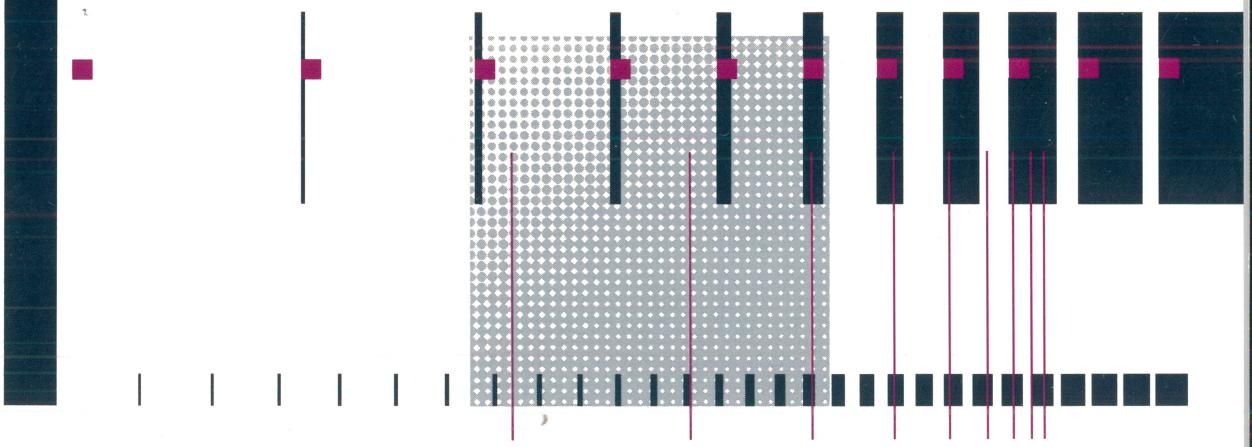
RISC/os (UMIPS)
Programmer's Guide
Volume II

Order Number 3207DOC



mips

The power of RISC is in the system.

RISC/os (UMIPS)
Programmer's Guide
Volume II

Order Number 3207DOC

March 1989

Your comments on our products and publications are welcome. A postage–paid form is provided for this purpose on the last page of this manual.

Mfg. Part Number 84–00062A/02–00298

MIPS Computer Systems, Inc.
930 Arques Ave.
Sunnyvale, CA 94086

Customer Service Telephone Numbers:

California:	(800)	992–MIPS
All other states:	(800)	443–MIPS
International:	(415)	330–7966

Mfg. Part Number 84–00062/02–00298

Chapter 12: curses/terminfo

Introduction

Screen management programs are a common component of many commercial computer applications. These programs handle input and output at a video display terminal. A screen program might move a cursor, print a menu, divide a terminal screen into windows, or draw a display on the screen to help users enter and retrieve information from a database.

This tutorial explains how to use the Terminal Information Utilities package, commonly called **curses/terminfo**, to write screen management programs on a UNIX system. This package includes a library of C routines, a database, and a set of UNIX system support tools. To start you writing screen management programs as soon as possible, the tutorial does not attempt to cover every part of the package. For instance, it covers only the most frequently used routines and then points you to **curses**(3X) and **terminfo**(4) in the *Programmer's Reference Manual* for more information. Keep the manual close at hand; you'll find it invaluable when you want to know more about one of these routines or about other routines not discussed here.

Because the routines are compiled C functions, you should be familiar with the C programming language before using **curses/terminfo**. You should also be familiar with the UNIX system/C language standard I/O package (see "System Calls and Subroutines" and "Input/Output" in Chapter 2 and **stdio**(3S)). With that knowledge and an appreciation for the UNIX philosophy of building on the work of others, you can design screen management programs for many purposes.

This chapter has five sections:

- Overview

  This section briefly describes **curses**, **terminfo**, and the other components of the Terminal Information Utilities package.

- Working with **curses** Routines

  This section describes the basic routines making up the **curses**(3X) library. It covers the routines for writing to a screen, reading from a screen, and building windows. It also covers routines for more advanced screen management programs that draw line graphics, use a terminal's soft labels, and work with more than one terminal at the same time. Many examples are included to show the effect of using these routines.

- Working with **terminfo** Routines

  This section describes the routines in the **curses** library that deal directly with the **terminfo** database to handle certain terminal capabilities, such as programming function keys.

- Working with the **terminfo** Database

  This section describes the **terminfo** database, related support tools, and their relationship to the **curses** library.

- **curses** Program Examples

  This section includes six programs that illustrate uses of **curses** routines.

Overview

What is curses?

curses(3X) is the library of routines that you use to write screen management programs on the UNIX system. The routines are C functions and macros; many of them resemble routines in the standard C library. For example, there's a routine **printw()** that behaves much like **printf**(3S) and another routine **getch()** that behaves like **getc**(3S). The automatic teller program at your bank might use **printw()** to print its menus and **getch()** to accept your requests for withdrawals (or, better yet, deposits). A visual screen editor like the UNIX system screen editor **vi**(1) might also use these and other **curses** routines.

The **curses** routines are usually located in **/usr/lib/libcurses.a**. To compile a program using these routines, you must use the **cc**(1) command and include **–lcurses** on the command line so that the link editor can locate and load them:

cc *file*.**c** **–lcurses** **–o** *file*

The name **curses** comes from the cursor optimization that this library of routines provides. Cursor optimization minimizes the amount a cursor has to move around a screen to update it. For example, if you designed a screen editor program with **curses** routines and edited the sentence

```
curses/terminfo is a great package for creating screens.
```

to read

```
curses/terminfo is the best package for creating screens.
```

the program would output only `the best` in place of `a great`. The other characters would be preserved. Because the amount of data transmitted—the output—is minimized, cursor optimization is also referred to as output optimization.

Cursor optimization takes care of updating the screen in a manner appropriate for the terminal on which a **curses** program is run. This means that the **curses** library can do whatever is required to update many different terminal types. It searches the **terminfo** database (described below) to find the correct description for a terminal.

How does cursor optimization help you and those who use your programs? First, it saves you time in describing in a program how you want to update screens. Second, it saves a user's time when the screen is updated. Third, it reduces the load on your UNIX system's communication lines when the updating takes place. Fourth, you don't have to worry about the myriad of terminals on which your program might be run.

Here's a simple **curses** program. It uses some of the basic **curses** routines to move a cursor to the middle of a terminal screen and print the character string **BullsEye**. Each of these routines is described in the following section "Working with **curses** Routines" in this chapter. For now, just look at their names and you will get an idea of what each of them does:

```
#include <curses.h>

main()
{
   initscr();

   move( LINES/2 - 1, COLS/2 - 4 );
   addstr("Bulls");
   refresh();
   addstr("Eye");
   refresh();
   endwin();
}
```

Figure 12-1: A Simple **curses** Program

What Is terminfo?

terminfo refers to both of the following:

- It is a group of routines within the **curses** library that handles certain terminal capabilities. You can use these routines to program function keys, if your terminal has programmable keys, or write filters, for example. Shell programmers, as well as C programmers, can use the **terminfo** routines in their programs.
- It is a database containing the descriptions of many terminals that can be used with **curses** programs. These descriptions specify the capabilities of a terminal and the way it performs various operations—for example, how many lines and columns it has and how its control characters are interpreted.

  Each terminal description in the database is a separate, compiled file. You use the source code that **terminfo**(4) describes to create these files and the command **tic**(1M) to compile them.

  The compiled files are normally located in the directories **/usr/lib/terminfo/?**. These directories have single character names, each of which is the first character in the name of a terminal. For example, an entry for the AT&T Teletype 5425 is normally located in the file **/usr/lib/terminfo/a/att5425**.

Here's a simple shell script that uses the **terminfo** database.

```
#   Clear the screen and show the 0,0 position.
#
tput clear
tput cup 0 0            # or tput home
echo "<- this is 0 0"

#
#   Show the 5,10 position.
#
tput cup 5 10
echo "<- this is 5 10"
```

Figure 12-2: A Shell Script Using **terminfo** Routines

How curses and terminfo Work Together

A screen management program with **curses** routines refers to the **terminfo** database at run time to obtain the information it needs about the terminal being used—what we'll call the current terminal from here on.

For example, suppose you are using an AT&T Teletype 5425 terminal to run the simple **curses** program shown in Figure 12-1. To execute properly, the program needs to know how many lines and columns the terminal screen has to print the **BullsEye** in the middle of it. The description of the AT&T Teletype 5425 in the **terminfo** database has this information. All the **curses** program needs to know before it goes looking for the information is the name of your terminal. You tell the program the name by putting it in the environment variable **$TERM** when you log in or by setting and exporting **$TERM** in your **.profile** file (see **profile**(4)). Knowing **$TERM**, a **curses** program run on the current terminal can search the **terminfo** database to find the correct terminal description.

For example, assume that the following example lines are in a **.profile**:

```
TERM=5425
export TERM
tput init
```

The first line names the terminal type, and the second line exports it. (See **profile**(4) in the *Programmer's Reference Manual*.) The third line of the example tells the UNIX system to initialize the current terminal. That is, it makes sure that the terminal is set up according to its description in the **terminfo** database. (The order of these lines is important. **$TERM** must be defined and exported first, so that when **tput** is called the proper initialization for the current terminal takes place.) If you had these lines in your **.profile** and you ran a **curses** program, the program would get the information that it needs about your terminal from the file **/usr/lib/terminfo/a/att5425**, which provides a match for **$TERM**.

Other Components of the Terminal Information Utilities

We said earlier that the Terminal Information Utilities is commonly referred to as **curses/terminfo**. The package, however, has other components. We've mentioned some of them, for instance **tic**(1M). Here's a complete list of the components discussed in this tutorial:

captoinfo(1M)	a tool for converting terminal descriptions developed on earlier releases of the UNIX system to **terminfo** descriptions
curses(3X)	
infocmp(1M)	a tool for printing and comparing compiled terminal descriptions
tabs(1)	a tool for setting non-standard tab stops
terminfo(4)	
tic(1M)	a tool for compiling terminal descriptions for the **terminfo** database
tput(1)	a tool for initializing the tab stops on a terminal and for outputting the value of a terminal capability

We also refer to **profile**(4), **scr_dump**(4), **term**(4), and **term**(5). For more information about any of these components, see the *Programmer's Reference Manual*, the *System Administrator's Reference Manual*, and the *User's Reference Manual*.

Working with curses Routines

This section describes the basic **curses** routines for creating interactive screen management programs. It begins by describing the routines and other program components that every **curses** program needs to work properly. Then it tells you how to compile and run a **curses** program. Finally, it describes the most frequently used **curses** routines that

- write output to and read input from a terminal screen
- control the data output and input — for example, to print output in bold type or prevent it from echoing (printing back on a screen)
- manipulate multiple screen images (windows)
- draw simple graphics
- manipulate soft labels on a terminal screen
- send output to and accept input from more than one terminal.

To illustrate the effect of using these routines, we include simple example programs as the routines are introduced. We also refer to a group of larger examples located in the section "**curses** Program Examples" in this chapter. These larger examples are more challenging; they sometimes make use of routines not discussed here. Keep the **curses**(3X) manual page handy.

What Every curses Program Needs

All **curses** programs need to include the header file **<curses.h>** and call the routines **initscr**(), **refresh**() or similar related routines, and **endwin**().

The Header File <curses.h>

The header file **<curses.h>** defines several global variables and data structures and defines several **curses** routines as macros.

To begin, let's consider the variables and data structures defined. **<curses.h>** defines all the parameters used by **curses** routines. It also defines the integer variables **LINES** and **COLS**; when a **curses** program is run on a particular terminal, these variables are assigned the vertical and horizontal dimensions of the terminal screen, respectively, by the routine **initscr**() described below. The header file defines the constants **OK** and **ERR**, too. Most **curses** routines have return values; the **OK** value is returned if a routine is properly completed, and the **ERR** value if some error occurs.

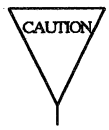
LINES and **COLS** are external (global) variables that represent the size of a terminal screen. Two similar variables, **$LINES** and **$COLUMNS**, may be set in a user's shell environment; a **curses** program uses the environment variables to determine the size of a screen. Whenever we refer to the environment variables in this chapter, we will use the **$** to distinguish them from the C declarations in the **<curses.h>** header file.

For more information about these variables, see the following sections "The Routines **initscr**(), **refresh**(), and **endwin**()" and "More about **initscr**() and Lines and Columns."

Now let's consider the macro definitions. **<curses.h>** defines many **curses** routines as macros that call other macros or **curses** routines. For instance, the simple routine **refresh**() is a macro. The line

```
#define refresh() wrefresh(stdscr)
```

shows when **refresh** is called, it is expanded to call the **curses** routine **wrefresh()**. The latter routine in turn calls the two **curses** routines **wnoutrefresh()** and **doupdate()**. Many other routines also group two or three routines together to achieve a particular result.



Macro expansion in **curses** programs may cause problems with certain sophisticated C features, such as the use of automatic incrementing variables.

One final point about **<curses.h>**: it automatically includes **<stdio.h>** and the **<termio.h>** tty driver interface file. Including either file again in a program is harmless but wasteful.

The Routines initscr(), refresh(), endwin()

The routines **initscr()**, **refresh()**, and **endwin()** initialize a terminal screen to an "in **curses** state," update the contents of the screen, and restore the terminal to an "out of **curses** state," respectively. Use the simple program that we introduced earlier to learn about each of these routines:

```
#include <curses.h>

main()
{
   initscr();    /* initialize terminal settings
                    and <curses.h> data
                    structures and variables */

   move( LINES/2 - 1, COLS/2 - 4 );
   addstr("Bulls");
   refresh();    /* send output to (update) terminal screen */
   addstr("Eye");
   refresh();    /* send more output to terminal screen */
   endwin();     /* restore all terminal settings */
}
```

Figure 12-3: The Purposes of **initscr()**, **refresh()**, and **endwin()** in a Program

A **curses** program usually starts by calling **initscr()**; the program should call **initscr()** only once. Using the environment variable **$TERM** as the section "How **curses** and **terminfo** Work Together" describes, this routine determines what terminal is being used. It then initializes all the declared data structures and other variables from **<curses.h>**. For example, **initscr()** would initialize **LINES** and **COLS** for the sample program on whatever terminal it was run. If the Teletype 5425 were used, this routine would initialize **LINES** to 24 and **COLS** to 80. Finally, this routine writes error messages to **stderr** and exits if errors occur.

During the execution of the program, output and input is handled by routines like **move()** and **addstr()** in the sample program. For example,

```
move( LINES/2 - 1, COLS/2 - 4 );
```

says to move the cursor to the left of the middle of the screen. Then the line

```
addstr("Bulls");
```

says to write the character string `Bulls`. For example, if the Teletype 5425 were used, these routines would position the cursor and write the character string at (11,36).

All **curses** routines that move the cursor move it from its home position in the upper left corner of a screen. The **(LINES,COLS)** coordinate at this position is (0,0) not (1,1). Notice that the vertical coordinate is given first and the horizontal second, which is the opposite of the more common 'x,y' order of screen (or graph) coordinates. The `-1` in the sample program takes the (0,0) position into account to place the cursor on the center line of the terminal screen.

Routines like **move()** and **addstr()** do not actually change a physical terminal screen when they are called. The screen is updated only when **refresh()** is called. Before this, an internal representation of the screen called a window is updated. This is a very important concept, which we discuss below under "More about **refresh()** and Windows."

Finally, a **curses** program ends by calling **endwin()**. This routine restores all terminal settings and positions the cursor at the lower left corner of the screen.

