then waits for all the threads to finish, at which time the timer is stopped and the total number of microseconds is printed.

Figure A.35 shows the `incr` function that is executed by each thread.

bench/incr_pxmutex1.c

```
39 void *
40 incr(void *arg)
41 {
42     int     i;

43     for (i = 0; i < nloop; i++) {
44         Pthread_mutex_lock(&shared.mutex);
45         shared.counter++;
46         Pthread_mutex_unlock(&shared.mutex);
47     }
48     return (NULL);
49 }
```

bench/incr_pxmutex1.c

Figure A.35 Increment a shared counter using a Posix mutex.

Increment counter in critical region

44-46 The counter is incremented after obtaining the mutex. The mutex is released.

Read–Write Lock Program

Our program that uses read–write locks is a slight modification to our program that uses Posix mutexes. Each thread must obtain a write lock on the read–write lock before incrementing the shared counter.

> Few systems implement the Posix read–write locks that we described in Chapter 8, which are part of Unix 98 and are being considered by the Posix.1j working group. The read–write lock measurements described in this Appendix were made under Solaris 2.6 using the Solaris read–write locks described in the `rwlock(3T)` manual page. This implementation provides the same functionality as the proposed read–write locks, and the wrapper functions required to use these functions from the functions we described in Chapter 8 are trivial.

Under Digital Unix 4.0B, our measurements were made using the Digital thread-independent services read–write locks, described on the `tis_rwlock` manual pages. We do not show the simple modifications to Figures A.36 and A.37 for these read–write locks.

Figure A.36 shows the `main` function, and Figure A.37 shows the `incr` function.

bench/incr_rwlock1.c

```
 1 #include    "unpipc.h"
 2 #include    <synch.h>           /* Solaris header */

 3 void    Rw_wrlock(rwlock_t *rwptr);
 4 void    Rw_unlock(rwlock_t *rwptr);

 5 #define MAXNTHREADS 100

 6 int     nloop;

 7 struct {
 8     rwlock_t rwlock;            /* the Solaris datatype */
 9     long    counter;
10 } shared;                       /* init to 0 -> USYNC_THREAD */

11 void   *incr(void *);

12 int
13 main(int argc, char **argv)
14 {
15     int     i, nthreads;
16     pthread_t tid[MAXNTHREADS];

17     if (argc != 3)
18         err_quit("usage: incr_rwlock1 <#loops> <#threads>");
19     nloop = atoi(argv[1]);
20     nthreads = min(atoi(argv[2]), MAXNTHREADS);

21         /* obtain write lock */
22     Rw_wrlock(&shared.rwlock);

23         /* create all the threads */
24     Set_concurrency(nthreads);
25     for (i = 0; i < nthreads; i++) {
26         Pthread_create(&tid[i], NULL, incr, NULL);
27     }
28         /* start the timer and release the write lock */
29     Start_time();
30     Rw_unlock(&shared.rwlock);

31         /* wait for all the threads */
32     for (i = 0; i < nthreads; i++) {
33         Pthread_join(tid[i], NULL);
34     }
35     printf("microseconds: %.0f usec\n", Stop_time());
36     if (shared.counter != nloop * nthreads)
37         printf("error: counter = %ld\n", shared.counter);

38     exit(0);
39 }
```

bench/incr_rwlock1.c

Figure A.36 `main` function to measure read–write lock synchronization.

bench/incr_rwlock1.c

```
40 void *
41 incr(void *arg)
42 {
43      int     i;

44      for (i = 0; i < nloop; i++) {
45          Rw_wrlock(&shared.rwlock);
46          shared.counter++;
47          Rw_unlock(&shared.rwlock);
48      }
49      return (NULL);
50 }
```

bench/incr_rwlock1.c

Figure A.37 Increment a shared counter using a read–write lock.

Posix Memory-Based Semaphore Program

We measure both Posix memory-based semaphores and Posix named semaphores. Figure A.39 shows the `main` function for the memory-based semaphore program, and Figure A.38 shows its `incr` function.

18–19 A semaphore is created with a value of 0, and the second argument of 0 to `sem_init` says that the semaphore is shared between the threads of the calling process.

20–27 After all the threads are created, the timer is started and `sem_post` is called once by the main thread.

bench/incr_pxsem1.c

```
37 void *
38 incr(void *arg)
39 {
40      int     i;

41      for (i = 0; i < nloop; i++) {
42          Sem_wait(&shared.mutex);
43          shared.counter++;
44          Sem_post(&shared.mutex);
45      }
46      return (NULL);
47 }
```

bench/incr_pxsem1.c

Figure A.38 Increment a shared counter using a Posix memory-based semaphore.

bench/incr_pxsem1.c

```
 1 #include    "unpipc.h"
 2 #define MAXNTHREADS 100
 3 int     nloop;
 4 struct {
 5     sem_t   mutex;              /* the memory-based semaphore */
 6     long    counter;
 7 } shared;
 8 void   *incr(void *);
 9 int
10 main(int argc, char **argv)
11 {
12     int     i, nthreads;
13     pthread_t tid[MAXNTHREADS];
14     if (argc != 3)
15         err_quit("usage: incr_pxsem1 <#loops> <#threads>");
16     nloop = atoi(argv[1]);
17     nthreads = min(atoi(argv[2]), MAXNTHREADS);
18         /* initialize memory-based semaphore to 0 */
19     Sem_init(&shared.mutex, 0, 0);
20         /* create all the threads */
21     Set_concurrency(nthreads);
22     for (i = 0; i < nthreads; i++) {
23         Pthread_create(&tid[i], NULL, incr, NULL);
24     }
25         /* start the timer and release the semaphore */
26     Start_time();
27     Sem_post(&shared.mutex);
28         /* wait for all the threads */
29     for (i = 0; i < nthreads; i++) {
30         Pthread_join(tid[i], NULL);
31     }
32     printf("microseconds: %.0f usec\n", Stop_time());
33     if (shared.counter != nloop * nthreads)
34         printf("error: counter = %ld\n", shared.counter);
35     exit(0);
36 }
```

bench/incr_pxsem1.c

Figure A.39 `main` function to measure Posix memory-based semaphore synchronization.

Posix Named Semaphore Program

Figure A.41 shows the `main` function that measures Posix named semaphores, and Figure A.40 shows its `incr` function.

bench/incr_pxsem2.c

```
40 void *
41 incr(void *arg)
42 {
43      int     i;

44      for (i = 0; i < nloop; i++) {
45          Sem_wait(shared.mutex);
46          shared.counter++;
47          Sem_post(shared.mutex);
48      }
49      return (NULL);
50 }
```

bench/incr_pxsem2.c

Figure A.40 Increment a shared counter using a Posix named semaphore.

System V Semaphore Program

The `main` function of our program that measures System V semaphores is shown in Figure A.42, and Figure A.43 shows its `incr` function.

20–23 A semaphore is created consisting of one member, and its value is initialized to 0.

24–29 Two `semop` structures are initialized: one to post-to the semaphore and one to wait-for the semaphore. Notice that the `sem_flg` member of both structures is 0: the `SEM_UNDO` flag is not specified.

System V Semaphore with `SEM_UNDO` Program

The only difference in our program that measures System V semaphores with the `SEM_UNDO` feature from Figure A.42 is setting the `sem_flg` member of the two `semop` structures to `SEM_UNDO` instead of 0. We do not show this simple modification.

bench/incr_pxsem2.c

```
 1 #include    "unpipc.h"
 2 #define MAXNTHREADS 100
 3 #define NAME    "incr_pxsem2"
 4 int     nloop;
 5 struct {
 6     sem_t  *mutex;              /* pointer to the named semaphore */
 7     long    counter;
 8 } shared;
 9 void   *incr(void *);
10 int
11 main(int argc, char **argv)
12 {
13     int     i, nthreads;
14     pthread_t tid[MAXNTHREADS];
15     if (argc != 3)
16         err_quit("usage: incr_pxsem2 <#loops> <#threads>");
17     nloop = atoi(argv[1]);
18     nthreads = min(atoi(argv[2]), MAXNTHREADS);
19         /* initialize named semaphore to 0 */
20     sem_unlink(Px_ipc_name(NAME));  /* error OK */
21     shared.mutex = Sem_open(Px_ipc_name(NAME), O_CREAT | O_EXCL, FILE_MODE, 0);
22         /* create all the threads */
23     Set_concurrency(nthreads);
24     for (i = 0; i < nthreads; i++) {
25         Pthread_create(&tid[i], NULL, incr, NULL);
26     }
27         /* start the timer and release the semaphore */
28     Start_time();
29     Sem_post(shared.mutex);
30         /* wait for all the threads */
31     for (i = 0; i < nthreads; i++) {
32         Pthread_join(tid[i], NULL);
33     }
34     printf("microseconds: %.0f usec\n", Stop_time());
35     if (shared.counter != nloop * nthreads)
36         printf("error: counter = %ld\n", shared.counter);
37     Sem_unlink(Px_ipc_name(NAME));
38     exit(0);
39 }
```

bench/incr_pxsem2.c

Figure A.41 `main` function to measure Posix named semaphore synchronization.

bench/incr_svsem1.c

```
 1 #include    "unpipc.h"
 2 #define MAXNTHREADS 100
 3 int     nloop;
 4 struct {
 5     int     semid;
 6     long    counter;
 7 } shared;
 8 struct sembuf postop, waitop;
 9 void   *incr(void *);
10 int
11 main(int argc, char **argv)
12 {
13     int     i, nthreads;
14     pthread_t tid[MAXNTHREADS];
15     union semun arg;
16     if (argc != 3)
17         err_quit("usage: incr_svsem1 <#loops> <#threads>");
18     nloop = atoi(argv[1]);
19     nthreads = min(atoi(argv[2]), MAXNTHREADS);
20         /* create semaphore and initialize to 0 */
21     shared.semid = Semget(IPC_PRIVATE, 1, IPC_CREAT | SVSEM_MODE);
22     arg.val = 0;
23     Semctl(shared.semid, 0, SETVAL, arg);
24     postop.sem_num = 0;          /* and init the two semop() structures */
25     postop.sem_op = 1;
26     postop.sem_flg = 0;
27     waitop.sem_num = 0;
28     waitop.sem_op = -1;
29     waitop.sem_flg = 0;
30         /* create all the threads */
31     Set_concurrency(nthreads);
32     for (i = 0; i < nthreads; i++) {
33         Pthread_create(&tid[i], NULL, incr, NULL);
34     }
35         /* start the timer and release the semaphore */
36     Start_time();
37     Semop(shared.semid, &postop, 1);     /* up by 1 */
38         /* wait for all the threads */
39     for (i = 0; i < nthreads; i++) {
40         Pthread_join(tid[i], NULL);
41     }
42     printf("microseconds: %.0f usec\n", Stop_time());
43     if (shared.counter != nloop * nthreads)
44         printf("error: counter = %ld\n", shared.counter);
45     Semctl(shared.semid, 0, IPC_RMID);
46     exit(0);
47 }
```

bench/incr_svsem1.c

Figure A.42 `main` function to measure System V semaphore synchronization.

bench/incr_svsem1.c

```
48 void *
49 incr(void *arg)
50 {
51     int     i;

52     for (i = 0; i < nloop; i++) {
53         Semop(shared.semid, &waitop, 1);
54         shared.counter++;
55         Semop(shared.semid, &postop, 1);
56     }
57     return (NULL);
58 }
```

bench/incr_svsem1.c

Figure A.43 Increment a shared counter using a System V semaphore.

`fcntl` Record Locking Program

Our final program uses `fcntl` record locking to provide synchronization. The `main` function is shown in Figure A.45. This program will run successfully when only one thread is specified, because `fcntl` locks are between different processes, not between the different threads of a single process. When multiple threads are specified, each thread can always obtain the requested lock (that is, the calls to `writew_lock` never block, since the calling process already owns the lock), and the final value of the counter is wrong.

18–22 The pathname of the file to create and then use for locking is a command-line argument. This allows us to measure this program when this file resides on different filesystems. We expect this program to run slower when this file is on an NFS mounted filesystem, which requires that both systems (the NFS client and NFS server) support NFS record locking.