Compiling a curses Program

You compile programs that include **curses** routines as C language programs using the **cc**(1) command (documented in the *Programmer's Reference Manual*), which invokes the C compiler (see Chapter 2 in this guide for details).

The routines are usually stored in the library **/usr/lib/libcurses.a**. To direct the link editor to search this library, you must use the **–l** option with the **cc** command.

The general command line for compiling a **curses** program follows:

cc *file*.**c** **–lcurses** **–o** *file*

file.**c** is the name of the source program; and *file* is the executable object module.

Running a curses Program

curses programs count on certain information being in a user's environment to run properly. Specifically, users of a **curses** program should usually include the following three lines in their **.profile** files:

```
TERM=current terminal type
export TERM
tput init
```

For an explanation of these lines, see the section "How **curses** and **terminfo** Work Together" in this chapter. Users of a **curses** program could also define the environment variables **$LINES**, **$COLUMNS**, and **$TERMINFO** in their **.profile** files. However, unlike **$TERM**, these variables do not have to be defined.

If a **curses** program does not run as expected, you might want to debug it with **dbx**(1), which is documented in the *Programmer's Reference Manual*). When using **dbx**, you have to keep a few points in mind. First, a **curses** program is interactive and always has knowledge of where the cursor is located. An interactive debugger like **dbx**, however, may cause changes to the contents of the screen of which the **curses** program is not aware.

Second, a **curses** program outputs to a window until **refresh**() or a similar routine is called. Because output from the program may be delayed, debugging the output for consistency may be difficult.

Third, setting break points on **curses** routines that are macros, such as **refresh**(), does not work. You have to use the routines defined for these macros, instead; for example, you have to use **wrefresh**() instead of **refresh**(). See the above section, "The Header File **<curses.h>**," for more information about macros.

More about initscr() and Lines and Columns

After determining a terminal's screen dimensions, **initscr**() sets the variables **LINES** and **COLS**. These variables are set from the **terminfo** variables **lines** and **columns**. These, in turn, are set from the values in the **terminfo** database, unless these values are overridden by the values of the environment **$LINES** and **$COLUMNS**.

More about refresh() and Windows

As mentioned above, **curses** routines do not update a terminal until **refresh**() is called. Instead, they write to an internal representation of the screen called a window. When **refresh**() is called, all the accumulated output is sent from the window to the current terminal screen.

A window acts a lot like a buffer does when you use a UNIX system editor. When you invoke **vi**(1), for instance, to edit a file, the changes you make to the contents of the file are reflected in the buffer. The changes become part of the permanent file only when you use the **w** or **ZZ** command. Similarly, when you invoke a screen program made up of **curses** routines, they change the contents of a window. The changes become part of the current terminal screen only when **refresh**() is called.

<curses.h> supplies a default window named **stdscr** (standard screen), which is the size of the current terminal's screen, for all programs using **curses** routines. The header file defines **stdscr** to be of the type **WINDOW***, a pointer to a C structure which you might think of as a two-dimensional array of characters representing a terminal screen. The program always keeps track of what is on the physical screen, as well as what is in **stdscr**. When **refresh**() is called, it compares the two screen images and sends a stream of characters to the terminal that make the current screen look like **stdscr**. A **curses** program considers many different ways to do this, taking into account the various capabilities of the terminal and similarities between what is on the screen and what is on the window. It optimizes output by printing as few characters as is possible. Figure 12-4 illustrates what happens when you execute the sample curses program that prints **BullsEye** at the center of a terminal screen (see Figure 12-1). Notice in the figure that the terminal screen retains whatever garbage is on it until the first **refresh**() is called. This **refresh**() clears the screen and updates it with the current contents of **stdscr**.

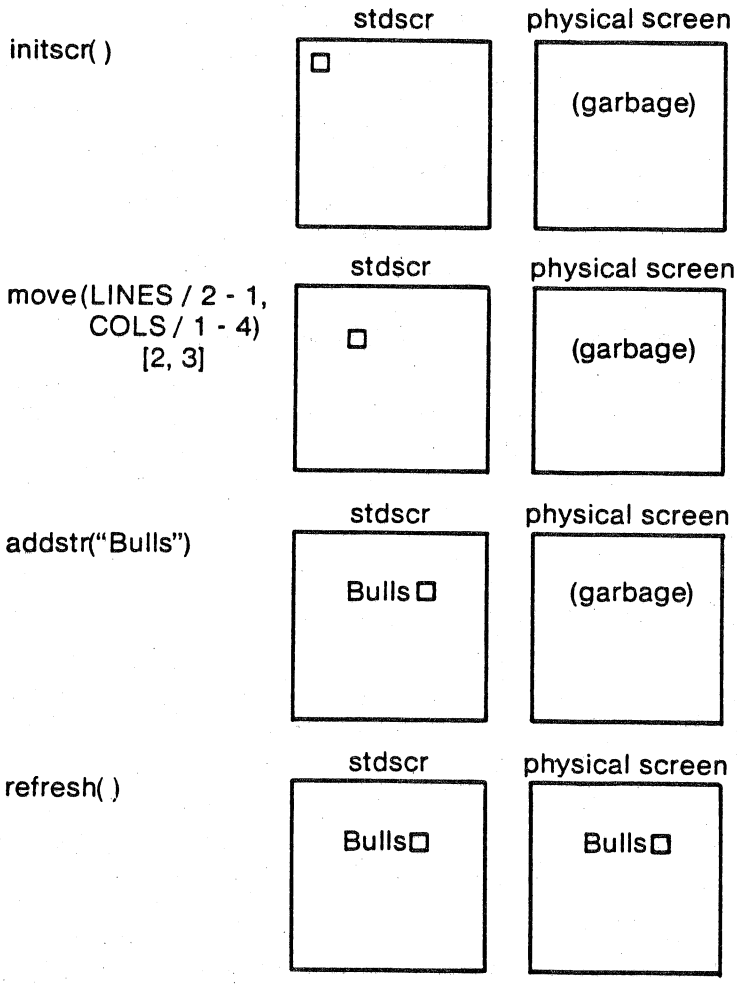


Figure 12-4: The Relationship between **stdscr** and a Terminal Screen (Sheet 1 of 2)

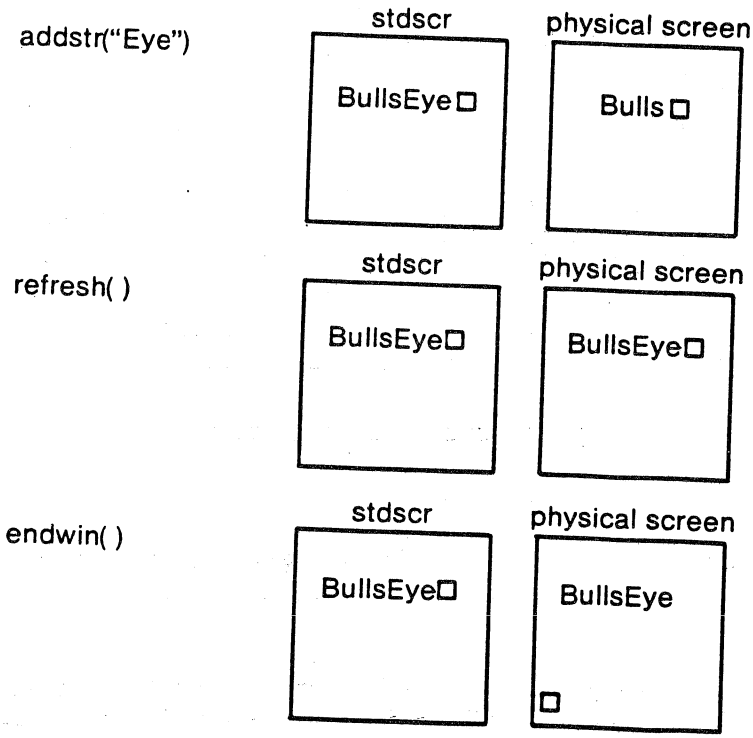


Figure 12-4: The Relationship Between **stdscr** and a Terminal Screen (Sheet 2 of 2)

You can create other windows and use them instead of **stdscr**. Windows are useful for maintaining several different screen images. For example, many data entry and retrieval applications use two windows: one to control input and output and one to print error messages that don't mess up the other window.

It's possible to subdivide a screen into many windows, refreshing each one of them as desired. When windows overlap, the contents of the current screen show the most recently refreshed window. It's also possible to create a window within a window; the smaller window is called a subwindow. Assume that you are designing an application that uses forms, for example, an expense voucher, as a user interface. You could use subwindows to control access to certain fields on the form.

Some **curses** routines are designed to work with a special type of window called a pad. A pad is a window whose size is not restricted by the size of a screen or associated with a particular part of a screen. You can use a pad when you have a particularly large window or only need part of the window on the screen at any one time. For example, you might use a pad for an application with a spread sheet.

Figure 12-5 represents what a pad, a subwindow, and some other windows could look like in comparison to a terminal screen.

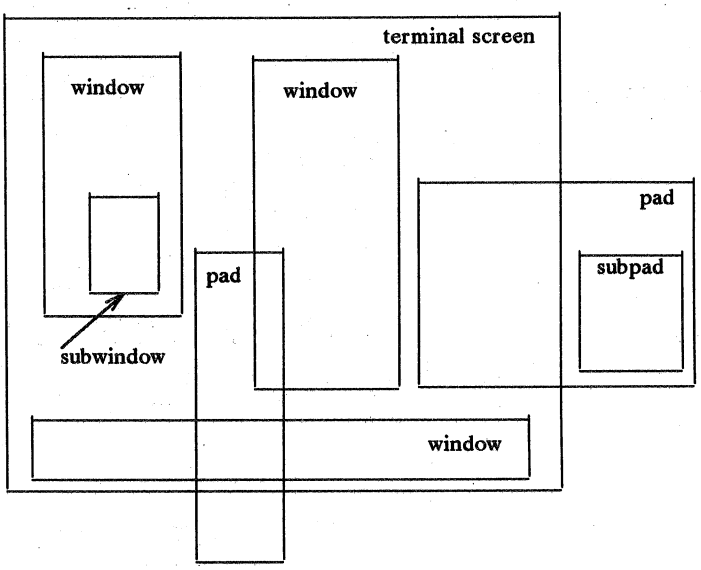


Figure 12-5: Multiple Windows and Pads Mapped to a Terminal Screen

The section "Building Windows and Pads" in this chapter describes the routines you use to create and use them. If you'd like to see a **curses** program with windows now, you can turn to the **window** program under the section "**curses** Program Examples" in this chapter.

Getting Simple Output and Input

Output

The routines that **curses** provides for writing to **stdscr** are similar to those provided by the **stdio**(3S) library for writing to a file. They let you:

- write a character at a time — **addch**()
- write a string — **addstr**()
- format a string from a variety of input arguments — **printw**()
- move a cursor or move a cursor and print character(s) — **move**(), **mvaddch**(), **mvaddstr**(), **mvprintw**()
- clear a screen or a part of it — **clear**(), **erase**(), **clrtoeol**(), **clrtobot**()

Following are descriptions and examples of these routines.

The **curses** library provides its own set of output and input functions. You should not use other I/O routines or system calls, like **read**(2) and **write**(2), in a **curses** program. They may cause undesirable results when you run the program.

addch()

SYNOPSIS

#include <curses.h>

int addch(ch)
chtype ch;

NOTES

- **addch**() writes a single character to **stdscr**.
- The character is of the type **chtype**, which is defined in **<curses.h>**. **chtype** contains data and attributes (see "Output Attributes" in this chapter for information about attributes).
- When working with variables of this type, make sure you declare them as **chtype** and not as the basic type (for example, **short**) that **chtype** is declared to be in **<curses.h>**. This will ensure future compatibility.
- **addch**() does some translations. For example, it converts
  - the **<NL>** character to a clear to end of line and a move to the next line
  - the tab character to an appropriate number of blanks
  - other control characters to their ^*X* notation
- **addch**() normally returns **OK**. The only time **addch**() returns **ERR** is after adding a character to the lower right-hand corner of a window that does not scroll.
- **addch**() is a macro.

EXAMPLE

```
#include <curses.h>

main()
{
   initscr();
   addch('a');
   refresh();
   endwin();
}
```

The output from this program will appear as follows:

```
a


$
```

Also see the **show** program under "**curses** Example Programs" in this chapter.

addstr()

SYNOPSIS

#include <curses.h>

int addstr(str)
char *str;

NOTES

- **addstr()** writes a string of characters to **stdscr**.
- **addstr()** calls **addch()** to write each character.
- **addstr()** follows the same translation rules as **addch()**.
- **addstr()** returns **OK** on success and **ERR** on error.
- **addstr()** is a macro.

EXAMPLE

Recall the sample program that prints the character string **BullsEye**. See Figures 12-1, 12-2, and 12-4.

printw()

SYNOPSIS

#include <curses.h>

int printw(fmt [,arg...])
char *fmt

NOTES

- **printw()** handles formatted printing on **stdscr**.
- Like **printf**, **printw()** takes a format string and a variable number of arguments.
- Like **addstr()**, **printw()** calls **addch()** to write the string.
- **printw()** returns **OK** on success and **ERR** on error.

EXAMPLE

```
#include <curses.h>

main()
{
   char* title = "Not specified";
   int no = 0;

      /* Missing code. */

   initscr();

      /* Missing code. */

   printw("%s is not in stock.\n", title);
   printw("Please ask the cashier to order %d for you.\n", no);

   refresh();
   endwin();
}
```

The output from this program will appear as follows:

```
Not specified is not in stock.
Please ask the cashier to order 0 for you.


$
```

move()

SYNOPSIS

#include <curses.h>

int move(y, x);
int y, x;

NOTES

- **move**() positions the cursor for **stdscr** at the given row **y** and the given column **x.**
- Notice that **move**() takes the **y** coordinate before the **x** coordinate. The upper left-hand coordinates for **stdscr** are (0,0), the lower right-hand (**LINES** - 1, **COLS** - 1). See the section "The Routines **initscr**(), **refresh**(), and **endwin**()" for more information.

- **move()** may be combined with the write functions to form:
  - **mvaddch(y, x, ch)**, which moves to a given position and prints a character
  - **mvaddstr(y, x, str)**, which moves to a given position and prints a string of characters
  - **mvprintw(y, x, fmt** [,*arg*...]),
    which moves to a given position and prints a formatted string.
- **move()** returns **OK** on success and **ERR** on error. Trying to move to a screen position of less than (0,0) or more than (**LINES** - 1, **COLS** - 1) causes an error.
- **move()** is a macro.

EXAMPLE

```
#include <curses.h>

main()
{
   initscr();
   addstr("Cursor should be here --> if move() works.");
   printw("\n\n\nPress <CR> to end test.");
   move(0,25);
   refresh();
   getch();      /* Gets <CR>; discussed below. */
   endwin();
}
```

Here's the output generated by running this program:

```
Cursor should be here —>□if move() works.

Press <CR> to end test.
```

After you press **<CR>**, the screen looks like this:

```
Cursor should be here —>

Press <CR> to end test.
$□
```

See the **scatter** program under "**curses** Program Examples" in this chapter for another example of using **move()**.

clear() and erase()

SYNOPSIS

#include <curses.h>

int clear()
int erase()

NOTES

- Both routines change **stdscr** to all blanks.
- **clear()** also assumes that the screen may have garbage that it doesn't know about; this routine first calls **erase()** and then **clearok()** which clears the physical screen completely on the next call to **refresh()** for **stdscr**. See the **curses**(3X) manual page for more information about **clearok()**.
- **initscr()** automatically calls **clear()**.
- **clear()** always returns **OK**; **erase()** returns no useful value.
- Both routines are macros.

clrtoeol() and clrtobot()

SYNOPSIS

#include <curses.h>

int clrtoeol()
int clrtobot()

NOTES

- **clrtoeol()** changes the remainder of a line to all blanks.
- **clrtobot()** changes the remainder of a screen to all blanks.
- Both begin at the current cursor position inclusive.
- Neither returns any useful value.