The `incr` function using record locking is shown in Figure A.44.

bench/incr_fcntl1.c

```
44 void *
45 incr(void *arg)
46 {
47     int     i;

48     for (i = 0; i < nloop; i++) {
49         Writew_lock(shared.fd, 0, SEEK_SET, 0);
50         shared.counter++;
51         Un_lock(shared.fd, 0, SEEK_SET, 0);
52     }
53     return (NULL);
54 }
```

bench/incr_fcntl1.c

Figure A.44 Increment a shared counter using `fcntl` record locking.

bench/incr_fcntl1.c

```
 4 #include    "unpipc.h"
 5 #define MAXNTHREADS 100
 6 int     nloop;
 7 struct {
 8     int     fd;
 9     long    counter;
10 } shared;
11 void   *incr(void *);
12 int
13 main(int argc, char **argv)
14 {
15     int     i, nthreads;
16     char   *pathname;
17     pthread_t tid[MAXNTHREADS];
18     if (argc != 4)
19         err_quit("usage: incr_fcntl1 <pathname> <#loops> <#threads>");
20     pathname = argv[1];
21     nloop = atoi(argv[2]);
22     nthreads = min(atoi(argv[3]), MAXNTHREADS);
23         /* create the file and obtain write lock */
24     shared.fd = Open(pathname, O_RDWR | O_CREAT | O_TRUNC, FILE_MODE);
25     Writew_lock(shared.fd, 0, SEEK_SET, 0);
26         /* create all the threads */
27     Set_concurrency(nthreads);
28     for (i = 0; i < nthreads; i++) {
29         Pthread_create(&tid[i], NULL, incr, NULL);
30     }
31         /* start the timer and release the write lock */
32     Start_time();
33     Un_lock(shared.fd, 0, SEEK_SET, 0);
34         /* wait for all the threads */
35     for (i = 0; i < nthreads; i++) {
36         Pthread_join(tid[i], NULL);
37     }
38     printf("microseconds: %.0f usec\n", Stop_time());
39     if (shared.counter != nloop * nthreads)
40         printf("error: counter = %ld\n", shared.counter);
41     Unlink(pathname);
42     exit(0);
43 }
```

bench/incr_fcntl1.c

Figure A.45 `main` function to measure `fcntl` record locking.

A.6 Process Synchronization Programs

In the programs in the previous section, sharing a counter between multiple threads was simple: we just stored the counter as a global variable. We now modify these programs to provide synchronization between different processes.

To share the counter between a parent and its children, we store the counter in shared memory that is allocated by our `my_shm` function, shown in Figure A.46.

lib/my_shm.c

```
 1 #include    "unpipc.h"
 2 void *
 3 my_shm(size_t nbytes)
 4 {
 5     void   *shared;
 6 #if     defined(MAP_ANON)
 7     shared = mmap(NULL, nbytes, PROT_READ | PROT_WRITE,
 8                   MAP_ANON | MAP_SHARED, -1, 0);
 9 #elif   defined(HAVE_DEV_ZERO)
10     int     fd;
11         /* memory map /dev/zero */
12     if ( (fd = open("/dev/zero", O_RDWR)) == -1)
13         return (MAP_FAILED);
14     shared = mmap(NULL, nbytes, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
15     close(fd);
16 #else
17 #error cannot determine what type of anonymous shared memory to use
18 #endif
19     return (shared);            /* MAP_FAILED on error */
20 }
```

lib/my_shm.c

Figure A.46 Create some shared memory for a parent and its children.

If the system supports the `MAP_ANON` flag (Section 12.4), we use it; otherwise, we memory map `/dev/zero` (Section 12.5).

Further modifications depend on the type of synchronization and what happens to the underlying datatype when `fork` is called. We described some of these details in Section 10.12.

- Posix mutex: the mutex must be stored in shared memory (with the shared counter), and the `PTHREAD_PROCESS_SHARED` attribute must be set when the mutex is initialized. We show the code for this program shortly.
- Posix read–write lock: the read–write lock must be stored in shared memory (with the shared counter), and the `PTHREAD_PROCESS_SHARED` attribute must be set when the read–write is initialized.

- Posix memory-based semaphores: the semaphore must be stored in shared memory (with the shared counter), and the second argument to `sem_init` must be 1, to specify that the semaphore is shared between processes.
- Posix named semaphores: either we can have the parent and each child call `sem_open` or we can have the parent call `sem_open`, knowing that the semaphore will be shared by the child across the `fork`.
- System V semaphores: nothing special need be coded, since these semaphores can always be shared between processes. The children just need to know the semaphore's identifier.
- `fcntl` record locking: nothing special need be coded, since descriptors are shared by the child across a `fork`.

We show only the code for the Posix mutex program.

Posix Mutex Program

The `main` function for our first program uses a Posix mutex to provide synchronization and is shown in Figure A.48. Its `incr` function is shown in Figure A.47.

19-20 Since we are using multiple processes (the children of a parent), we must place our `shared` structure into shared memory. We call our `my_shm` function (Figure A.46).

21-26 Since the mutex is in shared memory, we cannot statically initialize it, so we call `pthread_mutex_init` after setting the `PTHREAD_PROCESS_SHARED` attribute. The mutex is locked.

27-36 All the children are created, the timer is started, and the mutex is unlocked.

37-43 The parent waits for all the children and then stops the timer.

bench/incr_pxmutex5.c

```
46 void *
47 incr(void *arg)
48 {
49     int     i;

50     for (i = 0; i < nloop; i++) {
51         Pthread_mutex_lock(&shared->mutex);
52         shared->counter++;
53         Pthread_mutex_unlock(&shared->mutex);
54     }
55     return (NULL);
56 }
```

bench/incr_pxmutex5.c

Figure A.47 `incr` function to measure Posix mutex locking between processes.

bench/incr_pxmutex5.c

```
 1 #include    "unpipc.h"
 2 #define MAXNPROC    100
 3 int     nloop;
 4 struct shared {
 5     pthread_mutex_t mutex;
 6     long    counter;
 7 } *shared;                 /* pointer; actual structure in shared memory */
 8 void   *incr(void *);
 9 int
10 main(int argc, char **argv)
11 {
12     int     i, nprocs;
13     pid_t   childpid[MAXNPROC];
14     pthread_mutexattr_t mattr;
15     if (argc != 3)
16         err_quit("usage: incr_pxmutex5 <#loops> <#processes>");
17     nloop = atoi(argv[1]);
18     nprocs = min(atoi(argv[2]), MAXNPROC);
19         /* get shared memory for parent and children */
20     shared = My_shm(sizeof(struct shared));
21         /* initialize the mutex and lock it */
22     Pthread_mutexattr_init(&mattr);
23     Pthread_mutexattr_setpshared(&mattr, PTHREAD_PROCESS_SHARED);
24     Pthread_mutex_init(&shared->mutex, &mattr);
25     Pthread_mutexattr_destroy(&mattr);
26     Pthread_mutex_lock(&shared->mutex);
27         /* create all the children */
28     for (i = 0; i < nprocs; i++) {
29         if ( (childpid[i] = Fork()) == 0) {
30             incr(NULL);
31             exit(0);
32         }
33     }
34         /* parent: start the timer and unlock the mutex */
35     Start_time();
36     Pthread_mutex_unlock(&shared->mutex);
37         /* wait for all the children */
38     for (i = 0; i < nprocs; i++) {
39         Waitpid(childpid[i], NULL, 0);
40     }
41     printf("microseconds: %.0f usec\n", Stop_time());
42     if (shared->counter != nloop * nprocs)
43         printf("error: counter = %ld\n", shared->counter);
44     exit(0);
45 }
```

bench/incr_pxmutex5.c

Figure A.48 `main` function to measure Posix mutex locking between processes.

Appendix B

A Threads Primer

B.1 Introduction

This appendix summarizes the basic Posix thread functions. In the traditional Unix model, when a process needs something performed by another entity, it `fork`s a child process and lets the child perform the processing. Most network servers under Unix, for example, are written this way.

Although this paradigm has served well for many years, there are problems with `fork`:

- `fork` is expensive. Memory is copied from the parent to the child, all descriptors are duplicated in the child, and so on. Current implementations use a technique called *copy-on-write*, which avoids a copy of the parent's data space to the child until the child needs its own copy; but regardless of this optimization, `fork` is expensive.
- Interprocess communication (IPC) is required to pass information between the parent and child *after* the `fork`. Information from the parent to the child *before* the `fork` is easy, since the child starts with a copy of the parent's data space and with a copy of all the parent's descriptors. But returning information from the child to the parent takes more work.

Threads help with both problems. Threads are sometimes called *lightweight processes*, since a thread is "lighter weight" than a process. That is, thread creation can be 10–100 times faster than process creation.

All threads within a process share the same global memory. This makes the sharing of information easy between the threads, but along with this simplicity comes the problem of *synchronization*. But more than just the global variables are shared. All threads within a process share:

- process instructions,
- most data,
- open files (e.g., descriptors),
- signal handlers and signal dispositions,
- current working directory, and
- user and group IDs.

But each thread has its own:

- thread ID,
- set of registers, including program counter and stack pointer,
- stack (for local variables and return addresses),
- `errno`,
- signal mask, and
- priority.

B.2 Basic Thread Functions: Creation and Termination

In this section, we cover five basic thread functions.

`pthread_create` Function

When a program is started by `exec`, a single thread is created, called the *initial thread* or *main thread*. Additional threads are created by `pthread_create`.

```
#include <pthread.h>

int pthread_create(pthread_t *tid, const pthread_attr_t *attr,
                   void *(*func)(void *), void *arg);
```

Returns: 0 if OK, positive Exxx value on error

Each thread within a process is identified by a *thread ID*, whose datatype is `pthread_t`. On successful creation of a new thread, its ID is returned through the pointer *tid*.

Each thread has numerous *attributes*: its priority, its initial stack size, whether it should be a daemon thread or not, and so on. When a thread is created, we can specify these attributes by initializing a `pthread_attr_t` variable that overrides the default. We normally take the default, in which case, we specify the *attr* argument as a null pointer.

Finally, when we create a thread, we specify a function for it to execute, called its *thread start function*. The thread starts by calling this function and then terminates either explicitly (by calling `pthread_exit`) or implicitly (by letting this function return). The

address of the function is specified as the *func* argument, and this function is called with a single pointer argument, *arg*. If we need multiple arguments to the function, we must package them into a structure and then pass the address of this structure as the single argument to the start function.

Notice the declarations of *func* and *arg*. The function takes one argument, a generic pointer (`void *`), and returns a generic pointer (`void *`). This lets us pass one pointer (to anything we want) to the thread, and lets the thread return one pointer (again, to anything we want).

The return value from the Pthread functions is normally 0 if OK or nonzero on an error. But unlike most system functions, which return –1 on an error and set `errno` to a positive value, the Pthread functions return the positive error indication as the function's return value. For example, if `pthread_create` cannot create a new thread because we have exceeded some system limit on the number of threads, the function return value is `EAGAIN`. The Pthread functions do not set `errno`. The convention of 0 for OK or nonzero for an error is fine, since all the `Exxx` values in `<sys/errno.h>` are positive. A value of 0 is never assigned to one of the `Exxx` names.

`pthread_join` Function

We can wait for a given thread to terminate by calling `pthread_join`. Comparing threads to Unix processes, `pthread_create` is similar to `fork`, and `pthread_join` is similar to `waitpid`.

```
#include <pthread.h>

int pthread_join(pthread_t tid, void **status);
```

Returns: 0 if OK, positive `Exxx` value on error

We must specify the *tid* of the thread for which we wish to wait. Unfortunately, we have no way to wait for any of our threads (similar to `waitpid` with a process ID argument of –1).

If the *status* pointer is nonnull, the return value from the thread (a pointer to some object) is stored in the location pointed to by *status*.

`pthread_self` Function

Each thread has an ID that identifies it within a given process. The thread ID is returned by `pthread_create`, and we saw that it was used by `pthread_join`. A thread fetches this value for itself using `pthread_self`.

```
#include <pthread.h>

pthread_t pthread_self(void);
```

Returns: thread ID of calling thread

Comparing threads to Unix processes, `pthread_self` is similar to `getpid`.