EXAMPLE

The following sample program uses **clrtobot()**.

```
#include <curses.h>

main()
{
    initscr();
    addstr("Press <CR> to delete from here to the end of the \
                    line and on.");
    addstr("\nDelete this too.\nAnd this.");
    move(0,30);
    refresh();
    getch();
    clrtobot();
    refresh();
    endwin();
}
```

Here's the output generated by running this program:

```
Press <CR> to delete from here to the end of the line and on.
Delete this too.
And this.
```

Notice the two calls to **refresh**(): one to send the full screen of text to a terminal, the other to clear from the position indicated to the bottom of a screen.

Here's what the screen looks like when you press **<CR>**:

```
Press <CR> to delete from here

$
```

See the **show** and **two** programs under "**curses** Example Programs" for examples of uses for **clrtoeol**().

Input

curses routines for reading from the current terminal are similar to those provided by the **stdio**(3S) library for reading from a file. They let you:

- read a character at a time — **getch**()
- read a **<NL>**-terminated string — **getstr**()
- parse input, converting and assigning selected data to an argument list — **scanw**()

The primary routine is **getch()**, which processes a single input character and then returns that character. This routine is like the C library routine **getchar**()(3S) except that it makes several terminal- or system-dependent options available that are not possible with **getchar**(). For example, you can use **getch**() with the **curses** routine **keypad**(), which allows a **curses** program to interpret extra keys on a user's terminal, such as arrow keys, function keys, and other special keys that transmit escape sequences, and treat them as just another key. See the descriptions of **getch**() and **keypad**() on the **curses**(3X) manual page for more information about **keypad**().

The following pages describe and give examples of the basic routines for getting input in a screen program.

getch()

SYNOPSIS

#include <curses.h>

int getch()

NOTES

- **getch**() reads a single character from the current terminal.
- **getch**() returns the value of the character or **ERR** on 'end of file,' receipt of signals, or non-blocking read with no input.
- **getch**() is a macro.
- See the discussions about **echo**(), **noecho**(), **cbreak**(), **nocbreak**(), **raw**(), **noraw**(), **halfdelay**(), **nodelay**(), and **keypad**() below and in **curses**(3X).

EXAMPLE

```
#include <curses.h>

main()
{
   int ch;

   initscr();
   cbreak(); /* Explained later in the */
             /*  section "Input Options" */
   addstr("Press any character:  ");
   refresh();
   ch = getch();
   printw("\n\n\nThe character entered was a '%c'.\n", ch);
   refresh();
   endwin();
}
```

The output from this program follows. The first **refresh**() sends the **addstr**() character string from **stdscr** to the terminal:

```
Press any character: □
```

Then assume that a **w** is typed at the keyboard. **getch**() accepts the character and assigns it to **ch**. Finally, the second **refresh**() is called and the screen appears as follows:

```
Press any character: w

The character entered was a 'w'.

$□
```

For another example of **getch**(), see the **show** program under "**curses** Example Programs" in this chapter.

getstr()

SYNOPSIS

```
#include <curses.h>

int getstr(str)
char *str;
```

NOTES

- **getstr**() reads characters and stores them in a buffer until a **<CR>**, **<NL>**, or **<ENTER>** is received from **stdscr**. **getstr**() does not check for buffer overflow.
- The characters read and stored are in a character string.
- **getstr**() is a macro; it calls **getch**() to read each character.
- **getstr**() returns **ERR** if **getch**() returns **ERR** to it. Otherwise it returns **OK**.
- See the discussions about **echo**(), **noecho**(), **cbreak**(), **nocbreak**(), **raw**(), **noraw**(), **halfdelay**(), **nodelay**(), and **keypad**() below and in **curses**(3X).

EXAMPLE

```
#include <curses.h>

main()
{
char str[256];

   initscr();
   cbreak(); /* Explained later in the */
             /* section "Input Options" */
   addstr("Enter a character string terminated by <CR>:\n\n");
   refresh()
   getstr(str);
   printw("\n\n\nThe string entered was \n'%s'\n", str);
   refresh();
   endwin();
}
```

Assume you entered the string 'I enjoy learning about the UNIX system.' The final screen (after entering **<CR>**) would appear as follows:

```
Enter a character string terminated by <CR>:

I enjoy learning about the UNIX system.


The string entered was
'I enjoy learning about the UNIX system.'

$
```

scanw()

SYNOPSIS

#include <curses.h>

int scanw(fmt [, arg...])
char *fmt;

NOTES

- **scanw**() calls **getstr**() and parses an input line.
- Like **scanf**(3S), **scanw**() uses a format string to convert and assign to a variable number of arguments.
- **scanw**() returns the same values as **scanf**().
- See **scanf**(3S) for more information.

EXAMPLE

```
#include <curses.h>

main()
{
    char string[100];
    float number;

    initscr();
    cbreak();           /* Explained later in the  */
    echo();             /* section "Input Options" */
    addstr("Enter a number and a string separated by a comma: ");
    refresh();
    scanw("%f,%s",&number,string);
    clear();
    printw("The string was \"%s\" and the number was %f.",string,number);
    refresh();
    endwin();
}
```

Notice the two calls to **refresh()**. The first call updates the screen with the character string passed to **addstr()**, the second with the string returned from **scanw()**. Also notice the call to **clear()**. Assume you entered the following when prompted: **2,twin**. After running this program, your terminal screen would appear, as follows:

```
The string was "twin" and the number was 2.000000.

$
```

Controlling Output and Input

Output Attributes

When we talked about **addch()**, we said that it writes a single character of the type **chtype** to **stdscr**. **chtype** has two parts: a part with information about the character itself and another part with information about a set of attributes associated with the character. The attributes allow a character to be printed in reverse video, bold, underlined, and so on.

stdscr always has a set of current attributes that it associates with each character as it is written. However, using the routine **attrset()** and related **curses** routines described below, you can change the current attributes. Below is a list of the attributes and what they mean:

- A_BLINK — blinking
- A_BOLD — extra bright or bold

- A_DIM — half bright
- A_REVERSE — reverse video
- A_STANDOUT — a terminal's best highlighting mode
- A_UNDERLINE — underlining
- A_ALTCHARSET — alternate character set (see the section "Drawing Lines and Other Graphics" in this chapter)

To use these attributes, you must pass them as arguments to **attrset()** and related routines; they can also be ORed with the bitwise OR (|) to **addch()**.

NOTE

Not all terminals are capable of displaying all attributes. If a particular terminal cannot display a requested attribute, a **curses** program attempts to find a substitute attribute. If none is possible, the attribute is ignored.

Let's consider a use of one of these attributes. To display a word in bold, you would use the following code:

```
...
printw("A word in ");
attrset(A_BOLD);
printw("boldface");
attrset(0);
printw(" really stands out.\n");
 ...
refresh();
```

Attributes can be turned on singly, such as **attrset**(A_BOLD) in the example, or in combination. To turn on blinking bold text, for example, you would use **attrset**(A_BLINK | A_BOLD). Individual attributes can be turned on and off with the **curses** routines **attron()** and **attroff()** without affecting other attributes. **attrset(0)** turns all attributes off.

Notice the attribute called A_STANDOUT. You might use it to make text attract the attention of a user. The particular hardware attribute used for standout is the most visually pleasing attribute a terminal has. Standout is typically implemented as reverse video or bold. Many programs don't really need a specific attribute, such as bold or reverse video, but instead just need to highlight some text. For such applications, the A_STANDOUT attribute is recommended. Two convenient functions, **standout()** and **standend()** can be used to turn on and off this attribute. **standend()**, in fact, turns of all attributes.

In addition to the attributes listed above, there are two bit masks called A_CHARTEXT and A_ATTRIBUTES. You can use these bit masks with the **curses** function **inch()** and the C logical AND (&) operator to extract the character or attributes of a position on a terminal screen. See the discussion of **inch()** on the **curses**(3X) manual page.

Following are descriptions of **attrset()** and the other **curses** routines that you can use to manipulate attributes.

attron(), attrset(), and attroff()

SYNOPSIS

#include <curses.h>

int attron(attrs)
chtype attrs;

int attrset(attrs)
chtype attrs;

int attroff(attrs)
chtype attrs;

NOTES

- **attron**() turns on the requested attribute **attrs** in addition to any that are currently on. **attrs** is of the type **chtype** and is defined in **<curses.h>**.
- **attrset**() turns on the requested attributes **attrs** instead of any that are currently turned on.
- **attroff**() turns off the requested attributes **attrs** if they are on.
- The attributes may be combined using the bitwise OR (|).
- All return **OK**.

EXAMPLE

See the **highlight** program under "**curses** Example Programs" in this chapter.

standout() and standend()

SYNOPSIS

#include <curses.h>

int standout()
int standend()

NOTES

- **standout**() turns on the preferred highlighting attribute, A_STANDOUT, for the current terminal. This routine is equivalent to **attron**(A_STANDOUT).
- **standend**() turns off all attributes. This routine is equivalent to **attrset**(0).
- Both always return **OK**.

EXAMPLE

See the **highlight** program under "**curses** Example Programs" in this chapter.

Bells, Whistles, and Flashing Lights

Occasionally, you may want to get a user's attention. Two **curses** routines were designed to help you do this. They let you ring the terminal's chimes and flash its screen.

flash() flashes the screen if possible, and otherwise rings the bell. Flashing the screen is intended as a bell replacement, and is particularly useful if the bell bothers someone within ear shot of the user. The routine **beep**() can be called when a real beep is desired. (If for some reason the terminal is unable to beep, but able to flash, a call to **beep**() will flash the screen.)

beep() and flash()

SYNOPSIS
#include <curses.h>

int flash()
int beep()

NOTES

- **flash**() tries to flash the terminals screen, if possible, and, if not, tries to ring the terminal bell.
- **beep**() tries to ring the terminal bell, if possible, and, if not, tries to flash the terminal screen.
- Neither returns any useful value.

Input Options

The UNIX system does a considerable amount of processing on input before an application ever sees a character. For example, it does the following:

- echoes (prints back) characters to a terminal as they are typed
- interprets an erase character (typically **#**) and a line kill character (typically **@**)
- interprets a CTRL-D (control d) as end of file (EOF)
- interprets interrupt and quit characters
- strips the character's parity bit
- translates **<CR>** to **<NL>**

Because a **curses** program maintains total control over the screen, **curses** turns off echoing on the UNIX system and does echoing itself. At times, you may not want the UNIX system to process other characters in the standard way in an interactive screen management program. Some **curses** routines, **noecho**() and **cbreak**(), for example, have been designed so that you can change the standard character processing. Using these routines in an application controls how input is interpreted. Figure 12-6 shows some of the major routines for controlling input.

Every **curses** program accepting input should set some input options. This is because when the program starts running, the terminal on which it runs may be in **cbreak**(), **raw**(), **nocbreak**(), or **noraw**() mode. Although the **curses** program starts up in **echo**() mode, as Figure 12-6 shows, none of the other modes are guaranteed.

The combination of **noecho()** and **cbreak()** is most common in interactive screen management programs. Suppose, for instance, that you don't want the characters sent to your application program to be echoed wherever the cursor currently happens to be; instead, you want them echoed at the bottom of the screen. The **curses** routine **noecho()** is designed for this purpose. However, when **noecho()** turns off echoing, normal erase and kill processing is still on. Using the routine **cbreak()** causes these characters to be uninterpreted.

Input Options	Characters Interpreted	Uninterpreted
Normal 'out of **curses** state'	interrupt, quit stripping **<CR>** to **<NL>** echoing erase, kill EOF	
Normal **curses** 'start up state'	echoing (simulated)	All else undefined.
cbreak() and **echo()**	interrupt, quit stripping echoing	erase, kill EOF
cbreak() and **noecho()**	interrupt, quit stripping	echoing erase, kill EOF
nocbreak() and **noecho()**	break, quit stripping erase, kill EOF	echoing
nocbreak() and **echo()**	See caution below.	
nl()	**<CR>** to **<NL>**	
nonl()		**<CR>** to **<NL>**
raw() (instead of **cbreak()**)		break, quit stripping

Figure 12-6: Input Option Settings for **curses** Programs

Do not use the combination **nocbreak()** and **noecho()**. If you use it in a program and also use **getch()**, the program will go in and out of **cbreak()** mode to get each character. Depending on the state of the tty driver when each character is typed, the program may produce undesirable output.

In addition to the routines noted in Figure 12-6, you can use the **curses** routines **noraw()**, **halfdelay()**, and **nodelay()** to control input. See the **curses**(3X) manual page for discussions of these routines.

The next few pages describe **noecho()**, **cbreak()** and the related routines **echo()** and **nocbreak()** in more detail.

echo() and noecho()

SYNOPSIS
#include <curses.h>

int echo()
int noecho()

NOTES

- **echo()** turns on echoing of characters by **curses** as they are read in. This is the initial setting.
- **noecho()** turns off the echoing.
- Neither returns any useful value.
- **curses** programs may not run properly if you turn on echoing with **nocbreak()**. See Figure 12-6 and accompanying caution. After you turn echoing off, you can still echo characters with **addch()**.

EXAMPLE

See the **editor** and **show** programs under "**curses** Program Examples" in this chapter.

cbreak() and nocbreak()

SYNOPSIS
#include < curses.h >
int cbreak()
int nocbreak()

NOTES

- **cbreak()** turns on 'break for each character' processing. A program gets each character as soon as it is typed, but the erase, line kill, and CTRL-D characters are not interpreted.
- **nocbreak()** returns to normal 'line at a time' processing. This is typically the initial setting.
- Neither returns any useful value.
- A **curses** program may not run properly if **cbreak()** is turned on and off within the same program or if the combination **nocbreak()** and **echo()** is used.
- See Figure 12-6 and accompanying caution.

EXAMPLE

See the **editor** and **show** programs under "**curses** Program Examples" in this chapter.

Building Windows and Pads

An earlier section in this chapter, "More about **refresh**() and Windows" explained what windows and pads are and why you might want to use them. This section describes the **curses** routines you use to manipulate and create windows and pads.

Output and Input

The routines that you use to send output to and get input from windows and pads are similar to those you use with **stdscr**. The only difference is that you have to give the name of the window to receive the action. Generally, these functions have names formed by putting the letter **w** at the beginning of the name of a **stdscr** routine and adding the window name as the first parameter. For example, **addch**('c') would become **waddch(mywin,** '**c**') if you wanted to write the character **c** to the window **mywin**. Here's a list of the window (or **w**) versions of the output routines discussed in "Getting Simple Output and Input."

- **waddch**(*win, ch*)
- **mvwaddch**(*win, y, x, ch*)
- **waddstr**(*win, str*)
- **mvwaddstr**(*win, y, x, str*)
- **wprintw**(*win, fmt [, arg...]*)
- **mvwprintw**(*win, y, x, fmt [, arg...]*)
- **wmove**(*win, y, x*)
- **wclear**(*win*) and **werase**(*win*)
- **wclrtoeol**(*win*) and **wclrtobot**(*win*)
- **wrefresh**()

You can see from their declarations that these routines differ from the versions that manipulate **stdscr** only in their names and the addition of a *win* argument. Notice that the routines whose names begin with **mvw** take the *win* argument before the *y, x* coordinates, which is contrary to what the names imply. See **curses**(3X) for more information about these routines or the versions of the input routines **getch, getstr**(), and so on that you should use with windows.