`pthread_detach` Function

A thread is either *joinable* (the default) or *detached*. When a joinable thread terminates, its thread ID and exit status are retained until another thread in the process calls `pthread_join`. But a detached thread is like a daemon process: when it terminates, all its resources are released, and we cannot wait for it to terminate. If one thread needs to know when another thread terminates, it is best to leave the thread as joinable.

The `pthread_detach` function changes the specified thread so that it is detached.

```
#include <pthread.h>

int pthread_detach(pthread_t tid);
```

Returns: 0 if OK, positive Exxx value on error

This function is commonly called by the thread that wants to detach itself, as in

```
pthread_detach(pthread_self());
```

`pthread_exit` Function

One way for a thread to terminate is to call `pthread_exit`.

```
#include <pthread.h>

void pthread_exit(void *status);
```

Does not return to caller

If the thread is not detached, its thread ID and exit status are retained for a later `pthread_join` by some other thread in the calling process.

The pointer *status* must not point to an object that is local to the calling thread (e.g., an automatic variable in the thread start function), since that object disappears when the thread terminates.

A thread can terminate in two other ways:

- The function that started the thread (the third argument to `pthread_create`) can `return`. Since this function must be declared as returning a `void` pointer, that return value is the exit status of the thread.
- If the `main` function of the process returns or if any thread calls `exit` or `_exit`, the process terminates immediately, including any threads that are still running.

C.1

Appendix C

Miscellaneous Source Code

C.1 `unpipc.h` Header

Almost every program in the text includes our `unpipc.h` header, shown in Figure C.1. This header includes all the standard system headers that most network programs need, along with some general system headers. It also defines constants such as `MAXLINE` and ANSI C function prototypes for the functions that we define in the text (e.g., `px_ipc_name`) and all the wrapper functions that we use. We do not show these prototypes.

lib/unpipc.h

```
 1 /* Our own header.  Tabs are set for 4 spaces, not 8 */

 2 #ifndef __unpipc_h
 3 #define __unpipc_h

 4 #include    "../config.h"       /* configuration options for current OS */
 5                                 /* "../config.h" is generated by configure */

 6 /* If anything changes in the following list of #includes, must change
 7    ../aclocal.m4 and ../configure.in also, for configure's tests. */

 8 #include    <sys/types.h>       /* basic system data types */
 9 #include    <sys/time.h>        /* timeval{} for select() */
10 #include    <time.h>            /* timespec{} for pselect() */
11 #include    <errno.h>
12 #include    <fcntl.h>           /* for nonblocking */
13 #include    <limits.h>          /* PIPE_BUF */
14 #include    <signal.h>
15 #include    <stdio.h>
16 #include    <stdlib.h>
17 #include    <string.h>
18 #include    <sys/stat.h>        /* for S_xxx file mode constants */
```

```
19 #include    <unistd.h>
20 #include    <sys/wait.h>

21 #ifdef  HAVE_MQUEUE_H
22 #include    <mqueue.h>          /* Posix message queues */
23 #endif

24 #ifdef  HAVE_SEMAPHORE_H
25 #include    <semaphore.h>       /* Posix semaphores */

26 #ifndef SEM_FAILED
27 #define SEM_FAILED  ((sem_t *)(-1))
28 #endif

29 #endif

30 #ifdef  HAVE_SYS_MMAN_H
31 #include    <sys/mman.h>        /* Posix shared memory */
32 #endif

33 #ifndef MAP_FAILED
34 #define MAP_FAILED  ((void *)(-1))
35 #endif

36 #ifdef  HAVE_SYS_IPC_H
37 #include    <sys/ipc.h>         /* System V IPC */
38 #endif

39 #ifdef  HAVE_SYS_MSG_H
40 #include    <sys/msg.h>         /* System V message queues */
41 #endif

42 #ifdef  HAVE_SYS_SEM_H
43 #ifdef  __bsdi__
44 #undef  HAVE_SYS_SEM_H          /* hack: BSDI's semctl() prototype is wrong */
45 #else
46 #include    <sys/sem.h>         /* System V semaphores */
47 #endif

48 #ifndef HAVE_SEMUN_UNION
49 union semun {                   /* define union for semctl() */
50     int     val;
51     struct semid_ds *buf;
52     unsigned short *array;
53 };
54 #endif
55 #endif  /* HAVE_SYS_SEM_H */

56 #ifdef  HAVE_SYS_SHM_H
57 #include    <sys/shm.h>         /* System V shared memory */
58 #endif

59 #ifdef  HAVE_SYS_SELECT_H
60 #include    <sys/select.h>      /* for convenience */
61 #endif

62 #ifdef  HAVE_POLL_H
63 #include    <poll.h>            /* for convenience */
64 #endif
```

```
 65 #ifdef  HAVE_STROPTS_H
 66 #include    <stropts.h>         /* for convenience */
 67 #endif

 68 #ifdef  HAVE_STRINGS_H
 69 #include    <strings.h>         /* for convenience */
 70 #endif

 71 /* Next two headers are normally needed for socket/file ioctl's:
 72  * <sys/ioctl.h> and <sys/filio.h>.
 73  */
 74 #ifdef  HAVE_SYS_IOCTL_H
 75 #include    <sys/ioctl.h>
 76 #endif
 77 #ifdef  HAVE_SYS_FILIO_H
 78 #include    <sys/filio.h>
 79 #endif

 80 #ifdef  HAVE_PTHREAD_H
 81 #include    <pthread.h>
 82 #endif

 83 #ifdef  HAVE_DOOR_H
 84 #include    <door.h>            /* Solaris doors API */
 85 #endif

 86 #ifdef  HAVE_RPC_RPC_H
 87 #ifdef _PSX4_NSPACE_H_TS        /* Digital Unix 4.0b hack, hack, hack */
 88 #undef  SUCCESS
 89 #endif
 90 #include    <rpc/rpc.h>         /* Sun RPC */
 91 #endif

 92 /* Define bzero() as a macro if it's not in standard C library. */
 93 #ifndef HAVE_BZERO
 94 #define bzero(ptr,n)        memset(ptr, 0, n)
 95 #endif

 96 /* Posix.1g requires that an #include of <poll.h> define INFTIM, but many
 97    systems still define it in <sys/stropts.h>.  We don't want to include
 98    all the streams stuff if it's not needed, so we just define INFTIM here.
 99    This is the standard value, but there's no guarantee it is -1. */
100 #ifndef INFTIM
101 #define INFTIM          (-1)    /* infinite poll timeout */
102 #ifdef  HAVE_POLL_H
103 #define INFTIM_UNPH             /* tell unpxti.h we defined it */
104 #endif
105 #endif

106 /* Miscellaneous constants */
107 #ifndef PATH_MAX                /* should be in <limits.h> */
108 #define PATH_MAX    1024        /* max # of characters in a pathname */
109 #endif

110 #define MAX_PATH    1024
111 #define MAXLINE     4096        /* max text line length */
112 #define BUFFSIZE    8192        /* buffer size for reads and writes */
```

```
113 #define FILE_MODE   (S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH)
114                     /* default permissions for new files */
115 #define DIR_MODE    (FILE_MODE | S_IXUSR | S_IXGRP | S_IXOTH)
116                     /* default permissions for new directories */

117 #define SVMSG_MODE  (MSG_R | MSG_W | MSG_R>>3 | MSG_R>>6)
118                     /* default permissions for new SV message queues */
119 #define SVSEM_MODE  (SEM_R | SEM_A | SEM_R>>3 | SEM_R>>6)
120                     /* default permissions for new SV semaphores */
121 #define SVSHM_MODE  (SHM_R | SHM_W | SHM_R>>3 | SHM_R>>6)
122                     /* default permissions for new SV shared memory */

123 typedef void Sigfunc (int);     /* for signal handlers */

124 #ifdef  HAVE_SIGINFO_T_STRUCT
125 typedef void Sigfunc_rt (int, siginfo_t *, void *);
126 #endif

127 #define min(a,b)    ((a) < (b) ? (a) : (b))
128 #define max(a,b)    ((a) > (b) ? (a) : (b))

129 #ifndef HAVE_TIMESPEC_STRUCT
130 struct timespec {
131     time_t  tv_sec;             /* seconds */
132     long    tv_nsec;            /* and nanoseconds */
133 };
134 #endif

135 /*
136  * In our wrappers for open(), mq_open(), and sem_open() we handle the
137  * optional arguments using the va_XXX() macros.  But one of the optional
138  * arguments is of type "mode_t" and this breaks under BSD/OS because it
139  * uses a 16-bit integer for this datatype.  But when our wrapper function
140  * is called, the compiler expands the 16-bit short integer to a 32-bit
141  * integer.  This breaks our call to va_arg().  All we can do is the
142  * following hack.  Other systems in addition to BSD/OS might have this
143  * problem too ...
144  */

145 #ifdef  __bsdi__
146 #define va_mode_t   int
147 #else
148 #define va_mode_t   mode_t
149 #endif

150         /* our record locking macros */
151 #define read_lock(fd, offset, whence, len) \
152             lock_reg(fd, F_SETLK, F_RDLCK, offset, whence, len)
153 #define readw_lock(fd, offset, whence, len) \
154             lock_reg(fd, F_SETLKW, F_RDLCK, offset, whence, len)
155 #define write_lock(fd, offset, whence, len) \
156             lock_reg(fd, F_SETLK, F_WRLCK, offset, whence, len)
157 #define writew_lock(fd, offset, whence, len) \
158             lock_reg(fd, F_SETLKW, F_WRLCK, offset, whence, len)
159 #define un_lock(fd, offset, whence, len) \
160             lock_reg(fd, F_SETLK, F_UNLCK, offset, whence, len)
161 #define is_read_lockable(fd, offset, whence, len) \
```

```
162             lock_test(fd, F_RDLCK, offset, whence, len)
163 #define is_write_lockable(fd, offset, whence, len) \
164             lock_test(fd, F_WRLCK, offset, whence, len)
```
lib/unpipc.h

Figure C.1 Our header `unpipc.h`.

C.2 `config.h` Header

The GNU `autoconf` tool was used to aid in the portability of all the source code in this text. It is available from `ftp://prep.ai.mit.edu/pub/gnu/`. This tool generates a shell script named `configure` that you must run after downloading the software onto your system. This script determines the features provided by your Unix system: are System V message queues supported? is the `uint8_t` datatype defined? is the `gethostname` function provided? and so on, generating a header named `config.h`. This header is the first header included by our `unpipc.h` header in the previous section. Figure C.2 shows the `config.h` header for Solaris 2.6 when used with the `gcc` compiler.