All **w** routines can be used with pads except for **wrefresh**() and **wnoutrefresh**() (see below). In place of these two routines, you have to use **prefresh**() and **pnoutrefresh**() with pads.

The Routines wnoutrefresh() and doupdate()

If you recall from the earlier discussion about **refresh**(), we said that it sends the output from **stdscr** to the terminal screen. We also said that it was a macro that expands to **wrefresh(stdscr)** (see "What Every **curses** Program Needs" and "More about **refresh**() and Windows").

The **wrefresh**() routine is used to send the contents of a window (**stdscr** or one that you create) to a screen; it calls the routines **wnoutrefresh**() and **doupdate**(). Similarly, **prefresh**() sends the contents of a pad to a screen by calling **pnoutrefresh**() and **doupdate**().

Using **wnoutrefresh()**—or **pnoutrefresh()** (this discussion will be limited to the former routine for simplicity)—and **doupdate()**, you can update terminal screens with more efficiency than using **wrefresh()** by itself. **wrefresh()** works by first calling **wnoutrefresh()**, which copies the named window to a data structure referred to as the virtual screen. The virtual screen contains what a program intends to display at a terminal. After calling **wnoutrefresh()**, **wrefresh()** then calls **doupdate()**, which compares the virtual screen to the physical screen and does the actual update. If you want to output several windows at once, calling **wrefresh()** will result in alternating calls to **wnoutrefresh()** and **doupdate()**, causing several bursts of output to a screen. However, by calling **wnoutrefresh()** for each window and then **doupdate()** only once, you can minimize the total number of characters transmitted and the processor time used. The following sample program uses only one **doupdate()**:

```
#include <curses.h>

main()
{
   WINDOW *w1, *w2;

   initscr();
   w1 = newwin(2,6,0,3);
   w2 = newwin(1,4,5,4);
   waddstr(w1, "Bulls");
   wnoutrefresh(w1);
   waddstr(w2, "Eye");
   wnoutrefresh(w2);
   doupdate();
   endwin();
}
```

Notice from the sample that you declare a new window at the beginning of a **curses** program. The lines

```
w1 = newwin(2,6,0,3);
w2 = newwin(1,4,5,4);
```

declare two windows named `w1` and `w2` with the routine **newwin()** according to certain specifications. **newwin()** is discussed in more detail below.

Figure 12-7 illustrates the effect of **wnoutrefresh()** and **doupdate()** on these two windows, the virtual screen, and the physical screen:

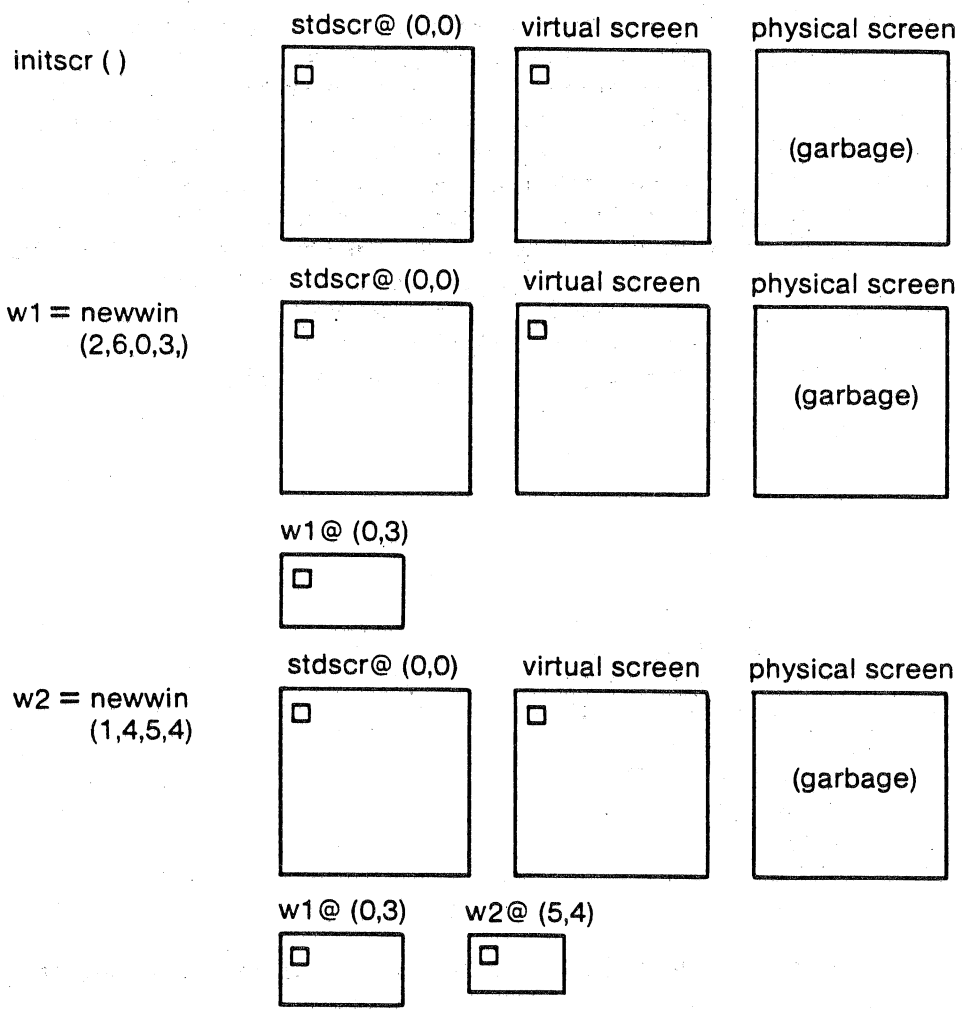


Figure 12-7: The Relationship Between a Window and a Terminal Screen (Sheet 1 of 3)

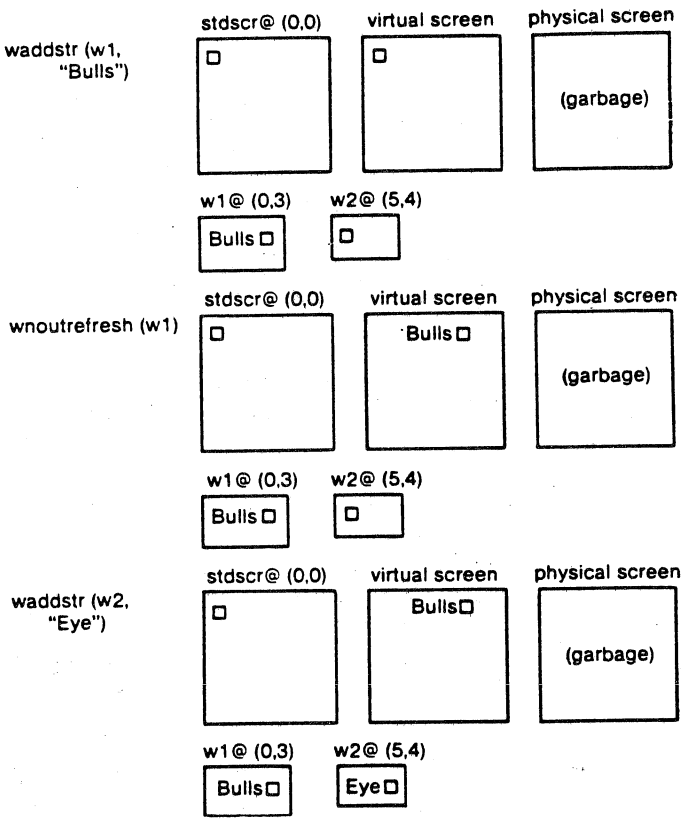


Figure 12-7: The Relationship Between a Window and a Terminal Screen (Sheet 2 of 3)

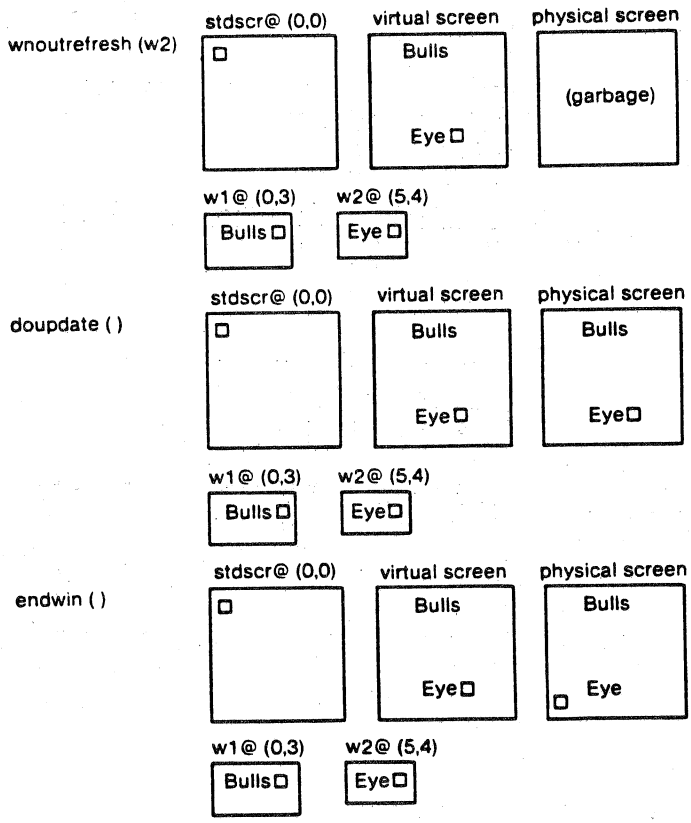


Figure 12-7: The Relationship Between a Window and a Terminal Screen (Sheet 3 of 3)

New Windows

Following are descriptions of the routines **newwin()** and **subwin()**, which you use to create new windows. For information about creating new pads with **newpad()** and **subpad()**, see the **curses**(3X) manual page.

newwin()

SYNOPSIS

#include <curses.h>

WINDOW *newwin(nlines, ncols, begin_y, begin_x)
int nlines, ncols, begin_y, begin_x;

NOTES

- **newwin()** returns a pointer to a new window with a new data area.
- The variables **nlines** and **ncols** give the size of the new window.
- **begin_y** and **begin_x** give the screen coordinates from (0,0) of the upper left corner of the window as it is refreshed to the current screen.

EXAMPLE

Recall the sample program using two windows; see Figure 12-7. Also see the **window** program under "**curses** Program Examples" in this chapter.

subwin()

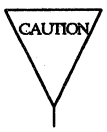
SYNOPSIS

#include <curses.h>

WINDOW *subwin(orig, nlines, ncols, begin_y, begin_x)
WINDOW *orig;
int nlines, ncols, begin_y, begin_x;

NOTES

- **subwin()** returns a new window that points to a section of another window, **orig**.
- **nlines** and **ncols** give the size of the new window.
- **begin_y** and **begin_x** give the screen coordinates of the upper left corner of the window as it is refreshed to the current screen.
- Subwindows and original windows can accidentally overwrite one another.



Subwindows of subwindows do not work (as of the copyright date of this *Programmer's Guide*).

EXAMPLE

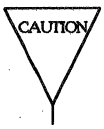
```
#include <curses.h>

main()
{
 WINDOW *sub;

  initscr();
  box(stdscr,'w','w');      /* See the curses(3X) manual */
                            /* page for box() */
  mvwaddstr(stdscr,7,10,"------- this is 10,10");
  mvwaddch(stdscr,8,10,'|');
  mvwaddch(stdscr,9,10,'v');
  sub = subwin(stdscr,10,20,10,10);
  box(sub,'s','s');
  wnoutrefresh(stdscr);
  wrefresh(sub);
  endwin();
}
```

This program prints a border of ws around the stdscr (the sides of your terminal screen) and a border of s's around the subwindow **sub** when it is run. For another example, see the **window** program under "**curses** Program Examples" in this chapter.

Using Advanced curses Features

Knowing how to use the basic **curses** routines to get output and input and to work with windows, you can design screen management programs that meet the needs of many users. The **curses** library, however, has routines that let you do more in a program than handle I/O and multiple windows. The following few pages briefly describe some of these routines and what they can help you do—namely, draw simple graphics, use a terminal's soft labels, and work with more than one terminal in a single **curses** program.

You should be comfortable using the routines previously discussed in this chapter and the other routines for I/O and window manipulation discussed on the **curses**(3X) manual page before you try to use the advanced **curses** features.



The routines described under "Routines for Drawing Lines and Other Graphics" and "Routines for Using Soft Labels" are features that are new for UNIX System V Release 3.0. If a program uses any of these routines, it may not run on earlier releases of the UNIX system. You must use the Release 3.0 version of the **curses** library on UNIX System V Release 3.0 to work with these routines.

Routines for Drawing Lines and Other Graphics

Many terminals have an alternate character set for drawing simple graphics (or glyphs or graphic symbols). You can use this character set in **curses** programs. **curses** use the same names for glyphs as the VT100 line drawing character set.

To use the alternate character set in a **curses** program, you pass a set of variables whose names begin with ACS_ to the **curses** routine **waddch**() or a related routine. For example, ACS_ULCORNER is the variable for the upper left corner glyph. If a terminal has a line drawing character for this glyph, ACS_ULCORNER's value is the terminal's character for that glyph OR'd (|) with the bit-mask A_ALTCHARSET.

If no line drawing character is available for that glyph, a standard ASCII character that approximates the glyph is stored in its place. For example, the default character for ACS_HLINE, a horizontal line, is a – (minus sign). When a close approximation is not available, a + (plus sign) is used. All the standard ACS_ names and their defaults are listed on the **curses**(3X) manual page.

Part of an example program that uses line drawing characters follows. The example uses the **curses** routine **box**() to draw a box around a menu on a screen. **box**() uses the line drawing characters by default or when | (the pipe) and – are chosen. (See **curses**(3X).) Up and down more indicators are drawn on the box border (using **ACS_UARROW** and **ACS_DARROW**) if the menu contained within the box continues above or below the screen:

```
box(menuwin, ACS_VLINE, ACS_HLINE);
 ...

/* output the up/down arrows */
wmove(menuwin, maxy, maxx - 5);

/* output up arrow or horizontal line */
if (moreabove)
   waddch(menuwin, ACS_UARROW);
else
   addch(menuwin, ACS_HLINE);

/*output down arrow or horizontal line */
if (morebelow)
   waddch(menuwin, ACS_DARROW);
else
   waddch(menuwin, ACS_HLINE);
```

Here's another example. Because a default down arrow (like the lowercase letter v) isn't very discernible on a screen with many lowercase characters on it, you can change it to an uppercase V.

```
if ( ! (ACS_DARROW & A_ALTCHARSET))
   ACS_DARROW = 'V';
```

For more information, see **curses**(3X) in the *Programmer's Reference Manual.*

Routines for Using Soft Labels

Another feature available on most terminals is a set of soft labels across the bottom of their screens. A terminal's soft labels are usually matched with a set of hard function keys on the keyboard. There are usually eight of these labels, each of which is usually eight characters wide and one or two lines high.

The **curses** library has routines that provide a uniform model of eight soft labels on the screen. If a terminal does not have soft labels, the bottom line of its screen is converted into a soft label area. It is not necessary for the keyboard to have hard function keys to match the soft labels for a **curses** program to make use of them.

Let's briefly discuss most of the **curses** routines needed to use soft labels: **slk_init**(), **slk_set**(), **slk_refresh**() and **slk_noutrefresh**(), **slk_clear**, and **slk_restore**.