The lines beginning with `#define` in column 1 are for features that the system provides. The lines that are commented out and contain `#undef` are features that the system does not provide.

sparc-sun-solaris2.6/config.h

```
 1 /* config.h.  Generated automatically by configure.  */
 2 /* Define the following if you have the corresponding header */
 3 #define CPU_VENDOR_OS "sparc-sun-solaris2.6"
 4 #define HAVE_DOOR_H 1                 /* <door.h> */
 5 #define HAVE_MQUEUE_H 1               /* <mqueue.h> */
 6 #define HAVE_POLL_H 1                 /* <poll.h> */
 7 #define HAVE_PTHREAD_H 1              /* <pthread.h> */
 8 #define HAVE_RPC_RPC_H 1              /* <rpc/rpc.h> */
 9 #define HAVE_SEMAPHORE_H 1            /* <semaphore.h> */
10 #define HAVE_STRINGS_H 1              /* <strings.h> */
11 #define HAVE_SYS_FILIO_H 1            /* <sys/filio.h> */
12 #define HAVE_SYS_IOCTL_H 1            /* <sys/ioctl.h> */
13 #define HAVE_SYS_IPC_H 1              /* <sys/ipc.h> */
14 #define HAVE_SYS_MMAN_H 1             /* <sys/mman.h> */
15 #define HAVE_SYS_MSG_H 1              /* <sys/msg.h> */
16 #define HAVE_SYS_SEM_H 1              /* <sys/sem.h> */
17 #define HAVE_SYS_SHM_H 1              /* <sys/shm.h> */
18 #define HAVE_SYS_SELECT_H 1           /* <sys/select.h> */
19 /* #undef   HAVE_SYS_SYSCTL_H */     /* <sys/sysctl.h> */
20 #define HAVE_SYS_TIME_H 1             /* <sys/time.h> */
21 /* Define if we can include <time.h> with <sys/time.h> */
22 #define TIME_WITH_SYS_TIME 1
23 /* Define the following if the function is provided */
24 #define HAVE_BZERO 1
25 #define HAVE_FATTACH 1
26 #define HAVE_POLL 1
```

```
27 /* #undef   HAVE_PSELECT */
28 #define HAVE_SIGWAIT 1
29 #define HAVE_VALLOC 1
30 #define HAVE_VSNPRINTF 1

31 /* Define the following if the function prototype is in a header */
32 #define HAVE_GETHOSTNAME_PROTO 1     /* <unistd.h> */
33 #define HAVE_GETRUSAGE_PROTO 1       /* <sys/resource.h> */
34 /* #undef   HAVE_PSELECT_PROTO */    /* <sys/select.h> */
35 #define HAVE_SHM_OPEN_PROTO 1        /* <sys/mman.h> */
36 #define HAVE_SNPRINTF_PROTO 1        /* <stdio.h> */
37 #define HAVE_THR_SETCONCURRENCY_PROTO 1     /* <thread.h> */

38 /* Define the following if the structure is defined. */
39 #define HAVE_SIGINFO_T_STRUCT 1      /* <signal.h> */
40 #define HAVE_TIMESPEC_STRUCT 1       /* <time.h> */
41 /* #undef   HAVE_SEMUN_UNION */      /* <sys/sem.h> */

42 /* Devices */
43 #define HAVE_DEV_ZERO 1

44 /* Define the following to the appropriate datatype, if necessary */
45 /* #undef   int8_t */                /* <sys/types.h> */
46 /* #undef   int16_t */               /* <sys/types.h> */
47 /* #undef   int32_t */               /* <sys/types.h> */
48 /* #undef   uint8_t */               /* <sys/types.h> */
49 /* #undef   uint16_t */              /* <sys/types.h> */
50 /* #undef   uint32_t */              /* <sys/types.h> */
51 /* #undef   size_t */                /* <sys/types.h> */
52 /* #undef   ssize_t */               /* <sys/types.h> */

53 #define POSIX_IPC_PREFIX "/"
54 #define RPCGEN_ANSIC 1          /* defined if rpcgen groks -C option */
```

sparc-sun-solaris2.6/config.h

Figure C.2 Our `config.h` header for Solaris 2.6.

C.3 Standard Error Functions

We define our own set of error functions that are used throughout the text to handle error conditions. The reason for our own error functions is to let us write our error handling with a single line of C code, as in

```
if (error condition)
    err_sys(printf format with any number of arguments);
```

instead of

```
if (error condition) {
    char   buff[200];
    snprintf(buff, sizeof(buff), printf format with any number of arguments);
    perror(buff);
    exit(1);
}
```

Our error functions use the variable-length argument list facility from ANSI C. See Section 7.3 of [Kernighan and Ritchie 1988] for additional details.

Figure C.3 lists the differences between the various error functions. If the global integer `daemon_proc` is nonzero, the message is passed to `syslog` with the indicated level (see Chapter 12 of UNPv1 for details on `syslog`); otherwise, the error is output to standard error.

Function	strerror (errno) ?	Terminate ?	syslog level
err_dump	yes	abort();	LOG_ERR
err_msg	no	return;	LOG_INFO
err_quit	no	exit(1);	LOG_ERR
err_ret	yes	return;	LOG_INFO
err_sys	yes	exit(1);	LOG_ERR

Figure C.3 Summary of our standard error functions.

Figure C.4 shows the five functions from Figure C.3.

lib/error.c

```
 1 #include    "unpipc.h"
 2 #include    <stdarg.h>          /* ANSI C header file */
 3 #include    <syslog.h>          /* for syslog() */
 4 int     daemon_proc;           /* set nonzero by daemon_init() */
 5 static void err_doit(int, int, const char *, va_list);
 6 /* Nonfatal error related to a system call.
 7  * Print a message and return. */
 8 void
 9 err_ret(const char *fmt,...)
10 {
11     va_list ap;
12     va_start(ap, fmt);
13     err_doit(1, LOG_INFO, fmt, ap);
14     va_end(ap);
15     return;
16 }
17 /* Fatal error related to a system call.
18  * Print a message and terminate. */
19 void
20 err_sys(const char *fmt,...)
21 {
22     va_list ap;
23     va_start(ap, fmt);
24     err_doit(1, LOG_ERR, fmt, ap);
25     va_end(ap);
26     exit(1);
27 }
```

```
28 /* Fatal error related to a system call.
29  * Print a message, dump core, and terminate. */
30 void
31 err_dump(const char *fmt,...)
32 {
33     va_list ap;

34     va_start(ap, fmt);
35     err_doit(1, LOG_ERR, fmt, ap);
36     va_end(ap);
37     abort();                    /* dump core and terminate */
38     exit(1);                    /* shouldn't get here */
39 }
40 /* Nonfatal error unrelated to a system call.
41  * Print a message and return. */
42 void
43 err_msg(const char *fmt,...)
44 {
45     va_list ap;

46     va_start(ap, fmt);
47     err_doit(0, LOG_INFO, fmt, ap);
48     va_end(ap);
49     return;
50 }
51 /* Fatal error unrelated to a system call.
52  * Print a message and terminate. */
53 void
54 err_quit(const char *fmt,...)
55 {
56     va_list ap;

57     va_start(ap, fmt);
58     err_doit(0, LOG_ERR, fmt, ap);
59     va_end(ap);
60     exit(1);
61 }
62 /* Print a message and return to caller.
63  * Caller specifies "errnoflag" and "level". */
64 static void
65 err_doit(int errnoflag, int level, const char *fmt, va_list ap)
66 {
67     int     errno_save, n;
68     char    buf[MAXLINE + 1];

69     errno_save = errno;         /* value caller might want printed */
70 #ifdef  HAVE_VSNPRINTF
71     vsnprintf(buf, MAXLINE, fmt, ap);   /* this is safe */
72 #else
73     vsprintf(buf, fmt, ap);     /* this is not safe */
74 #endif
75     n = strlen(buf);
```

```
76      if (errnoflag)
77          snprintf(buf + n, MAXLINE - n, ": %s", strerror(errno_save));
78      strcat(buf, "\n");
79      if (daemon_proc) {
80          syslog(level, buf);
81      } else {
82          fflush(stdout);         /* in case stdout and stderr are the same */
83          fputs(buf, stderr);
84          fflush(stderr);
85      }
86      return;
87  }
```

lib/error.c

Figure C.4 Our standard error functions.

Appendix D

Solutions to Selected Exercises

Chapter 1

1.1 Both processes only need to specify the `O_APPEND` flag to the `open` function, or the append mode to the `fopen` function. The kernel then ensures that each `write` is appended to the file. This is the easiest form of file synchronization to specify. (Pages 60–61 of APUE talk about this in more detail.) The synchronization issues become more complex when existing data in the file is updated, as in a database system.

1.2 Something like the following is typical:

```
#ifdef  _REENTRANT
#define errno   (*_errno())
#else
extern int  errno;
#endif
```

If `_REENTRANT` is defined, references to `errno` call a function named `_errno` that returns the address of the calling thread's `errno` variable. This variable is possibly stored as thread-specific data (Section 23.5 of UNPv1). If `_REENTRANT` is not defined, then `errno` is a global `int`.

Chapter 2

2.1 These two bits can change the effective user ID and/or the effective group ID of the program that is running. These two effective IDs are used in Section 2.4.

2.2 First specify both O_CREAT and O_EXCL, and if this returns success, a new object has been created. But if this fails with an error of EEXIST, then the object already exists and the program must call the open function again, without specifying either O_CREAT or O_EXCL. This second call should succeed, but a chance exists (albeit small) that it fails with an error of ENOENT, which indicates that some other thread or process has removed the object between the two calls.

Chapter 3

3.1 Our program is shown in Figure D.1.

svmsg/slotseq.c

```
1 #include    "unpipc.h"

2 int
3 main(int argc, char **argv)
4 {
5     int     i, msqid;
6     struct msqid_ds info;

7     for (i = 0; i < 10; i++) {
8         msqid = Msgget(IPC_PRIVATE, SVMSG_MODE | IPC_CREAT);
9         Msgctl(msqid, IPC_STAT, &info);
10        printf("msqid = %d, seq = %lu\n", msqid, info.msg_perm.seq);

11        Msgctl(msqid, IPC_RMID, NULL);
12    }
13    exit(0);
14 }
```

svmsg/slotseq.c

Figure D.1 Print identifier and slot usage sequence number.

3.2 The first call to msgget uses the first available message queue, whose slot usage sequence number is 20 after running the program in Figure 3.7 two times, returning an identifier of 1000. Assuming the next available message queue has never been used, its slot usage sequence number will be 0, returning an identifier of 1.

3.3 Our simple program is shown in Figure D.2.

svmsg/testumask.c

```
1 #include    "unpipc.h"

2 int
3 main(int argc, char **argv)
4 {
5     Msgget(IPC_PRIVATE, 0666 | IPC_CREAT | IPC_EXCL);
6     unlink("/tmp/fifo.1");
7     Mkfifo("/tmp/fifo.1", 0666);

8     exit(0);
9 }
```

svmsg/testumask.c

Figure D.2 Test whether the file mode creation mask is used by msgget.

When we run this program we see that our file mode creation mask is 2 (turn off the other-write bit) and this bit is turned off in the FIFO, but this bit is not turned off in the message queue.

```
solaris % umask
02
solaris % testumask
solaris % ls -l /tmp/fifo.1
prw-rw-r--   1 rstevens other1       0 Mar 25 16:05 /tmp/fifo.1
solaris % ipcs -q
IPC status from <running system> as of Wed Mar 25 16:06:03 1998
T         ID      KEY         MODE         OWNER    GROUP
Message Queues:
q        200   00000000 --rw-rw-rw- rstevens   other1
```

3.4 With `ftok`, the possibility always exists that some other pathname on the system can lead to the same key as the one being used by our server. With `IPC_PRIVATE`, the server knows that it is creating a new message queue, but the server must then write the resulting identifier into some file for the clients to read.

3.5 Here is one way to detect the collisions:

```
solaris % find / -links 1 -not -type l -print |
xargs -n1 ftok1 > temp.1
solaris % wc -l temp.1
  109351 temp.1

solaris % sort +0 -1 temp.1 |
nawk '{ if (lastkey == $1)
            print lastline, $0
        lastline = $0
        lastkey = $1
}' > temp.2
solaris % wc -l temp.2
   82188 temp.2
```

In the `find` program, we ignore files with more than one link (since each link will have the same i-node), and we ignore symbolic links (since the `stat` function follows the link). The extremely high percentage of collisions (75.2%) is due to Solaris 2.x using only 12 bits of the i-node number. This means lots of collisions can occur on any filesystem with more than 4096 files. For example, the four files with i-node numbers 4096, 8192, 12288, and 16384 all have the same IPC key (assuming they are on the same filesystem).

This example was run on the same filesystems but using the `ftok` function from BSD/OS, which adds the entire i-node number into the key, and the number of collisions was only 849 (less than 1%).

Chapter 4

4.1 If `fd[1]` were left open in the child when the parent terminated, the child's `read` of `fd[1]` would not return an end-of-file, because this descriptor is still open in

the child. By closing `fd[1]` in the child, this guarantees that as soon as the parent terminates, all its descriptors are closed, causing the child's `read` of `fd[1]` to return 0.

4.2 If the order of the calls is swapped, some other process can create the FIFO between the calls to `open` and `mkfifo`, causing the latter to fail.

4.3 If we execute

```
solaris % mainpopen 2>temp.stderr
/etc/ntp.conf > /myfile
solaris % cat temp.stderr
sh: /myfile: cannot create
```

we see that `popen` returns success, but we read just an end-of-file with `fgets`. The shell error message is written to standard error.