When you use soft labels in a **curses** program, you have to call the routine **slk_int()** before **initscr()**. This sets an internal flag for **initscr()** to look at that says to use the soft labels. If **initscr()** discovers that there are fewer than eight soft labels on the screen, that they are smaller than eight characters in size, or that there is no way to program them, then it will remove a line from the bottom of **stdscr** to use for the soft labels. The size of **stdscr** and the **LINES** variable will be reduced by 1 to reflect this change. A properly written program, one that is written to use the **LINES** and **COLS** variables, will continue to run as if the line had never existed on the screen.

slk_init() takes a single argument. It determines how the labels are grouped on the screen should a line get removed from **stdscr**. The choices are between a 3-2-3 arrangement as appears on AT&T terminals, or a 4-4 arrangement as appears on Hewlett-Packard terminals. The **curses** routines adjust the width and placement of the labels to maintain the pattern. The widest label generated is eight characters.

The routine **slk_set()** takes three arguments, the label number (1-8), the string to go on the label (up to eight characters), and the justification within the label (0 = left justified, 1 = centered, and 2 = right justified).

The routine **slk_noutrefresh()** is comparable to **wnoutrefresh()** in that it copies the label information onto the internal screen image, but it does not cause the screen to be updated. Since a **wrefresh()** commonly follows, **slk_noutrefresh()** is the function that is most commonly used to output the labels.

Just as **wrefresh()** is equivalent to a **wnoutrefresh()** followed by a **doupdate()**, so too the function **slk_refresh()** is equivalent to a **slk_noutrefresh()** followed by a **doupdate()**.

To prevent the soft labels from getting in the way of a shell escape, **slk_clear()** may be called before doing the **endwin()**. This clears the soft labels off the screen and does a **doupdate()**. The function **slk_restore()** may be used to restore them to the screen. See the **curses**(3X) manual page for more information about the routines for using soft labels.

Working with More than One Terminal

A **curses** program can produce output on more than one terminal at the same time. This is useful for single process programs that access a common database, such as multi-player games.

Writing programs that output to multiple terminals is a difficult business, and the **curses** library does not solve all the problems you might encounter. For instance, the programs—not the library routines—must determine the file name of each terminal line, and what kind of terminal is on each of those lines. The standard method, checking **$TERM** in the environment, does not work, because each process can only examine its own environment.

Another problem you might face is that of multiple programs reading from one line. This situation produces a race condition and should be avoided. However, a program trying to take over another terminal cannot just shut off whatever program is currently running on that line. (Usually, security reasons would also make this inappropriate. But, for some applications, such as an inter-terminal communication program, or a program that takes over unused terminal lines, it would be appropriate.) A typical solution to this problem requires each user logged in on a line to run a program that notifies a master program that the user is interested in joining the master program and tells it the notification program's process ID, the name of the tty line, and the type of terminal being used. Then the program goes to sleep until the master program finishes. When done, the master program wakes up the notification program

and all programs exit.

A **curses** program handles multiple terminals by always having a current terminal. All function calls always affect the current terminal. The master program should set up each terminal, saving a reference to the terminals in its own variables. When it wishes to affect a terminal, it should set the current terminal as desired, and then call ordinary **curses** routines.

References to terminals in a **curses** program have the type **SCREEN***. A new terminal is initialized by calling **newterm**(*type, outfd, infd*). **newterm** returns a screen reference to the terminal being set up. *type* is a character string, naming the kind of terminal being used. *outfd* is a **stdio**(3S) file pointer (**FILE***) used for output to the terminal and *infd* a file pointer for input from the terminal. This call replaces the normal call to **initscr**(), which calls **newterm(getenv("TERM"), stdout, stdin)**.

To change the current terminal, call **set_term**(*sp*) where *sp* is the screen reference to be made current. **set_term**() returns a reference to the previous terminal.

It is important to realize that each terminal has its own set of windows and options. Each terminal must be initialized separately with **newterm**(). Options such as **cbreak**() and **noecho**() must be set separately for each terminal. The functions **endwin**() and **refresh**() must be called separately for each terminal. Figure 12-8 shows a typical scenario to output a message to several terminals.

```
for (i=0; i<nterm; i++)
{
   set_term(terms[i]);
   mvaddstr(0, 0, "Important message");
   refresh();
}
```

Figure 12-8: Sending a Message to Several Terminals

See the **two** program under "**curses** Program Examples" in this chapter for a more complete example.

Working with terminfo Routines

Some programs need to use lower level routines (i.e., primitives) than those offered by the **curses** routines. For such programs, the **terminfo** routines are offered. They do not manage your terminal screen, but rather give you access to strings and capabilities which you can use yourself to manipulate the terminal.

There are three circumstances when it is proper to use **terminfo** routines. The first is when you need only some screen management capabilities, for example, making text standout on a screen. The second is when writing a filter. A typical filter does one transformation on an input stream without clearing the screen or addressing the cursor. If this transformation is terminal dependent and clearing the screen is inappropriate, use of the **terminfo** routines is worthwhile. The third is when you are writing a special purpose tool that sends a special purpose string to the terminal, such as programming a function key, setting tab stops, sending output to a printer port, or dealing with the status line. Otherwise, you are discouraged from using these routines: the higher level **curses** routines make your program more portable to other UNIX systems and to a wider class of terminals.

NOTE

You are discouraged from using **terminfo** routines except for the purposes noted, because **curses** routines take care of all the glitches present in physical terminals. When you use the **terminfo** routines, you must deal with the glitches yourself. Also, these routines may change and be incompatible with previous releases.

What Every terminfo Program Needs

A **terminfo** program typically includes the header files and routines shown in Figure 12-9.

```
#include <curses.h>
#include <term.h>
 ...
   setupterm( (char*)0, 1, (int*)0 );
    ...
   putp(clear_screen);
    ...
   reset_shell_mode();
   exit(0);
```

Figure 12-9: Typical Framework of a **terminfo** Program

The header files **<curses.h>** and **<term.h>** are required because they contain the definitions of the strings, numbers, and flags used by the **terminfo** routines. **setupterm()** takes care of initialization. Passing this routine the values **(char*)0, 1,** and **(int*)0** invokes reasonable defaults. If **setupterm()** can't figure out what kind of terminal you are on, it prints an error message and exits. **reset_shell_mode()** performs functions similar to **endwin()** and should be called before a **terminfo** program exits.

A global variable like **clear_screen** is defined by the call to **setupterm()**. It can be output using the **terminfo** routines **putp()** or **tputs()**, which gives a user more control. This string should not be directly output to the terminal using the C library routine **printf**(3S), because it contains padding information. A program that directly outputs strings will fail on terminals that require padding or that use the **xon/xoff** flow control protocol.

At the **terminfo** level, the higher level routines like **addch()** and **getch()** are not available. It is up to you to output whatever is needed. For a list of capabilities and a description of what they do, see **terminfo**(4); see **curses**(3X) for a list of all the **terminfo** routines.

Compiling and Running a terminfo Program

The general command line for compiling and the guidelines for running a program with **terminfo** routines are the same as those for compiling any other **curses** program. See the sections "Compiling a **curses** Program" and "Running a **curses** Program" in this chapter for more information.

An Example terminfo Program

The example program **termhl** shows a simple use of **terminfo** routines. It is a version of the **highlight** program (see "**curses** Program Examples") that does not use the higher level **curses** routines. **termhl** can be used as a filter. It includes the strings to enter bold and underline mode and to turn off all attributes.

```
/*
 * A terminfo level version of the highlight program.
 */


#include <curses.h>
#include <term.h>

int ulmode = 0;                 /* Currently underlining */

main(argc, argv)
  int argc;
  char **argv;
{
   FILE *fd;
   int c, c2;
   int outch();

   if (argc > 2)
   {
      fprintf(stderr, "Usage: termhl [file]\n");
      exit(1);
   }

   if (argc == 2)
   {
      fd = fopen(argv[1], "r");
      if (fd == NULL)
      {
         perror(argv[1]);
         exit(2);
      }
   }
   else
   {
      fd = stdin;
   }
   setupterm((char*)0, 1, (int*)0);

   for (;;)
   {
      c = getc(fd);
      if (c == EOF)
      break;
      if (c == '\')
      {
         c2 = getc(fd);
         switch (c2)
         {
            case 'B':
            tputs(enter_bold_mode, 1, outch);
            continue;
            case 'U':
            tputs(enter_underline_mode, 1, outch);
```

```
            ulmode = 1;
            continue;
            case 'N':
            tputs(exit_attribute_mode, 1, outch);
            ulmode = 0;
            continue;
         }
         putch(c);
         putch(c2);
      }
      else
         putch(c);
   }
   fclose(fd);
   fflush(stdout);
   resetterm();
   exit(0);
}

/*
 * This function is like putchar,
 * but it checks for underlining.
 */
putch(c)
   int c;
{
   outch(c);
   if (ulmode && underline_char)
   {
      outch('\b');
      tputs(underline_char, 1, outch);
   }
}

/*
 * Outchar is a function version of putchar that
 * can be passed to tputs as a routine to call.
 */
outch(c)
   int c;
{
   putchar(c);
}
```

Let's discuss the use of the function **tputs**(*cap, affcnt, outc*) in this program to gain some insight into the **terminfo** routines. **tputs**() applies padding information. Some terminals have the capability to delay output. Their terminal descriptions in the **terminfo** database probably contain strings like **$<20>**, which means to pad for 20 milliseconds (see the following section "Specify Capabilities" in this chapter). **tputs** generates enough pad characters to delay for the appropriate time.

tput() has three parameters. The first parameter is the string capability to be output. The second is the number of lines affected by the capability. (Some capabilities may require padding that depends on the number of lines affected. For example, **insert_line** may have to copy all lines below the current line, and may require time proportional to the number of lines copied. By convention *affcnt* is 1 if no lines are affected. The value 1 is used, rather than 0, for safety, since *affcnt* is multiplied by the amount of time per item, and anything multiplied by 0 is 0.) The third parameter is a routine to be called with each character.

For many simple programs, *affcnt* is always 1 and *outc* always calls **putchar**. For these programs, the routine **putp**(*cap*) is a convenient abbreviation. **termhl** could be simplified by using **putp()**.

Now to understand why you should use the **curses** level routines instead of **terminfo** level routines whenever possible, note the special check for the **underline_char** capability in this sample program. Some terminals, rather than having a code to start underlining and a code to stop underlining, have a code to underline the current character. **termhl** keeps track of the current mode, and if the current character is supposed to be underlined, outputs **underline_char**, if necessary. Low level details such as this are precisely why the **curses** level is recommended over the terminfo level. **curses** takes care of terminals with different methods of underlining and other terminal functions. Programs at the **terminfo** level must handle such details themselves.

termhl was written to illustrate a typical use of the **terminfo** routines. It is more complex than it need be in order to illustrate some properties of **terminfo** programs. The routine **vidattr** (see **curses**(3X)) could have been used instead of directly outputting **enter_bold_mode**, **enter_underline_mode**, and **exit_attribute_mode**. In fact, the program would be more robust if it did, since there are several ways to change video attribute modes.

Working with the terminfo Database

The **terminfo** database describes the many terminals with which **curses** programs, as well as some UNIX system tools, like **vi**(1), can be used. Each terminal description is a compiled file containing the names that the terminal is known by and a group of comma-separated fields describing the actions and capabilities of the terminal. This section describes the **terminfo** database, related support tools, and their relationship to the **curses** library.

Writing Terminal Descriptions

Descriptions of many popular terminals are already described in the **terminfo** database. However, it is possible that you'll want to run a **curses** program on a terminal for which there is not currently a description. In that case, you'll have to build the description.

The general procedure for building a terminal description is as follows:

1. Give the known names of the terminal.
2. Learn about, list, and define the known capabilities.
3. Compile the newly-created description entry.
4. Test the entry for correct operation.
5. Go back to step 2, add more capabilities, and repeat, as necessary.

Building a terminal description is sometimes easier when you build small parts of the description and test them as you go along. These tests can expose deficiencies in the ability to describe the terminal. Also, modifying an existing description of a similar terminal can make the building task easier. (Lest we forget the UNIX motto: Build on the work of others.)

In the next few pages, we follow each step required to build a terminal description for the fictitious terminal named "myterm."

Name the Terminal

The name of a terminal is the first information given in a **terminfo** terminal description. This string of names, assuming there is more than one name, is separated by pipe symbols (|). The first name given should be the most common abbreviation for the terminal. The last name given should be a long name that fully identifies the terminal. The long name is usually the manufacturer's formal name for the terminal. All names between the first and last entries should be known synonyms for the terminal name. All names but the formal name should be typed in lowercase letters and contain no blanks. Naturally, the formal name is entered as closely as possible to the manufacturer's name.

Here is the name string from the description of the AT&T Teletype 5420 Buffered Display Terminal:

```
5420|att5420|AT&T Teletype 5420,
```

Notice that the first name is the most commonly used abbreviation and the last is the long name. Also notice the comma at the end of the name string.

Here's the name string for our fictitious terminal, myterm:

```
myterm|mytm|mine|fancy|terminal|My FANCY Terminal,
```

Terminal names should follow common naming conventions. These conventions start with a root name, like 5425 or myterm, for example. The root name should not contain odd characters, like hyphens, that may not be recognized as a synonym for the terminal name. Possible hardware modes or user preferences should be shown by adding a hyphen and a 'mode indicator' at the end of the name. For example, the 'wide mode' (which is shown by a **-w**) version of our fictitious terminal would be described as **myterm-w**. **term**(5) describes mode indicators in greater detail.

Learn About the Capabilities

After you complete the string of terminal names for your description, you have to learn about the terminal's capabilities so that you can properly describe them. To learn about the capabilities your terminal has, you should do the following:

- See the owner's manual for your terminal. It should have information about the capabilities available and the character strings that make up the sequence transmitted from the keyboard for each capability.
- Test the keys on your terminal to see what they transmit, if this information is not available in the manual. You can test the keys in one of the following ways — type:

  stty -echo; cat -vu
  Type in the keys you want to test;
  for example, see what right arrow (→) transmits.
  <CR>
  <CTRL-D>
  stty echo

  or

  cat >dev/null
  Type in the escape sequences you want to test;
  for example, see what **\E [H** transmits.
  <CTRL-D>

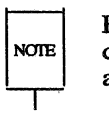
- The first line in each of these testing methods sets up the terminal to carry out the tests. The **<CTRL-D>** helps return the terminal to its normal settings.
- See the **terminfo**(4) manual page. It lists all the capability names you have to use in a terminal description. The following section, "Specify Capabilities," gives details.

Specify Capabilities

Once you know the capabilities of your terminal, you have to describe them in your terminal description. You describe them with a string of comma-separated fields that contain the abbreviated **terminfo** name and, in some cases, the terminal's value for each capability. For example, **bel** is the abbreviated name for the beeping or ringing capability. On most terminals, a CTRL-G is the instruction that produces a beeping sound. Therefore, the beeping capability would be shown in the terminal description as **bel=^G,**.

The list of capabilities may continue onto multiple lines as long as white space (that is, tabs and spaces) begins every line but the first of the description. Comments can be included in the description by putting a **#** at the beginning of the line.

The **terminfo**(4) manual page has a complete list of the capabilities you can use in a terminal description. This list contains the name of the capability, the abbreviated name used in the database, the two-letter code that corresponds to the old **termcap** database name, and a short description of the capability. The abbreviated name that you will use in your database descriptions is shown in the column titled "Capname."



For a **curses** program to run on any given terminal, its description in the **terminfo** database must include, at least, the capabilities to move a cursor in all four directions and to clear the screen.