4.5 Change the first call to `open` to specify the nonblocking flag:

```
readfifo = Open(SERV_FIFO, O_RDONLY | O_NONBLOCK, 0);
```

This call then returns immediately, and the next call to `open` (for write-only) also returns immediately, since the FIFO is already open for reading. But to avoid an error from `readline`, the `O_NONBLOCK` flag must be turned off for the descriptor `readfifo` before calling `readline`.

4.6 If the client were to `open` its client-specific FIFO (read-only) before opening the server's well-known FIFO (write-only), a deadlock would occur. The only way to avoid the deadlock is to `open` the two FIFOs in the order shown in Figure 4.24 or to use the nonblocking flag.

4.7 The disappearance of the writer is signaled by an end-of-file for the reader.

4.8 Figure D.3 shows our program.

pipe/test1.c

```
 1 #include    "unpipc.h"

 2 int
 3 main(int argc, char **argv)
 4 {
 5     int     fd[2];
 6     char    buff[7];
 7     struct stat info;

 8     if (argc != 2)
 9         err_quit("usage: test1 <pathname>");

10     Mkfifo(argv[1], FILE_MODE);
11     fd[0] = Open(argv[1], O_RDONLY | O_NONBLOCK);
12     fd[1] = Open(argv[1], O_WRONLY | O_NONBLOCK);

13         /* check sizes when FIFO is empty */
14     Fstat(fd[0], &info);
15     printf("fd[0]: st_size = %ld\n", (long) info.st_size);
16     Fstat(fd[1], &info);
17     printf("fd[1]: st_size = %ld\n", (long) info.st_size);
```

```
18      Write(fd[1], buff, sizeof(buff));

19          /* check sizes when FIFO contains 7 bytes */
20      Fstat(fd[0], &info);
21      printf("fd[0]: st_size = %ld\n", (long) info.st_size);
22      Fstat(fd[1], &info);
23      printf("fd[1]: st_size = %ld\n", (long) info.st_size);

24      exit(0);
25  }
```

pipe/test1.c

Figure D.3 Determine whether `fstat` returns the number of bytes in a FIFO.

4.9 `select` returns that the descriptor is writable, but the call to `write` then elicits `SIGPIPE`. This concept is described on pages 153–155 of UNPv1; when a read (or write) error occurs, `select` returns that the descriptor is readable (or writable), and the actual error is returned by `read` (or `write`). Figure D.4 shows our program.

pipe/test2.c

```
1  #include    "unpipc.h"

2  int
3  main(int argc, char **argv)
4  {
5      int     fd[2], n;
6      pid_t   childpid;
7      fd_set  wset;

8      Pipe(fd);
9      if ( (childpid = Fork()) == 0) {      /* child */
10         printf("child closing pipe read descriptor\n");
11         Close(fd[0]);
12         sleep(6);
13         exit(0);
14     }
15         /* parent */
16     Close(fd[0]);                /* in case of a full-duplex pipe */
17     sleep(3);
18     FD_ZERO(&wset);
19     FD_SET(fd[1], &wset);
20     n = select(fd[1] + 1, NULL, &wset, NULL, NULL);
21     printf("select returned %d\n", n);

22     if (FD_ISSET(fd[1], &wset)) {
23         printf("fd[1] writable\n");
24         Write(fd[1], "hello", 5);
25     }
26     exit(0);
27 }
```

pipe/test2.c

Figure D.4 Determine what `select` returns for writability when the read end of a pipe is closed.

Chapter 5

5.1 First create the queue without specifying any attributes, followed by a call to `mq_getattr` to obtain the default attributes. Then remove the queue and create it again, using the default value of either attribute that is not specified.

5.2 The signal is not generated for the second message, because the registration is removed every time the notification occurs.

5.3 The signal is not generated for the second message, because the queue was not empty when the message was received.

5.4 The GNU C compiler under Solaris 2.6 (which defines both constants as calls to `sysconf`) generates the errors

```
test1.c:13: warning: int format, long int arg (arg 2)
test1.c:13: warning: int format, long int arg (arg 3)
```

5.5 Under Solaris 2.6, we specify 1,000,000 messages of 10 bytes each. This leads to a file size of 20,000,536 bytes, which corresponds with our results from running Figure 5.5: 10 bytes of data per message, 8 bytes of overhead per message (perhaps for pointers), another 2 bytes of overhead per message (perhaps for 4-byte alignment), and 536 bytes of overhead per file. Before `mq_open` is called, the size of the program reported by `ps` is 1052 Kbytes, but after the message queue is created, the size is 20 Mbytes. This makes us think that Posix message queues are implemented using memory-mapped files, and that `mq_open` maps the file into the address space of the calling process. We obtain similar results under Digital Unix 4.0B.

5.6 A size argument of 0 is OK for the ANSI C `memXXX` functions. The original 1989 ANSI C standard X3.159−1989, also known as ISO/IEC 9899:1990, did not say this (and none of the manual pages that the author could find mentioned this), but *Technical Corrigendum Number 1* explicitly states that a size of 0 is OK (but the pointer arguments must still be valid). `http://www.lysator.liu.se/c/` is a wonderful reference point for information on the C language.

5.7 For two-way communication between two processes, two message queues are needed (see for example, Figure A.30). Indeed, if we were to modify Figure 4.14 to use Posix message queues instead of pipes, we would see the parent read back what it wrote to the queue.

5.8 The mutex and condition variable are contained in the memory-mapped file, which is shared by all processes that have the queue open. Other processes may have the queue open, so a process that is closing its handle to the queue cannot destroy the mutex and condition variable.

5.9 An array cannot be assigned across an equals sign in C, whereas a structure can.

5.10 The `main` function spends almost all of its time blocked in a call to `select`, waiting for the pipe to be readable. Every time the signal is delivered, the return from the signal handler interrupts this call to `select`, causing it to return an error of

EINTR. To handle this, our Select wrapper function checks for this error, and calls select again, as shown in Figure D.5.

lib/wrapunix.c

```
313 int
314 Select(int nfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds,
315        struct timeval *timeout)
316 {
317     int     n;

318   again:
319     if ( (n = select(nfds, readfds, writefds, exceptfds, timeout)) < 0) {
320         if (errno == EINTR)
321             goto again;
322         else
323             err_sys("select error");
324     } else if (n == 0 && timeout == NULL)
325         err_quit("select returned 0 with no timeout");
326     return (n);                 /* can return 0 on timeout */
327 }
```

lib/wrapunix.c

Figure D.5 Our Select wrapper function that handles EINTR.

Page 124 of UNPv1 talks more about interrupted system calls.

Chapter 6

6.1 The remaining programs must then accept a numeric message queue identifier instead of a pathname (recall the output of Figure 6.3). This change could be made with a new command-line option in these other programs, or the assumption could be made that a pathname argument that is entirely numeric is an identifier and not a pathname. Since most pathnames that are passed to ftok are absolute pathnames, and not relative (i.e., they contain at least one slash character), this assumption is probably OK.

6.2 Messages with a type of 0 are not allowed, and a client can never have a process ID of 1, since this is normally the init process.

6.3 When only one queue is used in Figure 6.14, this malicious client affects all other clients. When we have one return queue per client (Figure 6.19), this client affects only its own queue.

Chapter 7

7.2 The process will terminate, probably before the consumer thread has finished, because calling exit terminates any threads still running.

7.3 Under Solaris 2.6, omitting the call to the destroy functions causes a memory leak, implying that the init functions are performing dynamic memory allocation. We do not see this under Digital Unix 4.0B, which just implies an implementation difference. The calls to the matching destroy functions are still required.

From an implementation perspective, Digital Unix appears to use the `attr_t` variable as the attributes object itself, whereas Solaris uses this variable as a pointer to a dynamically allocated object. Either implementation is fine.

Chapter 9

9.1 Depending on your system, you may need to increase the loop counter from 20, to see the errors.

9.2 To make the standard I/O stream unbuffered, we add the line

```
setvbuf(stdout, NULL, _IONBF, 0);
```

to the `main` function, before the `for` loop. This should have no effect, because there is only one call to `printf` and the string is terminated with a newline. Normally, standard output is line buffered, so in either case (line buffered or unbuffered), the single call to `printf` ends up in a single `write` call to the kernel.

9.3 We change the call to `printf` to be

```
snprintf(line, sizeof(line), "%s: pid = %ld, seq# = %d\n",
        argv[0], (long) pid, seqno);
for (ptr = line; (c = *ptr++) != 0; )
    putchar(c);
```

and declare `c` as an integer and `ptr` as a `char*`. If we leave in the call to `setvbuf`, making standard output unbuffered, this causes the standard I/O library to call `write` once per character that is output, instead of once per line. This involves more CPU time, and provides more opportunities for the kernel to switch between the two processes. We should see more errors with this program.

9.4 Since multiple processes are allowed to have read locks for the same region of a file, this is the same as having no locks at all for our example.

9.5 Nothing changes, because the nonblocking flag for a descriptor has no effect on `fcntl` advisory locking. What determines whether a call to `fcntl` blocks or not is whether the command is `F_SETLKW` (which always blocks) or `F_SETLK` (which never blocks).

9.6 The `loopfcntlnonb` program operates as expected, because, as we showed in the previous exercise, the nonblocking flag has no effect on a program that performs `fcntl` locking. But the nonblocking flag does affect the `loopnonenonb` program, which performs no locking. As we said in Section 9.5, a nonblocking call to `read` or `write` for a file for which mandatory locking is enabled, returns an error of `EAGAIN` if the `read` or `write` conflicts with an existing lock. We see this error as either

```
read error: Resource temporarily unavailable
```

or

```
write error: Resource temporarily unavailable
```

and we can verify that the error is `EAGAIN` by executing

```
solaris % grep Resource /usr/include/sys/errno.h
#define EAGAIN  11      /* Resource temporarily unavailable */
```

9.7 Under Solaris 2.6, mandatory locking increases the clock time by about 16% and it increases the system CPU time by about 20%. The user CPU time remains the same, as we expect, because the extra time is within the kernel checking every `read` and `write`, not within our process.

9.8 Locks are granted on a per-process basis, not on a per-thread basis. To see contention for lock requests, we must have different processes trying to obtain the locks.

9.9 If another copy of the daemon were running and we `open` with the `O_TRUNC` flag, this would wipe out the process ID stored by the first copy of the daemon. We cannot truncate the file until we know we are the only copy running.

9.10 `SEEK_SET` is always preferable. The problem with `SEEK_CUR` is that it depends on the current offset in the file, which is specified by `lseek`. But if we call `lseek` and then `fcntl`, we are using two function calls to perform what is a single operation, and a chance exists that another thread can change the current offset by calling `lseek` between our two function calls. (Recall that all threads share the same descriptors. Also recall that `fcntl` record locks are for locking between different processes and not for locking between the different threads within one process.) Similarly, if we specify `SEEK_END`, a chance exists that another thread can append data to the file before we obtain a lock based on what we think is the end of the file.

Chapter 10

10.1 Here is the output under Solaris 2.6:

```
solaris % deadlock 100
prod: calling sem_wait(nempty)          i=0 loop for producer
prod: got sem_wait(nempty)
prod: calling sem_wait(mutex)
prod: got sem_wait(mutex), storing 0

prod: calling sem_wait(nempty)          i=1 loop for producer
prod: got sem_wait(nempty)
prod: calling sem_wait(mutex)
prod: got sem_wait(mutex), storing 1

prod: calling sem_wait(nempty)          start next loop, but no empty slots
                                        context switch from producer to consumer
cons: calling sem_wait(mutex)           i=0 loop for consumer
cons: got sem_wait(mutex)
cons: calling sem_wait(nstored)
cons: got sem_wait(nstored)
cons: fetched 0

cons: calling sem_wait(mutex)           i=0 loop for consumer
cons: got sem_wait(mutex)
cons: calling sem_wait(nstored)
```

```
cons: got sem_wait(nstored)
cons: fetched 1

cons: calling sem_wait(mutex)
cons: got sem_wait(mutex)
cons: calling sem_wait(nstored)        consumer blocks here forever
                                       context switch from consumer to producer
prod: got sem_wait(nempty)
prod: calling sem_wait(mutex)          producer blocks here forever
```

10.2 This is OK given the rules for semaphore initialization that we specified when we described `sem_open`: if the semaphore already exists, it is not initialized. So only the first of the four programs that calls `sem_open` actually initializes the semaphore value to 1. When the remaining three call `sem_open` with the `O_CREAT` flag, the semaphore will already exist, so its value is not initialized again.