A terminal's character sequence (value) for a capability can be a keyed operation (like CTRL-G), a numeric value, or a parameter string containing the sequence of operations required to achieve the particular capability. In a terminal description, certain characters are used after the capability name to show what type of character sequence is required. Explanations of these characters follow:

This shows a numeric value is to follow. This character follows a capability that needs a number as a value. For example, the number of columns is defined as **cols#80,**.

= This shows that the capability value is the character string that follows. This string instructs the terminal how to act and may actually be a sequence of commands. There are certain characters used in the instruction strings that have special meanings. These special characters follow:

^ This shows a control character is to be used. For example, the beeping sound is produced by a CTRL-G. This would be shown as **^G**.

\E or \e
These characters followed by another character show an escape instruction. An entry of \EC would transmit to the terminal as ESCAPE-C.

\n These characters provide a **<NL>** character sequence.

\l These characters provide a linefeed character sequence.

\r These characters provide a return character sequence.

\t These characters provide a tab character sequence.

\b These characters provide a backspace character sequence.

\f These characters provide a formfeed character sequence.

\s These characters provide a space character sequence.

nnn This is a character whose three-digit octal is *nnn*, where *nnn* can be one to three digits.

$< >** These symbols are used to show a delay in milliseconds. The desired length of delay is enclosed inside the "less than/greater than" symbols (< >). The amount of delay may be a whole number, a numeric value to one decimal place (tenths), or either form followed by an asterisk (*). The * shows that the delay will be proportional to the number of lines affected by the operation. For example, a 20-millisecond delay per line would appear as **$<20*>. See the **terminfo**(4) manual page for

more information about delays and padding.

Sometimes, it may be necessary to comment out a capability so that the terminal ignores this particular field. This is done by placing a period (.) in front of the abbreviated name for the capability. For example, if you would like to comment out the beeping capability, the description entry would appear as

.bel=^G,

With this background information about specifying capabilities, let's add the capability string to our description of myterm. We'll consider basic, screen-oriented, keyboard-entered, and parameter string capabilities.

Basic Capabilities

Some capabilities common to most terminals are bells, columns, lines on the screen, and overstriking of characters, if necessary. Suppose our fictitious terminal has these and a few other capabilities, as listed below. Note that the list gives the abbreviated **terminfo** name for each capability in the parentheses following the capability description:

- An automatic wrap around to the beginning of the next line whenever the cursor reaches the right-hand margin (**am**).
- The ability to produce a beeping sound. The instruction required to produce the beeping sound is ^G (**bel**).
- An 80-column wide screen (**cols**).
- A 30-line long screen (**lines**).
- Use of xon/xoff protocol (**xon**).

By combining the name string (see the section "Name the Terminal") and the capability descriptions that we now have, we get the following general **terminfo** database entry:

```
myterm|mytm|mine|fancy|terminal|My FANCY terminal,
        am, bel=^G, cols#80, lines#30, xon,
```

Screen-Oriented Capabilities

Screen-oriented capabilities manipulate the contents of a screen. Our example terminal myterm has the following screen-oriented capabilities. Again, the abbreviated command associated with the given capability is shown in parentheses.

- A **<CR>** is a CTRL-M (**cr**).
- A cursor up one line motion is a CTRL-K (**cuu1**).
- A cursor down one line motion is a CTRL-J (**cud1**).
- Moving the cursor to the left one space is a CTRL-H (**cub1**).
- Moving the cursor to the right one space is a CTRL-L (**cuf1**).
- Entering reverse video mode is an ESCAPE-D (**smso**).
- Exiting reverse video mode is an ESCAPE-Z (**rmso**).

- A clear to the end of a line sequence is an ESCAPE-K and should have a 3-millisecond delay (**el**).
- A terminal scrolls when receiving a **<NL>** at the bottom of a page (**ind**).

The revised terminal description for myterm including these screen-oriented capabilities follows:

```
myterm|mytm|mine|fancy|terminal|My FANCY Terminal,
        am, bel=^G, cols#80, lines#30, xon,
        cr=^M, cuu1=^K, cud1=^J, cub1=^H, cuf1=^L,
        smso=\ED, rmso=\EZ, el=\EK$<3>, ind=\n,
```

Keyboard-Entered Capabilities

Keyboard-entered capabilities are sequences generated when a key is typed on a terminal keyboard. Most terminals have, at least, a few special keys on their keyboard, such as arrow keys and the backspace key. Our example terminal has several of these keys whose sequences are, as follows:

- The backspace key generates a CTRL-H (**kbs**).
- The up arrow key generates an ESCAPE-[A (**kcuu1**).
- The down arrow key generates an ESCAPE-[B (**kcud1**).
- The right arrow key generates an ESCAPE-[C (**kcuf1**).
- The left arrow key generates an ESCAPE-[D (**kcub1**).
- The home key generates an ESCAPE-[H (**khome**).

Adding this new information to our database entry for myterm produces:

```
myterm|mytm|mine|fancy|terminal|My FANCY Terminal,
        am, bel=^G, cols#80, lines#30, xon,
        cr=^M, cuu1=^K, cud1=^J, cub1=^H, cuf1=^L,
        smso=\ED, rmso=\EZ, el=\EK$<3>, ind=0
        kbs=^H, kcuu1=\E[A, kcud1=\E[B, kcuf1=\E[C,
        kcub1=\E[D, khome=\E[H,
```

Parameter String Capabilities

Parameter string capabilities are capabilities that can take parameters — for example, those used to position a cursor on a screen or turn on a combination of video modes. To address a cursor, the **cup** capability is used and is passed two parameters: the row and column to address. String capabilities, such as **cup** and set attributes (**sgr**) capabilities, are passed arguments in a **terminfo** program by the **tparm**() routine.

The arguments to string capabilities are manipulated with special '%' sequences similar to those found in a **printf**(3S) statement. In addition, many of the features found on a simple stack-based RPN calculator are available. **cup**, as noted above, takes two arguments: the row and column. **sgr**, takes nine arguments, one for each of the nine video attributes. See **terminfo**(4) for the list and order of the attributes and further examples of **sgr**.

Our fancy terminal's cursor position sequence requires a row and column to be output as numbers separated by a semicolon, preceded by ESCAPE-[and followed with H. The coordinate numbers are 1-based rather than 0-based. Thus, to move to row 5, column 18, from (0,0), the sequence

Integer arguments are pushed onto the stack with a '%p' sequence followed by the argument number, such as '%p2' to push the second argument. A shorthand sequence to increment the first two arguments is '%i'. To output the top number on the stack as a decimal, a '%d' sequence is used, exactly as in **printf**. Our terminal's **cup** sequence is built up as follows:

cup=	Meaning
\E[	output ESCAPE-[
%i	increment the two arguments
%p1	push the 1st argument (the row) onto the stack
%d	output the row as a decimal
;	output a semi-colon
%p2	push the 2nd argument (the column) onto the stack
%d	output the column as a decimal
H	output the trailing letter

or

```
cup=\E[%i%p1%d;%p2%dH,
```

Adding this new information to our database entry for myterm produces:

```
myterm|mytm|mine|fancy|terminal|My FANCY Terminal,
    am, bel=^G, cols#80, lines#30, xon,
    cr=^M, cuu1=^K, cud1=^J, cub1=^H, cuf1=^L,
    smso=\ED, rmso=\EZ, el=\EK$<3>, ind=0
    kbs=^H, kcuu1=\E[A, kcud1=\E[B, kcuf1=\E[C,
    kcub1=\E[D, khome=\E[H,
    cup=\E[%i%p1%d;%p2%dH,
```

See **terminfo**(4) for more information about parameter string capabilities.

Compile the Description

The **terminfo** database entries are compiled using the **tic** compiler. This compiler translates **terminfo** database entries from the source format into the compiled format.

The source file for the description is usually in a file suffixed with **.ti**. For example, the description of myterm would be in a source file named **myterm.ti**. The compiled description of myterm would usually be placed in **/usr/lib/terminfo/m/myterm**, since the first letter in the description entry is **m**. Links would also be made to synonyms of **myterm**, for example, to **/f/fancy**. If the environment variable **$TERMINFO** were set to a directory and exported before the entry was compiled, the compiled entry would be placed in the **$TERMINFO** directory. All programs using the entry would then look in the new directory for the description file if **$TERMINFO** were set, before looking in the default **/usr/lib/terminfo**. The general format for the **tic** compiler is as follows:

tic [**−v**] [**−c**] *file*

The **-v** option causes the compiler to trace its actions and output information about its progress. The **-c** option causes a check for errors; it may be combined with the **-v** option. *file* shows what file is to be compiled. If you want to compile more than one file at the same time, you have to first use **cat**(1) to join them together. The following command line shows how to compile the **terminfo** source file for our fictitious terminal:

tic -v myterm.ti<CR>
(The trace information appears as the compilation proceeds.)

Refer to the **tic**(1M) manual page in the *System Administrator's Reference Manual* for more information about the compiler.

Test the Description

Let's consider three ways to test a terminal description. First, you can test it by setting the environment variable **$TERMINFO** to the path name of the directory containing the description. If programs run the same on the new terminal as they did on the older known terminals, then the new description is functional.

Second, you can test for correct insert line padding by commenting out **xon** in the description and then editing (using **vi**(1)) a large file (over 100 lines) at 9600 baud (if possible), and deleting about 15 lines from the middle of the screen. Type **u** (undo) several times quickly. If the terminal messes up, then more padding is usually required. A similar test can be used for inserting a character.

Third, you can use the **tput**(1) command. This command outputs a string or an integer according to the type of capability being described. If the capability is a Boolean expression, then **tput** sets the exit code (0 for TRUE, 1 for FALSE) and produces no output. The general format for the **tput** command is as follows:

tput [**-T***type*] *capname*

The type of terminal you are requesting information about is identified with the **-T***type* option. Usually, this option is not necessary because the default terminal name is taken from the environment variable **$TERM**. The *capname* field is used to show what capability to output from the **terminfo** database.

The following command line shows how to output the "clear screen" character sequence for the terminal being used:

tput clear
(The screen is cleared.)

The following command line shows how to output the number of columns for the terminal being used:

tput cols
(The number of columns used by the terminal appears here.)

The **tput**(1) manual page found in the *User's Reference Manual* contains more information on the usage and possible messages associated with this command.

Comparing or Printing terminfo Descriptions

Sometime you may want to compare two terminal descriptions or quickly look at a description without going to the **terminfo** source directory. The **infocmp**(1M) command was designed to help you with both of these tasks. Compare two descriptions of the same terminal; for example,

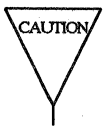
```
mkdir /tmp/old /tmp/new
TERMINFO=/tmp/old tic old5420.ti
TERMINFO=/tmp/new tic new5420.ti
infocmp -A /tmp/old -B /tmp/new -d 5420 5420
```

compares the old and new 5420 entries.

To print out the **terminfo** source for the 5420, type

```
infocmp -I 5420
```

Converting a termcap Description to a terminfo Description



The **terminfo** database is designed to take the place of the **termcap** database. Because of the many programs and processes that have been written with and for the **termcap** database, it is not feasible to do a complete cutover at one time. Any conversion from **termcap** to **terminfo** requires some experience with both databases. All entries into the databases should be handled with extreme caution. These files are important to the operation of your terminal.

The **captoinfo**(1M) command converts **termcap**(4) descriptions to **terminfo**(4) descriptions. When a file is passed to **captoinfo**, it looks for **termcap** descriptions and writes the equivalent **terminfo** descriptions on the standard output. For example,

```
captoinfo /etc/termcap
```

converts the file **/etc/termcap** to **terminfo** source, preserving comments and other extraneous information within the file. The command line

```
captoinfo
```

looks up the current terminal in the **termcap** database, as specified by the **$TERM** and **$TERMCAP** environment variables and converts it to **terminfo**.

If you must have both **termcap** and **terminfo** terminal descriptions, keep the **terminfo** description only and use **infocmp -C** to get the **termcap** descriptions.

If you have been using cursor optimization programs with the **-ltermcap** or **-ltermlib** option in the **cc** command line, those programs will still be functional. However, these options should be replaced with the **-lcurses** option.

curses Program Examples

The following examples demonstrate uses of **curses** routines.

The editor Program

This program illustrates how to use **curses** routines to write a screen editor. For simplicity, **editor** keeps the buffer in **stdscr**; obviously, a real screen editor would have a separate data structure for the buffer. This program has many other simplifications: no provision is made for files of any length other than the size of the screen, for lines longer than the width of the screen, or for control characters in the file.

Several points about this program are worth making. First, it uses the **move()**, **mvaddstr()**, **flash()**, **wnoutrefresh()** and **clrtoeol()** routines. These routines are all discussed in this chapter under "Working with **curses** Routines."

Second, it also uses some **curses** routines that we have not discussed. For example, the function to write out a file uses the **mvinch()** routine, which returns a character in a window at a given position. The data structure used to write out a file does not keep track of the number of characters in a line or the number of lines in the file, so trailing blanks are eliminated when the file is written. The program also uses the **insch()**, **delch()**, **insertln()**, and **deleteln()** routines. These functions insert and delete a character or line. See **curses**(3X) for more information about these routines.

Third, the editor command interpreter accepts special keys, as well as ASCII characters. On one hand, new users find an editor that handles special keys easier to learn about. For example, it's easier for new users to use the arrow keys to move a cursor than it is to memorize that the letter h means left, j means down, k means up, and l means right. On the other hand, experienced users usually like having the ASCII characters to avoid moving their hands from the home row position to use special keys.

Because not all terminals have arrow keys, your **curses** programs will work on more terminals if there is an ASCII character associated with each special key.

Fourth, the CTRL-L command illustrates a feature most programs using **curses** routines should have. Often some program beyond the control of the routines writes something to the screen (for instance, a broadcast message) or some line noise affects the screen so much that the routines cannot keep track of it. A user invoking **editor** can type CTRL-L, causing the screen to be cleared and redrawn with a call to **wrefresh(curscr)**.

Finally, another important point is that the input command is terminated by CTRL-D, not the escape key. It is very tempting to use escape as a command, since escape is one of the few special keys available on every keyboard. (Return and break are the only others.) However, using escape as a separate key introduces an ambiguity. Most terminals use sequences of characters beginning with escape (i.e., escape sequences) to control the terminal and have special keys that send escape sequences to the computer. If a computer receives an escape from a terminal, it cannot tell whether the user depressed the escape key or whether a special key was pressed.

editor and other **curses** programs handle the ambiguity by setting a timer. If another character is received during this time, and if that character might be the beginning of a special key, the program reads more input until either a full special key is read, the time out is reached, or a character is received that could not have been generated by a special key. While this strategy works most of the time, it is not foolproof. It is possible for the user to press escape, then to type another key quickly, which causes the **curses** program to think a special key has been pressed. Also, a pause occurs until the escape can be passed to the user program, resulting in a slower response to the escape key.

Many existing programs use escape as a fundamental command, which cannot be changed without infuriating a large class of users. These programs cannot make use of special keys without dealing with this ambiguity, and at best must resort to a time-out solution. The moral is clear: when designing your **curses** programs, avoid the escape key.

```
/* editor: A screen-oriented editor.  The user
 * interface is similar to a subset of vi.
 * The buffer is kept in stdscr to simplify
 * the program.
 */

#include <stdio.h>
#include <curses.h>

#define CTRL(c) ((c) & 037)

main(argc, argv)
int argc;
char **argv;
{
        extern void perror(), exit();
      int i, n, l;
      int c;
      int line = 0;
      FILE *fd;

      if (argc != 2)
      {
            fprintf(stderr, "Usage: %s file\n", argv[0]);
            exit(1);
      }

      fd = fopen(argv[1], "r");
      if (fd == NULL)
      {
            perror(argv[1]);
            exit(2);
      }

      initscr();
      cbreak();
      nonl();
      noecho();
      idlok(stdscr, TRUE);
```

```
        keypad(stdscr, TRUE);

        /* Read in the file */
        while ((c = getc(fd)) != EOF)
        {
                if (c == '\n')
                        line++;
                if (line > LINES - 2)
                        break;
                addch(c);
        }
        fclose(fd);

        move(0,0);
        refresh();
        edit();

        /* Write out the file */
        fd = fopen(argv[1], "w");
        for (l = 0; l < LINES - 1; l++)
        {
                n = len(l);
                for (i = 0; i < n; i++)
                        putc(mvinch(l, i) & A_CHARTEXT, fd);
                putc('\n', fd);
        }
        fclose(fd);

        endwin();
        exit(0);
}

len(lineno)
int lineno;
{
        int linelen = COLS - 1;

        while (linelen >= 0 && mvinch(lineno, linelen) == ' ')
                linelen--;
        return linelen + 1;
}

/* Global value of current cursor position */
int row, col;

edit()
{
        int c;

        for (;;)
        {
                move(row, col);
                refresh();
                c = getch();
```

```
/* Editor commands */
switch (c)
{

/* hjkl and arrow keys: move cursor
 * in direction indicated */
case 'h':
case KEY_LEFT:
      if (col > 0)
            col--;
      else
            flash();
      break;

case 'j':
case KEY_DOWN:
      if (row < LINES - 1)
            row++;
      else
            flash();
      break;

case 'k':
case KEY_UP:
      if (row > 0)
            row--;
      else
            flash();
      break;

case 'l':
case KEY_RIGHT:
      if (col < COLS - 1)
            col++;
      else
            flash();
      break;
/* i: enter input mode */
case KEY_IC:
case 'i':
      input();
      break;

/* x: delete current character */
case KEY_DC:
case 'x':
      delch();
      break;

/* o: open up a new line and enter input mode */
case KEY_IL:
case 'o':
      move(++row, col = 0);
      insertln();
      input();
```

```
                break;

        /* d: delete current line */
        case KEY_DL:
        case 'd':
                deleteln();
                break;

        /* ^L: redraw screen */
        case KEY_CLEAR:
        case CTRL('L'):
                wrefresh(curscr);
                break;

        /* w: write and quit */
        case 'w':
                return;
        /* q: quit without writing */
        case 'q':
                endwin();
                exit(2);
        default:
                flash();
                break;
        }
    }
}

/*
 * Insert mode: accept characters and insert them.
 *  End with ^D or EIC
 */
input()
{
    int c;

    standout();
    mvaddstr(LINES - 1, COLS - 20, "INPUT MODE");
    standend();
    move(row, col);
    refresh();
    for (;;)
    {
        c = getch();
        if (c == CTRL('D') || c == KEY_EIC)
            break;
        insch(c);
        move(row, ++col);
        refresh();
    }
    move(LINES - 1, COLS - 20);
    clrtoeol();
    move(row, col);
    refresh();
}
```

The highlight Program

This program illustrates a use of the routine **attrset()**. **highlight** reads a text file and uses embedded escape sequences to control attributes. **\U** turns on underlining, **\B** turns on bold, and **\N** restores the default output attributes.

Note the first call to **scrollok()**, a routine that we have not previously discussed (see **curses**(3X)). This routine allows the terminal to scroll if the file is longer than one screen. When an attempt is made to draw past the bottom of the screen, **scrollok()** automatically scrolls the terminal up a line and calls **refresh()**.

```
/*
 * highlight: a program to turn \U, \B, and
 * \N sequences into highlighted
 * output, allowing words to be
 * displayed underlined or in bold.
 */

#include <stdio.h>
#include <curses.h>

main(argc, argv)
int argc;
char **argv;
{
      FILE *fd;
      int c, c2;
      void exit(), perror();

      if (argc != 2)
      {
            fprintf(stderr, "Usage: highlight file\n");
            exit(1);
      }

      fd = fopen(argv[1], "r");

      if (fd == NULL)
      {
            perror(argv[1]);
            exit(2);
      }

      initscr();
      scrollok(stdscr, TRUE);
      nonl();
      while ((c = getc(fd)) != EOF)
      {
            if (c == '\\')
            {
                  c2 = getc(fd);
                  switch (c2)
                  {
```

```
                case 'B':
                        attrset(A_BOLD);
                        continue;
                case 'U':
                        attrset(A_UNDERLINE);
                        continue;
                case 'N':
                        attrset(0);
                        continue;
                }
                addch(c);
                addch(c2);
            }
            else
                addch(c);
        }
        fclose(fd);
        refresh();
        endwin();
        exit(0);
}
```

The scatter Program

This program takes the first **LINES − 1** lines of characters from the standard input and displays the characters on a terminal screen in a random order. For this program to work properly, the input file should not contain tabs or non-printing characters.

```
/*
 *    The scatter program.
 */

#include    <curses.h>
#include    <sys/types.h>

extern time_t time();

#define MAXLINES 120
#define MAXCOLS  160
char s[MAXLINES][MAXCOLS];    /* Screen Array */
int  T[MAXLINES][MAXCOLS];    /* Tag Array - Keeps track of    *
                               * the number of characters     *
                               * printed and their positions. */

main()
{
        register int row = 0,col = 0;
        register int c;
        int char_count = 0;
        time_t t;
        void exit(), srand();

        initscr();
```

```
        for(row = 0;row < MAXLINES;row++)
                for(col = 0;col < MAXCOLS;col++)
                        s[row][col]=' ';

        col = row = 0;
        /* Read screen in */
        while ((c=getchar()) != EOF && row < LINES ) {

                if(c != '\n')
                {
                        /* Place char in screen array */
                        s[row][col++] = c;
                        if(c != ' ')
                                char_count++;
                }
                else
                {
                        col = 0;
                        row++;
                }
        }

        time(&t);   /* Seed the random number generator */
        srand((unsigned)t);

        while (char_count)
        {
                row = rand() % LINES;
                col = (rand() >> 2) % COLS;
                if (T[row][col] != 1 && s[row][col] != ' ')
                {
                        move(row, col);
                        addch(s[row][col]);
                        T[row][col] = 1;
                        char_count--;
                        refresh();
                }
        }
        endwin();
        exit(0);
}
```

The show Program

show pages through a file, showing one screen of its contents each time you depress the space bar. The program calls **cbreak**() so that you can depress the space bar without having to hit return; it calls **noecho**() to prevent the space from echoing on the screen. The **nonl**() routine, which we have not previously discussed, is called to enable more cursor optimization. The **idlok**() routine, which we also have not discussed, is called to allow insert and delete line. (See **curses**(3X) for more information about these routines). Also notice that **clrtoeol**() and **clrtobot**() are called.

By creating an input file for **show** made up of screen-sized (about 24 lines) pages, each varying slightly from the previous page, nearly any exercise for a **curses()** program can be created. This type of input file is called a show script.

```
#include <curses.h>
#include <signal.h>

main(argc, argv)
int argc;
char *argv[];
{
      FILE *fd;
      char linebuf[BUFSIZ];
      int line;
      void done(), perror(), exit();

      if (argc != 2)
      {
            fprintf(stderr, "usage: %s file\n", argv[0]);
            exit(1);
      }

      if ((fd=fopen(argv[1], "r")) == NULL)
      {

            perror(argv[1]);
            exit(2);
      }
      signal(SIGINT, done);

      initscr();
      noecho();
      cbreak();
      nonl();
      idlok(stdscr, TRUE);

      while(1)
      {
            move(0,0);
            for (line = 0; line < LINES; line++)
            {
                  if (!fgets(linebuf, sizeof linebuf, fd))
                  {
                        clrtobot();
                        done();
                  }
                  move(line, 0);
                  printw("%s", linebuf);
            }
            refresh();
            if (getch() == 'q')
                  done();
      }
}
```

```
void done()
{
      move(LINES - 1, 0);
      clrtoeol();
      refresh();
      endwin();
      exit(0);
}
```

The two Program

This program pages through a file, writing one page to the terminal from which the program is invoked and the next page to the terminal named on the command line. It then waits for a space to be typed on either terminal and writes the next page to the terminal at which the space is typed.

two is just a simple example of a two-terminal **curses** program. It does not handle notification; instead, it requires the name and type of the second terminal on the command line. As written, the command "**sleep 100000**" must be typed at the second terminal to put it to sleep while the program runs, and the user of the first terminal must have both read and write permission on the second terminal.

```
#include <curses.h>
#include <signal.h>

SCREEN *me, *you;
SCREEN *set_term();

FILE *fd, *fdyou;
char linebuf[512];

main(argc, argv)
int argc;
char **argv;
{
   void done(), exit();
   unsigned sleep();
   char *getenv();
   int c;

   if (argc != 4)
   {
        fprintf(stderr, "Usage: two othertty otherttytype \
            inputfile\n");
        exit(1);
   }

   fd = fopen(argv[3], "r");
   fdyou = fopen(argv[1], "w+");
   signal(SIGINT, done);   /* die gracefully */

   me = newterm(getenv("TERM"), stdout, stdin);
            /* initialize my tty */
```

```
    you = newterm(argv[2], fdyou, fdyou);
             /* Initialize the other terminal */

    set_term(me);  /* Set modes for my terminal */
    noecho();    /* turn off tty echo */
    cbreak();    /* enter cbreak mode */
    nonl();      /* Allow linefeed */
    nodelay(stdscr, TRUE);  /* No hang on input */

    set_term(you); /* Set modes for other terminal */
    noecho();
    cbreak();
    nonl();
    nodelay(stdscr,TRUE);

    /* Dump first screen full on my terminal */
    dump_page(me);

    /* Dump second screen full on the other terminal */
    dump_page(you);

    for (;;) /* for each screen full */
    {
         set_term(me);
         c = getch();
         if (c == 'q')    /* wait for user to read it */
         done();
         if (c == ' ')
         dump_page(me);

         set_term(you);
         c = getch();
         if (c == 'q')    /* wait for user to read it */
         done();
         if (c == ' ')
         dump_page(you);
         sleep(1);
    }
}
dump_page(term)
  SCREEN *term;
{
    int line;

    set_term(term);
    move(0, 0);
    for (line = 0; line < LINES - 1; line++) {
         if (fgets(linebuf, sizeof linebuf, fd) == NULL) {
         clrtobot();
         done();
         }
         mvaddstr(line, 0, linebuf);
    }
    standout();
    mvprintw(LINES - 1, 0, "--More--");
```

```
    standend();
    refresh();  /* sync screen */
}
/*
 * Clean up and exit.
 */
void done()
{
    /* Clean up first terminal */
    set_term(you);
    move(LINES - 1,0);   /* to lower left corner */

    clrtoeol(); /* clear bottom line */
    refresh();  /* flush out everything */
    endwin();   /* curses cleanup */

    /* Clean up second terminal */
    set_term(me);
    move(LINES - 1,0);   /* to lower left corner */
    clrtoeol(); /* clear bottom line */
    refresh();  /* flush out everything */
    endwin();   /* curses cleanup */
    exit(0);
}
```

The window Program

This example program demonstrates the use of multiple windows. The main display is kept in **stdscr**. When you want to put something other than what is in **stdscr** on the physical terminal screen temporarily, a new window is created covering part of the screen. A call to **wrefresh()** for that window causes it to be written over the **stdscr** image on the terminal screen. Calling **refresh()** on **stdscr** results in the original window being redrawn on the screen. Note the calls to the **touchwin()** routine (which we have not discussed — see **curses**(3X)) that occur before writing out a window over an existing window on the terminal screen. This routine prevents screen optimization in a **curses** program. If you have trouble refreshing a new window that overlaps an old window, it may be necessary to call **touchwin()** for the new window to get it completely written out.

```
#include <curses.h>

WINDOW cursesmdwin;

main()

{

        int i, c;
        char buf[120];
        void exit();

        initscr();
        nonl();
        noecho();
```

```
        cbreak();

        cmdwin = newwin(3, COLS, 0, 0);/* top 3 lines */
        for (i = 0; i < LINES; i++)
              mvprintw(i, 0, "This is line %d of stdscr", i);

        for (;;)

        {
              refresh();
              c = getch();
              switch (c)

              {

              case 'c':   /* Enter command from keyboard */
                    werase(cmdwin);
                    wprintw(cmdwin, "Enter command:");
                    wmove(cmdwin, 2, 0);
                    for (i = 0; i < COLS; i++)
                          waddch(cmdwin, '-');
                    wmove(cmdwin, 1, 0);
                    touchwin(cmdwin);
                    wrefresh(cmdwin);
                    wgetstr(cmdwin, buf);
                    touchwin(stdscr);

                    /*
                     * The command is now in buf.
                     * It should be processed here.
                     */

              case 'q':
                    endwin();
                    exit(0);
              }

        }

}
```

Chapter 13: File and Record Locking

Introduction

Mandatory and advisory file and record locking both are available on current releases of the UNIX system. The intent of this capability to is provide a synchronization mechanism for programs accessing the same stores of data simultaneously. Such processing is characteristic of many multi-user applications, and the need for a standard method of dealing with the problem has been recognized by standards advocates like */usr/group*, an organization of UNIX system users from businesses and campuses across the country.

Advisory file and record locking can be used to coordinate self-synchronizing processes. In mandatory locking, the standard I/O subroutines and I/O system calls enforce the locking protocol. In this way, at the cost of a little efficiency, mandatory locking double checks the programs against accessing the data out of sequence.

The remainder of this chapter describes how file and record locking capabilities can be used. Examples are given for the correct use of record locking. Misconceptions about the amount of protection that record locking affords are dispelled. Record locking should be viewed as a synchronization mechanism, not a security mechanism.

The manual pages for the **fcntl**(2) system call, the **lockf**(3) library function, and **fcntl**(5) data structures and commands are referred to throughout this section. You should read them before continuing.

Terminology

Before discussing how record locking should be used, let us first define a few terms.

Record

A contiguous set of bytes in a file. The UNIX operating system does not impose any record structure on files. This may be done by the programs that use the files.

Cooperating Processes

Processes that work together in some well defined fashion to accomplish the tasks at hand. Processes that share files must request permission to access the files before using them. File access permissions must be carefully set to restrict non-cooperating processes from accessing those files. The term process will be used interchangeably with cooperating process to refer to a task obeying such protocols.

Read (Share) Locks

These are used to gain limited access to sections of files. When a read lock is in place on a record, other processes may also read lock that record, in whole or in part. No other process, however, may have or obtain a write lock on an overlapping section of the file. If a process holds a read lock it may assume that no other process will be writing or updating that record at the same time. This access method also permits many processes to read the given record. This might be necessary when searching a file, without the contention involved if a write or exclusive lock were to be used.

Write (Exclusive) Locks

These are used to gain complete control over sections of files. When a write lock is in place on a record, no other process may read or write lock that record, in whole or in part. If a process holds a write lock it may assume that no other process will be reading or writing that record at the same time.

Advisory Locking

A form of record locking that does not interact with the I/O subsystem (i.e. **creat**(2), **open**(2), **read**(2), and **write**(2)). The control over records is accomplished by requiring an appropriate record lock request before I/O operations. If appropriate requests are always made by all processes accessing the file, then the accessibility of the file will be controlled by the interaction of these requests. Advisory locking depends on the individual processes to enforce the record locking protocol; it does not require an accessibility check at the time of each I/O request.