10.3 This is a problem. The semaphore is automatically closed when the process terminates, but the value of the semaphore is not changed. This will prevent any of the other three programs from obtaining the lock, causing another type of deadlock.

10.4 If we did not initialize the descriptors to –1, their initial value is unknown, since `malloc` does not initialize the memory that it allocates. So if one of the calls to `open` fails, the calls to `close` at the label `error` could close some descriptor that the process is using. By initializing the descriptors to –1, we know that the calls to `close` will have no effect (other than returning an error that we ignore) if that descriptor has not been opened yet.

10.5 A chance exists, albeit slight, that `close` could be called for a valid descriptor and could return some error, thereby changing `errno` from the value that we want to return. Since we want to save the value of `errno` to return to the caller, to do so explicitly is better than counting on some side effect (that `close` will not return an error when a valid descriptor is closed).

10.6 No race condition exists in this function, because the `mkfifo` function returns an error if the FIFO already exists. If two processes call this function at about the same time, the FIFO is created only once. The second process to call `mkfifo` will receive an error of `EEXIST`, causing the `O_CREAT` flag to be turned off, preventing another initialization of the FIFO.

10.7 Figure 10.37 does not have the race condition that we described with Figure 10.43 because the initialization of the semaphore is performed by writing data to the FIFO. If the process that creates the FIFO is suspended by the kernel after it calls `mkfifo` but before it `writes` the data bytes to the FIFO, the second process will just open the FIFO and block the first time it calls `sem_wait`, because the newly created FIFO will be empty until the first process (which created the FIFO) writes the data bytes to the FIFO.

10.8 Figure D.6 shows the test program. Both the Solaris 2.6 and Digital Unix 4.0B implementations detect being interrupted by a caught signal and return `EINTR`.

pxsem/testeintr.c

```
1 #include    "unpipc.h"
2 #define NAME    "testeintr"
3 static void sig_alrm(int);
4 int
5 main(int argc, char **argv)
6 {
7     sem_t  *sem1, sem2;
8         /* first test a named semaphore */
9     sem_unlink(Px_ipc_name(NAME));
10    sem1 = Sem_open(Px_ipc_name(NAME), O_RDWR | O_CREAT | O_EXCL,
11                    FILE_MODE, 0);
12    Signal(SIGALRM, sig_alrm);
13    alarm(2);
14    if (sem_wait(sem1) == 0)
15        printf("sem_wait returned 0?\n");
16    else
17        err_ret("sem_wait error");
18    Sem_close(sem1);
19        /* now a memory-based semaphore with process scope */
20    Sem_init(&sem2, 1, 0);
21    alarm(2);
22    if (sem_wait(&sem2) == 0)
23        printf("sem_wait returned 0?\n");
24    else
25        err_ret("sem_wait error");
26    Sem_destroy(&sem2);
27    exit(0);
28 }
29 static void
30 sig_alrm(int signo)
31 {
32    printf("SIGALRM caught\n");
33    return;
34 }
```

pxsem/testeintr.c

Figure D.6 Test whether `sem_wait` detects `EINTR`.

Our implementation using FIFOs returns `EINTR`, because `sem_wait` blocks in a call to `read` on a FIFO, which must return the error. Our implementation using memory-mapped I/O does not return any error, because `sem_wait` blocks in a call to `pthread_cond_wait` and this function does not return `EINTR` when interrupted by a caught signal. (We saw another example of this with Figure 5.29.) Our implementation using System V semaphores returns `EINTR`, because `sem_wait` blocks in a call to `semop`, which returns the error.

10.9 The implementation using FIFOs (Figure 10.40) is async-signal-safe because `write` is async-signal-safe. The implementation using a memory-mapped file

(Figure 10.47) is not, because none of the pthread_XXX functions are async-signal-safe. The implementation using System V semaphores (Figure 10.56) is not, because semop is not listed as async-signal-safe by Unix 98.

Chapter 11

11.1 Only one line needs to change:

```
<       semid = Semget(Ftok(argv[optind], 0), 0, 0);
---
>       semid = atol(argv[optind]);
```

11.2 The call to ftok will fail, causing our Ftok wrapper to terminate. The my_lock function could call ftok before calling semget, check for an error of ENOENT, and create the file if it does not exist.

Chapter 12

12.1 The file size would be increased by another 4096 bytes (to 36864), but our reference to the new end-of-file (index 36863) might generate a SIGSEGV signal, since the size of the memory-mapped region is 32768. The reason we say "might" and not "will" is that it depends on the page size.

12.2 Figure D.7 shows the scenario assuming a System V message queue, and Figure D.8 shows the Posix message queue scenario. The calls to memcpy in the sender occur when mq_send is called (Figure 5.30), and the calls to memcpy in the receiver occur when mq_receive is called (Figure 5.32).

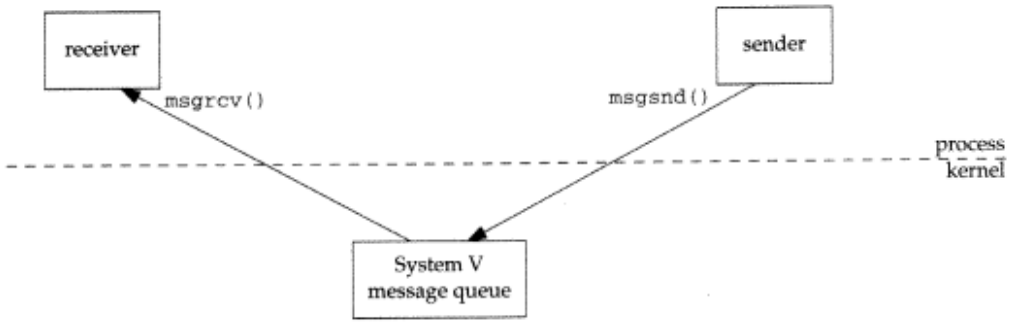

Figure D.7 Sending messages using a System V message queue.

12.3 Any read from /dev/zero returns the requested number of bytes, all containing 0. Any data written to this device is simply discarded, just like writes to /dev/null.

12.4 The final contents of the file are 4 bytes of 0 (assuming a 32-bit int).

12.5 Figure D.9 shows our program.

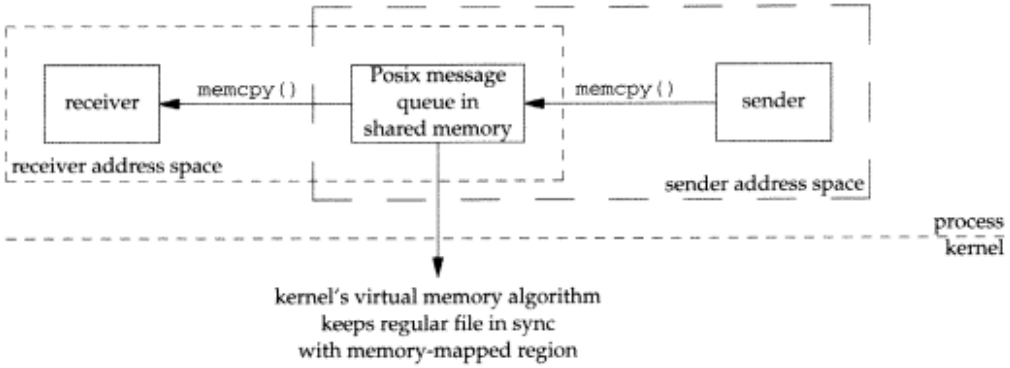

Figure D.8 Sending messages using a Posix message queue implemented using `mmap`.

shm/svmsgread.c

```
1 #include    "unpipc.h"
2 #define MAXMSG  (8192 + sizeof(long))
3 int
4 main(int argc, char **argv)
5 {
6     int     pipe1[2], pipe2[2], mqid;
7     char    c;
8     pid_t   childpid;
9     fd_set  rset;
10    ssize_t n, nread;
11    struct msgbuf *buff;
12    if (argc != 2)
13        err_quit("usage: svmsgread <pathname>");
14    Pipe(pipe1);                /* 2-way communication with child */
15    Pipe(pipe2);
16    buff = My_shm(MAXMSG);      /* anonymous shared memory with child */
17    if ( (childpid = Fork()) == 0) {
18        Close(pipe1[1]);        /* child */
19        Close(pipe2[0]);
20        mqid = Msgget(Ftok(argv[1], 1), MSG_R);
21        for ( ; ; ) {
22                /* block, waiting for message, then tell parent */
23            nread = Msgrcv(mqid, buff, MAXMSG, 0, 0);
24            Write(pipe2[1], &nread, sizeof(ssize_t));
25                /* wait for parent to say shm is available */
26            if ( (n = Read(pipe1[0], &c, 1)) != 1)
27                err_quit("child: read on pipe returned %d", n);
28        }
29        exit(0);
30    }
```

```
31          /* parent */
32      Close(pipe1[0]);
33      Close(pipe2[1]);

34      FD_ZERO(&rset);
35      FD_SET(pipe2[0], &rset);
36      for ( ; ; ) {
37          if ( (n = select(pipe2[0] + 1, &rset, NULL, NULL, NULL)) != 1)
38              err_sys("select returned %d", n);
39          if (FD_ISSET(pipe2[0], &rset)) {
40              n = Read(pipe2[0], &nread, sizeof(ssize_t));
41              if (n != sizeof(ssize_t))
42                  err_quit("parent: read on pipe returned %d", n);

43              printf("read %d bytes, type = %ld\n", nread, buff->mtype);
44              Write(pipe1[1], &c, 1);

45          } else
46              err_quit("pipe2[0] not ready");
47      }

48      Kill(childpid, SIGTERM);
49      exit(0);
50  }
```

shm/svmsgread.c

Figure D.9 Example of parent and child setup to use `select` with System V messages.

Chapter 13

13.1 Figure D.10 shows our modified version of Figure 12.16, and Figure D.11 shows our modified version of Figure 12.19. Notice in the first program that we must set the size of the shared memory object using `ftruncate`; we cannot use `lseek` and `write`.

pxshm/test1.c

```
 1 #include    "unpipc.h"

 2 int
 3 main(int argc, char **argv)
 4 {
 5      int     fd, i;
 6      char   *ptr;
 7      size_t  shmsize, mmapsize, pagesize;

 8      if (argc != 4)
 9          err_quit("usage: test1 <name> <shmsize> <mmapsize>");
10      shmsize = atoi(argv[2]);
11      mmapsize = atoi(argv[3]);

12      /* open shm: create or truncate; set shm size */
13      fd = Shm_open(Px_ipc_name(argv[1]), O_RDWR | O_CREAT | O_TRUNC,
14                    FILE_MODE);
15      Ftruncate(fd, shmsize);
```

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

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

20      for (i = 0; i < max(shmsize, mmapsize); i += pagesize) {
21          printf("ptr[%d] = %d\n", i, ptr[i]);
22          ptr[i] = 1;
23          printf("ptr[%d] = %d\n", i + pagesize - 1, ptr[i + pagesize - 1]);
24          ptr[i + pagesize - 1] = 1;
25      }
26      printf("ptr[%d] = %d\n", i, ptr[i]);

27      exit(0);
28  }
```

pxshm/test1.c

Figure D.10 Memory mapping when `mmap` equals shared memory size.

pxshm/test1.c

```
 1  #include    "unpipc.h"

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

 4  int
 5  main(int argc, char **argv)
 6  {
 7      int     fd, i;
 8      char   *ptr;

 9          /* open shm: create or truncate; then mmap shm */
10      fd = Shm_open(Px_ipc_name(FILE), O_RDWR | O_CREAT | O_TRUNC, FILE_MODE);
11      ptr = Mmap(NULL, SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);

12      for (i = 4096; i <= SIZE; i += 4096) {
13          printf("setting shm size to %d\n", i);
14          Ftruncate(fd, i);
15          printf("ptr[%d] = %d\n", i - 1, ptr[i - 1]);
16      }

17      exit(0);
18  }
```

pxshm/test1.c

Figure D.11 Memory-map example that lets the shared memory size grow.

13.2 One possible problem with `*ptr++` is that the pointer returned by `mmap` is modified, preventing a later call to `munmap`. If the pointer is needed at a later time, it must be either saved, or not modified.

Chapter 14

14.1 Only one line needs to change:

```
13c13
<      id = Shmget(Ftok(argv[1], 0), 0, SVSHM_MODE);
---
>      id = atol(argv[1]);
```

Chapter 15

15.1 There are `data_size` + (`desc_num` × `sizeof(door_desc_t)`) bytes of arguments.

15.2 No, we do not need to call `fstat`. If the descriptor does not refer to a door, `door_info` returns an error of `EBADF`:

```
solaris % doorinfo /etc/passwd
door_info error: Bad file number
```

15.3 The manual page is wrong. Posix.1 states correctly that "The *sleep()* function shall cause the current thread to be suspended from execution."

15.4 The results are unpredictable (although a core dump is a pretty safe bet), because the address of the server procedure associated with the door will cause some random code in the newly `exec`ed program to be called as a function.

15.5 When the client's `door_call` is terminated by the caught signal, the server process must be notified because the server thread handling this client (thread ID 4 in our output) is then sent a cancellation request. But we said with Figure 15.23 that for all the server threads automatically created by the doors library, cancellation is disabled, and hence this thread is not terminated. Instead, the call to `sleep(6)`, in which the server procedure is blocked, appears to return prematurely when the client's `door_call` is terminated, about 2 seconds after the server procedure was called. But the server thread still proceeds to completion.

15.6 The error that we see is

```
solaris % server6 /tmp/door6
my_thread: created server thread 4
door_bind error: Bad file number
```

When starting the server 20 times in a row, the error occurred five times. This error is nondeterministic.

15.7 No. All that is required is to enable cancellation each time the server procedure is called, as we do in Figure 15.31. Although this technique calls the function `pthread_setcancelstate` every time the server procedure is invoked, instead of just once when the thread starts, this overhead is probably trivial.

15.8 To test this, we modify one of our servers (say Figure 15.9) to call `door_revoke` from the server procedure. Since the door descriptor is the argument to

door_revoke, we must also make fd a global. We then execute our client (say Figure 15.2) twice:

```
solaris % client8 /tmp/door8 88
result: 7744
solaris % client8 /tmp/door8 99
door_call error: Bad file number
```

The first invocation returns successfully, verifying our statement that door_revoke does not affect a call that is in progress. The second invocation tells us that the error from door_call is EBADF.

15.9 To avoid making fd a global, we use the cookie pointer that we can pass to door_create and that is then passed to the server procedure every time it is called. Figure D.12 shows the server process.

doors/server9.c

```
 1 #include    "unpipc.h"

 2 void
 3 servproc(void *cookie, char *dataptr, size_t datasize,
 4          door_desc_t *descptr, size_t ndesc)
 5 {
 6     long    arg, result;

 7     Door_revoke(*((int *) cookie));
 8     arg = *((long *) dataptr);
 9     printf("thread id %ld, arg = %ld\n", pr_thread_id(NULL), arg);

10     result = arg * arg;
11     Door_return((char *) &result, sizeof(result), NULL, 0);
12 }

13 int
14 main(int argc, char **argv)
15 {
16     int     fd;

17     if (argc != 2)
18         err_quit("usage: server9 <server-pathname>");

19         /* create a door descriptor and attach to pathname */
20     fd = Door_create(servproc, &fd, 0);

21     unlink(argv[1]);
22     Close(Open(argv[1], O_CREAT | O_RDWR, FILE_MODE));
23     Fattach(fd, argv[1]);

24         /* servproc() handles all client requests */
25     for ( ; ; )
26         pause();
27 }
```

doors/server9.c

Figure D.12 Using the cookie pointer to avoid making fd a global.

We could easily make the same change to Figures 15.22 and 15.23, since the cookie pointer is available to our my_thread function (in the door_info_t

structure), which passes a pointer to this structure to the newly created thread (which needs the descriptor for the call to `door_bind`).

15.10 In this example, the thread attributes never change, so we could initialize the attributes once (in the `main` function).

Chapter 16

16.1 The port mapper does not monitor the servers that register with it, to try and detect if they crash. After we terminate our client, the port mapper mappings remain in place, as we can verify with the `rpcinfo` program. So a client who contacts the port mapper after our server terminates will get an OK return from the port mapper with the port numbers in use before the server terminated. But when a client tries to contact the TCP server, the RPC runtime will receive an RST (reset) in response to its SYN (assuming that no other process has since been assigned that same port on the server host), causing an error return from `clnt_create`. A UDP client's call to `clnt_create` will succeed (since there is no connection to establish), but when the client sends a UDP datagram to the old server port, nothing will be returned (assuming again that no other process has since been assigned that same port on the server host) and the client's procedure call will eventually time out.

16.2 The RPC runtime returns the server's first reply to the client when it is received, about 20 seconds after the client's call. The next reply for the server will just be held in the client's network buffer for this endpoint until either the endpoint is closed, or until the next read of this buffer by the RPC runtime. Assume that the client issues a second call to this server immediately after receiving the first reply. Assuming no network loss, the next datagram that will arrive on this endpoint will be the server's reply to the client's retransmission. But the RPC runtime will ignore this reply, since the XID will correspond to the client's first procedure call, which cannot equal the XID used for this second procedure call.

16.3 The C structure member is `char c[10]`, but this will be encoded by XDR as ten 4-byte integers. If you really want a fixed-length string, use the fixed-length opaque datatype.

16.4 The call to `xdr_data` returns `FALSE`, because its call to `xdr_string` (look at the `data_xdr.c` file) returns `FALSE`.

When a maximum length is specified, it is coded as the final argument to `xdr_string`. When this maximum length is omitted, the final argument is the one's complement of 0, (which is $2^{32} - 1$, assuming 32-bit integers).

16.5 The XDR routines all check that adequate room is available in the buffer for the data that is being encoded into the buffer, and they return an error of `FALSE` when the buffer is full. Unfortunately, there is no way to distinguish among the different possible errors from the XDR functions.

16.6 We could say that TCP's use of sequence numbers to detect duplicate data is, in effect, a duplicate request cache, because these sequence numbers identify any

old segment that arrives as containing duplicate data that TCP has already acknowledged. For a given connection (e.g., for a given client's IP address and port), the size of this cache would be one-half of TCP's 32-bit sequence number space, or 2^{31}, about 2 gigabytes.

16.7 Since all five values for a given request must be equal to all five values in the cache entry, the first value compared should be the one most likely to be unequal, and the last value compared should be the one least likely to be unequal. The actual order of the comparisons in the TI-RPC package is (1) XID, (2) procedure number, (3) version number, (4) program number, and (5) client's address. Given that the XID changes for every request, to compare it first makes sense.

16.8 In Figure 16.30, starting with the flag/length field and including 4 bytes for the long integer argument, there are 12 4-byte fields, for a total of 48 bytes. With the default of null authentication, the credential data and verifier data will both be empty. That is, the credentials and verifier will both take 8 bytes: 4 bytes for the authentication flavor (`AUTH_NONE`) and 4 bytes for the authentication length (which has a value of 0).

In the reply (look at Figure 16.32 but realize that since TCP is being used, a 4-byte flag/length field will precede the XID), there are eight 4-byte fields, starting with the flag/length field and ending with 4 bytes of long integer result. They total 32 bytes.

When UDP is used, the only change in the request and reply is the absence of the 4-byte flag/length field. This gives a request size of 44 bytes and a reply size of 28 bytes, which we can verify with `tcpdump`.

16.9 Yes. The difference in argument handling, both at the client end and at the server end, is local to that host and independent of the packets that traverse the network. The client `main` calls a function in the client stub to generate a network record, and the server `main` calls a function in the server stub to process this network record. The RPC record that is transmitted across the network is defined by the RPC protocol, and this does not change, regardless of whether either end supports threads or not.

16.10 The XDR runtime dynamically allocates space for these strings. We verify this fact by adding the following line to our `read` program:

```
printf("sbrk() = %p, buff = %p, in.vstring_arg = %p\n",
       sbrk(NULL), buff, in.vstring_arg);
```

The `sbrk` function returns the current address at the top of the program's data segment, and the memory just below this is normally the region from which `malloc` takes its memory. Running this program yields

```
sbrk() = 29638, buff = 25e48, in.vstring_arg = 27e58
```

which shows that the pointer `vstring_arg` points into the region used by `malloc`. Our 8192-byte `buff` goes from `0x25e48` to `0x27e47`, and the string is stored just beyond this buffer.

16.11 Figure D.13 shows the client program. Note that the final argument to `clnt_call` is an actual `timeval` structure and not a pointer to one of these structures. Also note that the third and fifth arguments to `clnt_call` must be nonnull function pointers to XDR routines, so we specify `xdr_void`, the XDR function that does nothing. (You can verify that this is the way to call a function with no arguments or no return values, by writing a trivial RPC specification file that defines a function with no arguments and no return values, running `rpcgen`, and examining the client stub that is generated.)

sunrpc/square10/client.c

```
 1 #include    "unpipc.h"          /* our header */
 2 #include    "square.h"          /* generated by rpcgen */

 3 int
 4 main(int argc, char **argv)
 5 {
 6     CLIENT *cl;
 7     struct timeval tv;

 8     if (argc != 3)
 9         err_quit("usage: client <hostname> <protocol>");

10     cl = Clnt_create(argv[1], SQUARE_PROG, SQUARE_VERS, argv[2]);

11     tv.tv_sec = 10;
12     tv.tv_usec = 0;
13     if (clnt_call(cl, NULLPROC, xdr_void, NULL,
14                   xdr_void, NULL, tv) != RPC_SUCCESS)
15         err_quit("%s", clnt_sperror(cl, argv[1]));

16     exit(0);
17 }
```

sunrpc/square10/client.c

Figure D.13 Client program that calls the server's null procedure.

16.12 The resulting UDP datagram size (65536 + 20 + RPC overhead) exceeds 65535, the maximum size of an IPv4 datagram. In Figure A.4, there are no values for Sun RPC using UDP for message sizes of 16384 and 32768, because this is an older RPCSRC 4.0 implementation that limits the size of the UDP datagrams to around 9000 bytes.

Bibliography

Whenever an electronic copy was found of a paper or report referenced in this bibliography, its URL is included. Be aware that these URLs can change over time, and readers are encouraged to check the Errata for this text on the author's home page for any changes: `http://www.kohala.com/~rstevens`.

Bach, M. J. 1986. *The Design of the UNIX Operating System*. Prentice Hall, Englewood Cliffs, N.J.

Birrell, A. D., and Nelson, B. J. 1984. "Implementing Remote Procedure Calls," *ACM Transactions on Computer Systems*, vol. 2, no. 1, pp. 39–59 (Feb.).

Butenhof, D. R. 1997. *Programming with POSIX Threads*. Addison-Wesley, Reading, Mass.

Corbin, J. R. 1991. *The Art of Distributed Applications: Programming Techniques for Remote Procedure Calls*. Springer-Verlag, New York.

Garfinkel, S. L., and Spafford, E. H. 1996. *Practical UNIX and Internet Security, Second Edition*. O'Reilly & Associates, Sebastopol, Calif.

Goodheart, B., and Cox, J. 1994. *The Magic Garden Explained: The Internals of UNIX System V Release 4, An Open Systems Design*. Prentice Hall, Englewood Cliffs, N.J.

Hamilton, G., and Kougiouris, P. 1993. "The Spring Nucleus: A Microkernel for Objects," *Proceedings of the 1993 Summer USENIX Conference*, pp. 147–159, Cincinnati, Oh.

`http://www.kohala.com/~rstevens/papers.others/springnucleus.1993.ps`

IEEE. 1996. "Information Technology—Portable Operating System Interface (POSIX)—Part 1: System Application Program Interface (API) [C Language]," IEEE Std 1003.1, 1996 Edition, Institute of Electrical and Electronics Engineers, Piscataway, N. J. (July).

This version of Posix.1 contains the 1990 base API, the 1003.1b realtime extensions (1993), the 1003.1c Pthreads (1995), and the 1003.1i technical corrections (1995). This is also International Standard ISO/IEC 9945-1: 1996 (E). Ordering information on IEEE standards and draft standards is available at `http://www.ieee.org`. Unfortunately, the IEEE standards are not freely available on the Internet.

Josey, A., ed. 1997. *Go Solo 2: The Authorized Guide to Version 2 of the Single UNIX Specification.* Prentice Hall, Upper Saddle River, N.J.

Also note that many of the Unix 98 specifications (e.g., all of the manual pages) are available online at `http://www.UNIX-systems.org/online.html`.

Kernighan, B. W., and Pike, R. 1984. *The UNIX Programming Environment.* Prentice Hall, Englewood Cliffs, N.J.

Kernighan, B. W., and Ritchie, D. M. 1988. *The C Programming Language, Second Edition.* Prentice Hall, Englewood Cliffs, N.J.

Kleiman, S., Shah, D., and Smaalders, B. 1996. *Programming with Threads.* Prentice Hall, Upper Saddle River, N.J.

Lewis, B., and Berg, D. J. 1998. *Multithreaded Programming with Pthreads.* Prentice Hall, Upper Saddle River, N.J.

McKusick, M. K., Bostic, K., Karels, M. J., and Quarterman, J. S. 1996. *The Design and Implementation of the 4.4BSD Operating System.* Addison-Wesley, Reading, Mass.

McVoy, L., and Staelin, C. 1996. "lmbench: Portable Tools for Performance Analysis," *Proceedings of the 1996 Winter Technical Conference*, pp. 279–294, San Diego, Calif.

This suite of benchmark tools, along with this paper, are available from `http://www.bitmover.com/lmbench`.

Rochkind, M. J. 1985. *Advanced UNIX Programming.* Prentice Hall, Englewood Cliffs, N.J.

Salus, P. H. 1994. *A Quarter Century of Unix.* Addison-Wesley, Reading, Mass.

Srinivasan, R. 1995a. "RPC: Remote Procedure Call Protocol Specification Version 2," RFC 1831, 18 pages (Aug.).

Srinivasan, R. 1995b. "XDR: External Data Representation Standard," RFC 1832, 24 pages (Aug.).

Srinivasan, R. 1995c. "Binding Protocols for ONC RPC Version 2," RFC 1833, 14 pages (Aug.).

Stevens, W. R. 1992. *Advanced Programming in the UNIX Environment.* Addison-Wesley, Reading, Mass.

All the details of Unix programming. Referred to throughout this text as APUE.

Stevens, W. R. 1994. *TCP/IP Illustrated, Volume 1: The Protocols.* Addison-Wesley, Reading, Mass.

A complete introduction to the Internet protocols. Referred to throughout this text as TCPv1.

Stevens, W. R. 1996. *TCP/IP Illustrated, Volume 3: TCP for Transactions, HTTP, NNTP, and the UNIX Domain Protocols*. Addison-Wesley, Reading, Mass.

Referred to throughout this text as TCPv3.

Stevens, W. R. 1998. *UNIX Network Programming, Volume 1, Second Edition, Networking APIs: Sockets and XTI*. Prentice Hall, Upper Saddle River, N.J.

Referred to throughout this text as UNPv1.

Vahalia, U. 1996. *UNIX Internals: The New Frontiers*. Prentice Hall, Upper Saddle River, N.J.

White, J. E. 1975. "A High-Level Framework for Network-Based Resource Sharing," RFC 707, 27 pages (Dec.).

`http://www.kohala.com/~rstevens/papers.others/rfc707.txt`

Wright, G. R., and Stevens, W. R. 1995. *TCP/IP Illustrated, Volume 2: The Implementation*. Addison-Wesley, Reading, Mass.

The implementation of the Internet protocols in the 4.4BSD-Lite operating system. Referred to throughout this text as TCPv2.

Index

Rather than provide a separate glossary (with most of the entries being acronyms), this index also serves as a glossary for all the acronyms used in this book. The primary entry for the acronym appears under the acronym name. For example, all references to Remote Procedure Call appear under RPC. The entry under the compound term "Remote Procedure Call" refers back to the main entry under RPC.

The notation "definition of" appearing with a C function refers to the boxed function prototype for that function, its primary description. The "definition of" notation for a structure refers to its primary definition. Some functions also contain the notation "source code" if a source code implementation for that function appears in the text.

Function prototype	page
`bool_t clnt_control(CLIENT *cl, unsigned int request, char *ptr);`	418
`CLIENT *clnt_create(const char *host, unsigned long prognum, unsigned long versnum, const char *protocol);`	401
`void clnt_destroy(CLIENT *cl);`	420
`int door_bind(int fd);`	390
`int door_call(int fd, door_arg_t *argp);`	361
`int door_create(Door_server_proc *proc, void *cookie, u_int attr);`	363
`int door_cred(door_cred_t *cred);`	365
`int door_info(int fd, door_info_t *info);`	365
`int door_return(char *dataptr, size_t datasize, door_desc_t *descptr, size_t ndesc);`	365
`int door_revoke(int fd);`	390
`Door_create_proc *door_server_create(Door_create_proc *proc);`	384
`int door_unbind(void);`	390
`void err_dump(const char *fmt, ...);`	512
`void err_msg(const char *fmt, ...);`	512
`void err_quit(const char *fmt, ...);`	512
`void err_ret(const char *fmt, ...);`	511
`void err_sys(const char *fmt, ...);`	511
`int fcntl(int fd, int cmd, ... /* struct flock *arg */ );`	199
`int fstat(int fd, struct stat *buf);`	328
`key_t ftok(const char *pathname, int id);`	28
`int ftruncate(int fd, off_t length);`	327
`int mkfifo(const char *pathname, mode_t mode);`	54
`void *mmap(void *addr, size_t len, int prot, int flags, int fd, off_t offset);`	308
`int msync(void *addr, size_t len, int flags);`	310
`int munmap(void *addr, size_t len);`	309
`int mq_close(mqd_t mqdes);`	77
`int mq_getattr(mqd_t mqdes, struct mq_attr *attr);`	79
`int mq_notify(mqd_t mqdes, const struct sigevent *notification);`	87
`mqd_t mq_open(const char *name, int oflag, ... /* mode_t mode, struct mq_attr *attr */ );`	76
`ssize_t mq_receive(mqd_t mqdes, char *ptr, size_t len, unsigned int *priop);`	83
`int mq_send(mqd_t mqdes, const char *ptr, size_t len, unsigned int prio);`	83
`int mq_setattr(mqd_t mqdes, const struct mq_attr *attr, struct mq_attr *oattr);`	79
`int mq_unlink(const char *name);`	77
`int msgctl(int msqid, int cmd, struct msqid_ds *buff);`	134
`int msgget(key_t key, int oflag);`	130
`ssize_t msgrcv(int msqid, void *ptr, size_t length, long type, int flag);`	132
`int msgsnd(int msqid, const void *ptr, size_t length, int flag);`	131
`int pclose(FILE *stream);`	52
`int pipe(int fd[2]);`	44
`FILE *popen(const char *command, const char *type);`	52

Function prototype	page
`int pthread_cancel(pthread_t tid);`	187
`void pthread_cleanup_pop(int execute);`	187
`void pthread_cleanup_push(void (*function)(void *), void *arg);`	187
`int pthread_create(pthread_t *tid, const pthread_attr_t *attr, void *(*func)(void *), void *arg);`	502
`int pthread_detach(pthread_t tid);`	504
`void pthread_exit(void *status);`	504
`int pthread_join(pthread_t tid, void **status);`	503
`pthread_t pthread_self(void);`	503
`int pthread_condattr_destroy(pthread_condattr_t *attr);`	172
`int pthread_condattr_getpshared(const pthread_condattr_t *attr, int *valptr);`	173
`int pthread_condattr_init(pthread_condattr_t *attr);`	172
`int pthread_condattr_setpshared(pthread_condattr_t *attr, int value);`	173
`int pthread_cond_broadcast(pthread_cond_t *cptr);`	171
`int pthread_cond_destroy(pthread_cond_t *cptr);`	172
`int pthread_cond_init(pthread_cond_t *cptr, const pthread_condattr_t *attr);`	172
`int pthread_cond_signal(pthread_cond_t *cptr);`	167
`int pthread_cond_timedwait(pthread_cond_t *cptr, pthread_mutex_t *mptr, const struct timespec *abstime);`	171
`int pthread_cond_wait(pthread_cond_t *cptr, pthread_mutex_t *mptr);`	167
`int pthread_mutexattr_destroy(pthread_mutexattr_t *attr);`	172
`int pthread_mutexattr_getpshared(const pthread_mutexattr_t *attr, int *valptr);`	173
`int pthread_mutexattr_init(pthread_mutexattr_t *attr);`	172
`int pthread_mutexattr_setpshared(pthread_mutexattr_t *attr, int value);`	173
`int pthread_mutex_destroy(pthread_mutex_t *mptr);`	172
`int pthread_mutex_init(pthread_mutex_t *mptr, const pthread_mutexattr_t *attr);`	172
`int pthread_mutex_lock(pthread_mutex_t *mptr);`	160
`int pthread_mutex_trylock(pthread_mutex_t *mptr);`	160
`int pthread_mutex_unlock(pthread_mutex_t *mptr);`	160
`int pthread_rwlockattr_destroy(pthread_rwlockattr_t *attr);`	179
`int pthread_rwlockattr_getpshared(const pthread_rwlockattr_t *attr, int *valptr);`	179
`int pthread_rwlockattr_init(pthread_rwlockattr_t *attr);`	179
`int pthread_rwlockattr_setpshared(pthread_rwlockattr_t *attr, int value);`	179
`int pthread_rwlock_destroy(pthread_rwlock_t *rwptr);`	179
`int pthread_rwlock_init(pthread_rwlock_t *rwptr, const pthread_rwlockattr_t *attr);`	179
`int pthread_rwlock_rdlock(pthread_rwlock_t *rwptr);`	178
`int pthread_rwlock_tryrdlock(pthread_rwlock_t *rwptr);`	178
`int pthread_rwlock_trywrlock(pthread_rwlock_t *rwptr);`	178
`int pthread_rwlock_unlock(pthread_rwlock_t *rwptr);`	178
`int pthread_rwlock_wrlock(pthread_rwlock_t *rwptr);`	178

Function prototype	page
`long pr_thread_id(pthread_t *ptr);`	371
`char *px_ipc_name(const char *name);`	21
`int sem_close(sem_t *sem);`	226
`int sem_destroy(sem_t *sem);`	239
`int sem_getvalue(sem_t *sem, int *valp);`	227
`int sem_init(sem_t *sem, int shared, unsigned int value);`	239
`sem_t *sem_open(const char *name, int oflag, ... /* mode_t mode, unsigned int value */ );`	225, 227
`int sem_post(sem_t *sem);`	226
`int sem_trywait(sem_t *sem);`	226
`int sem_unlink(const char *name);`	226
`int sem_wait(sem_t *sem);`	226
`int semctl(int semid, int semnum, int cmd, ... /* union semun arg */ );`	287
`int semget(key_t key, int nsems, int oflag);`	282
`int semop(int semid, struct sembuf *opsptr, size_t nops);`	285
`int shm_open(const char *name, int oflag, mode_t mode);`	326
`int shm_unlink(const char *name);`	326
`void *shmat(int shmid, const void *shmaddr, int flag);`	344
`int shmctl(int shmid, int cmd, struct shmid_ds *buff);`	345
`int shmdt(const void *shmaddr);`	345
`int shmget(key_t key, size_t size, int oflag);`	344
`Sigfunc_rt *signal_rt(int signo, Sigfunc_rt *func);`	105
`int sigwait(const sigset_t *set, int *sig);`	95
`int start_time(void);`	470
`double stop_time(void);`	470
`int svc_dg_enablecache(SVCXPRT *xprt, unsigned long size);`	422
`int touch(void *vptr, int nbytes);`	470
`void tv_sub(struct timeval *out, struct timeval *in);`	471

Structure Definitions