Mandatory Locking

A form of record locking that does interact with the I/O subsystem. Access to locked records is enforced by the **creat**(2), **open**(2), **read**(2), and **write**(2) system calls. If a record is locked, then access of that record by any other process is restricted according to the type of lock on the record. The control over records should still be performed explicitly by requesting an appropriate record lock before I/O operations, but an additional check is made by the system before each I/O operation to ensure the record locking protocol is being honored. Mandatory locking offers an extra synchronization check, but at the cost of some additional system overhead.

File Protection

There are access permissions for UNIX system files to control who may read, write, or execute such a file. These access permissions may only be set by the owner of the file or by the superuser. The permissions of the directory in which the file resides can also affect the ultimate disposition of a file. Note that if the directory permissions allow anyone to write in it, then files within the directory may be removed, even if those files do not have read, write or execute permission for that user. Any information that is worth protecting, is worth protecting properly. If your application warrants the use of record locking, make sure that the permissions on your files and directories are set properly. A record lock, even a mandatory record lock, will only protect the portions of the files that are locked. Other parts of these files might be corrupted if proper precautions are not taken.

Only a known set of programs and/or administrators should be able to read or write a data base. This can be done easily by setting the set-group-ID bit (see **chmod**(1)) of the data base accessing programs. The files can then be accessed by a known set of programs that obey the record locking protocol. An example of such file protection, although record locking is not used, is the **mail**(1) command. In that command only the particular user and the **mail** command can read and write in the unread mail files.

Opening a File for Record Locking

The first requirement for locking a file or segment of a file is having a valid open file descriptor. If read locks are to be done, then the file must be opened with at least read accessibility and likewise for write locks and write accessibility. For our example we will open our file for both read and write access:

```
#include <stdio.h>
#include <errno.h>
#include <fcntl.h>


int fd;     /* file descriptor */
char *filename;

main(argc, argv)
int argc;
char *argv[];
{
   extern void exit(), perror();

   /* get data base file name from command line and open the
    * file for read and write access.
    */
   if (argc < 2) {
      (void) fprintf(stderr, "usage: %s filename\n", argv[0]);
      exit(2);
   }
   filename = argv[1];
   fd = open(filename, O_RDWR);
   if (fd < 0) {
      perror(filename);
```

```
        exit(2);
    }
        .
        .
        .
```

The file is now open for us to perform both locking and I/O functions. We then proceed with the task of setting a lock.

Setting a File Lock

There are several ways for us to set a lock on a file. In part, these methods depend upon how the lock interacts with the rest of the program. There are also questions of performance as well as portability. Two methods will be given here, one using the **fcntl**(2) system call, the other using the */usr/group* standards compatible **lockf**(3) library function call.

Locking an entire file is just a special case of record locking. For both these methods the concept and the effect of the lock are the same. The file is locked starting at a byte offset of zero (0) until the end of the maximum file size. This point extends beyond any real end of the file so that no lock can be placed on this file beyond this point. To do this the value of the size of the lock is set to zero. The code using the **fcntl**(2) system call is as follows:

```
#include <fcntl.h>
#define MAX_TRY   10
int try;
struct flock lck;

try = 0;

/* set up the record locking structure, the address of which
 * is passed to the fcntl system call.
 */
lck.l_type = F_WRLCK;   /* setting a write lock */
lck.l_whence = 0; /* offset l_start from beginning of file */
lck.l_start = 0L;
lck.l_len = 0L;   /* until the end of the file address space */

/* Attempt locking MAX_TRY times before giving up.
 */
while (fcntl(fd, F_SETLK, &lck) < 0) {
   if (errno == EAGAIN || errno == EACCES) {
      /* there might be other errors cases in which
       * you might try again.
       */
      if (++try < MAX_TRY) {
         (void) sleep(2);
         continue;
      }
      (void) fprintf(stderr,"File busy try again later!\n");
      return;
   }
   perror("fcntl");
```

```
    exit(2);
}
    .
    .
    .
```

This portion of code tries to lock a file. This is attempted several times until one of the following things happens:

- the file is locked
- an error occurs
- it gives up trying because MAX_TRY has been exceeded

To perform the same task using the **lockf**(3) function, the code is as follows:

```
#include <unistd.h>
#define MAX_TRY   10
int try;
try = 0;

/* make sure the file pointer
 * is at the beginning of the file.
 */
lseek(fd, 0L, 0);

/* Attempt locking MAX_TRY times before giving up.
 */
while (lockf(fd, F_TLOCK, 0L) < 0) {
    if (errno == EAGAIN || errno == EACCES) {
        /* there might be other errors cases in which
         * you might try again.
         */
        if (++try < MAX_TRY) {
            sleep(2);
            continue;
        }
        (void) fprintf(stderr,"File busy try again later!\n");
        return;
    }
    perror("lockf");
    exit(2);
}
    .
    .
    .
```

It should be noted that the **lockf**(3) example appears to be simpler, but the **fcntl**(2) example exhibits additional flexibility. Using the **fcntl**(2) method, it is possible to set the type and start of the lock request simply by setting a few structure variables. **lockf**(3) merely sets write (exclusive) locks; an additional system call (**lseek**(2)) is required to specify the start of the lock.

Setting and Removing Record Locks

Locking a record is done the same way as locking a file except for the differing starting point and length of the lock. We will now try to solve an interesting and real problem. There are two records (these records may be in the same or different file) that must be updated simultaneously so that other processes get a consistent view of this information. (This type of problem comes up, for example, when updating the interrecord pointers in a doubly linked list.) To do this you must decide the following questions:

- What do you want to lock?
- For multiple locks, what order do you want to lock and unlock the records?
- What do you do if you succeed in getting all the required locks?
- What do you do if you fail to get all the locks?

In managing record locks, you must plan a failure strategy if one cannot obtain all the required locks. It is because of contention for these records that we have decided to use record locking in the first place. Different programs might:

- wait a certain amount of time, and try again
- abort the procedure and warn the user
- let the process sleep until signaled that the lock has been freed
- some combination of the above

Let us now look at our example of inserting an entry into a doubly linked list. For the example, we will assume that the record after which the new record is to be inserted has a read lock on it already. The lock on this record must be changed or promoted to a write lock so that the record may be edited.

Promoting a lock (generally from read lock to write lock) is permitted if no other process is holding a read lock in the same section of the file. If there are processes with pending write locks that are sleeping on the same section of the file, the lock promotion succeeds and the other (sleeping) locks wait. Promoting (or demoting) a write lock to a read lock carries no restrictions. In either case, the lock is merely reset with the new lock type. Because the */usr/group* **lockf** function does not have read locks, lock promotion is not applicable to that call. An example of record locking with lock promotion follows:

```
struct record {
    .
    .      /* data portion of record */
    .
    long prev;  /* index to previous record in the list */
    long next;  /* index to next record in the list */
};

/* Lock promotion using fcntl(2)
 * When this routine is entered it is assumed that there
 * are read locks on "here" and "next".
 * If write locks on "here" and "next" are obtained:
 *    Set a write lock on "this".
```

```
 *    Return index to "this" record.
 * If any write lock is not obtained:
 *    Restore read locks on "here" and "next".
 *    Remove all other locks.
 *    Return a -1.
 */
long
set3lock (this, here, next)
long this, here, next;
{
   struct flock lck;

   lck.l_type = F_WRLCK;   /* setting a write lock */
   lck.l_whence = 0;
      /* offset l_start from beginning of file */
   lck.l_start = here;
   lck.l_len = sizeof(struct record);

   /* promote lock on "here" to write lock */
   if (fcntl(fd, F_SETLKW, &lck) < 0) {
      return (-1);
   }
   /* lock "this" with write lock */
   lck.l_start = this;
   if (fcntl(fd, F_SETLKW, &lck) < 0) {
      /* Lock on "this" failed;
       * demote lock on "here" to read lock.
       */
      lck.l_type = F_RDLCK;
      lck.l_start = here;
      (void) fcntl(fd, F_SETLKW, &lck);
      return (-1);
   }
   /* promote lock on "next" to write lock */
   lck.l_start = next;
   if (fcntl(fd, F_SETLKW, &lck) < 0) {
      /* Lock on "next" failed;
       * demote lock on "here" to read lock,
       */
      lck.l_type = F_RDLCK;
      lck.l_start = here;
      (void) fcntl(fd, F_SETLK, &lck);
      /* and remove lock on "this".
       */
      lck.l_type = F_UNLCK;
      lck.l_start = this;
      (void) fcntl(fd, F_SETLK, &lck);
      return (-1);   /* cannot set lock, try again or quit */
   }

   return (this);
}
```

The locks on these three records were all set to wait (sleep) if another process was blocking them from being set. This was done with the F_SETLKW command. If the F_SETLK command was used instead, the **fcntl** system calls would fail if blocked. The program would then have to be changed to handle the blocked condition in each of the error return sections.

Let us now look at a similar example using the **lockf** function. Since there are no read locks, all (write) locks will be referenced generically as locks.

```
/* Lock promotion using lockf(3)
 * When this routine is entered it is assumed that there
 * are no locks on "here" and "next".
 * If locks are obtained:
 *    Set a lock on "this".
 *    Return index to "this" record.
 * If any lock is not obtained:
 *    Remove all other locks.
 *    Return a -1.
 */

#include <unistd.h>

long
set3lock (this, here, next)
long this, here, next;

{

   /* lock "here" */
   (void) lseek(fd, here, 0);
   if (lockf(fd, F_LOCK, sizeof(struct record)) < 0) {
      return (-1);
   }
   /* lock "this" */
   (void) lseek(fd, this, 0);
   if (lockf(fd, F_LOCK, sizeof(struct record)) < 0) {
      /* Lock on "this" failed.
       * Clear lock on "here".
       */
      (void) lseek(fd, here, 0);
      (void) lockf(fd, F_ULOCK, sizeof(struct record));
      return (-1);

   }

   /* lock "next" */
   (void) lseek(fd, next, 0);
   if (lockf(fd, F_LOCK, sizeof(struct record)) < 0) {

      /* Lock on "next" failed.
       * Clear lock on "here",
       */
      (void) lseek(fd, here, 0);
      (void) lockf(fd, F_ULOCK, sizeof(struct record));
```

```
        /* and remove lock on "this".
         */
        (void) lseek(fd, this, 0);
        (void) lockf(fd, F_ULOCK, sizeof(struct record));
        return (-1);    /* cannot set lock, try again or quit */

    }

    return (this);
}
```

Locks are removed in the same manner as they are set, only the lock type is different (F_UNLCK or F_ULOCK). An unlock cannot be blocked by another process and will only affect locks that were placed by this process. The unlock only affects the section of the file defined in the previous example by **lck**. It is possible to unlock or change the type of lock on a subsection of a previously set lock. This may cause an additional lock (two locks for one system call) to be used by the operating system. This occurs if the subsection is from the middle of the previously set lock.

Getting Lock Information

One can determine which processes, if any, are blocking a lock from being set. This can be used as a simple test or as a means to find locks on a file. A lock is set up as in the previous examples and the F_GETLK command is used in the **fcntl** call. If the lock passed to **fcntl** would be blocked, the first blocking lock is returned to the process through the structure passed to **fcntl**. That is, the lock data passed to **fcntl** is overwritten by blocking lock information. This information includes two pieces of data that have not been discussed yet, **l_pid** and **l_sysid**, that are only used by F_GETLK. (For systems that do not support a distributed architecture the value in **l_sysid** should be ignored.) These fields uniquely identify the process holding the lock.

If a lock passed to **fcntl** using the F_GETLK command would not be blocked by another process' lock, then the **l_type** field is changed to F_UNLCK and the remaining fields in the structure are unaffected. Let us use this capability to print all the segments locked by other processes. Note that if there are several read locks over the same segment only one of these will be found.

```
struct flock lck;

/* Find and print "write lock" blocked segments of this file. */
   (void) printf("sysid   pid type    start   length\n");
   lck.l_whence = 0;
   lck.l_start = 0L;
   lck.l_len = 0L;
   do {
      lck.l_type = F_WRLCK;
      (void) fcntl(fd, F_GETLK, &lck);
      if (lck.l_type != F_UNLCK) {
         (void) printf("%5d %5d   %c  %8d %8d\n",
            lck.l_sysid,
            lck.l_pid,
            (lck.l_type == F_WRLCK) ? 'W' : 'R',
            lck.l_start,
            lck.l_len);
         /* if this lock goes to the end of the address
          * space, no need to look further, so break out.
          */
         if (lck.l_len == 0)
            break;
         /* otherwise, look for new lock after the one
          * just found.
          */
         lck.l_start += lck.l_len;
      }
   } while (lck.l_type != F_UNLCK);
```

fcntl with the F_GETLK command will always return correctly (that is, it will not sleep or fail) if the values passed to it as arguments are valid.

The **lockf** function with the F_TEST command can also be used to test if there is a process blocking a lock. This function does not, however, return the information about where the lock actually is and which process owns the lock. A routine using **lockf** to test for a lock on a file follows:

```
/* find a blocked record. */

/* seek to beginning of file */
(void) lseek(fd, 0, 0L);
/* set the size of the test region to zero (0)
 * to test until the end of the file address space.
 */
if (lockf(fd, F_TEST, 0L) < 0) {
   switch (errno) {
   case EACCES:
   case EAGAIN:
      (void) printf("file is locked by another process\n");
      break;
   case EBADF:
      /* bad argument passed to lockf */
      perror("lockf");
      break;
   default:
      (void) printf("lockf: unknown error <%d>\n", errno);
      break;
   }
}
```

When a process forks, the child receives a copy of the file descriptors that the parent has opened. The parent and child also share a common file pointer for each file. If the parent were to seek to a point in the file, the child's file pointer would also be at that location. This feature has important implications when using record locking. The current value of the file pointer is used as the reference for the offset of the beginning of the lock, as described by **l_start**, when using a **l_whence** value of 1. If both the parent and child process set locks on the same file, there is a possibility that a lock will be set using a file pointer that was reset by the other process. This problem appears in the **lockf**(3) function call as well and is a result of the */usr/group* requirements for record locking. If forking is used in a record locking program, the child process should close and reopen the file if either locking method is used. This will result in the creation of a new and separate file pointer that can be manipulated without this problem occurring. Another solution is to use the **fcntl** system call with a **l_whence** value of 0 or 2. This makes the locking function atomic, so that even processes sharing file pointers can be locked without difficulty.

Deadlock Handling

There is a certain level of deadlock detection/avoidance built into the record locking facility. This deadlock handling provides the same level of protection granted by the */usr/group* standard **lockf** call. This deadlock detection is only valid for processes that are locking files or records on a single system. Deadlocks can only potentially occur when the system is about to put a record locking system call to sleep. A search is made for constraint loops of processes that would cause the system call to sleep indefinitely. If such a situation is found, the locking system call will fail and set **errno** to the deadlock error number. If a process wishes to avoid the use of the systems deadlock detection it should set its locks using F_GETLK instead of F_GETLKW.

Selecting Advisory or Mandatory Locking

The use of mandatory locking is not recommended for reasons that will be made clear in a subsequent section. Whether or not locks are enforced by the I/O system calls is determined at the time the calls are made and the state of the permissions on the file (see **chmod**(2)). For locks to be under mandatory enforcement, the file must be a regular file with the set-group-ID bit on and the group execute permission off. If either condition fails, all record locks are advisory. Mandatory enforcement can be assured by the following code:

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

int mode;
struct stat buf;
      .
      .
      .
   if (stat(filename, &buf) < 0) [
      perror("program");
      exit (2);
   ]
   /* get currently set mode */
   mode = buf.st_mode;
   /* remove group execute permission from mode */
   mode &= ~(S_IEXEC>>3);
   /* set 'set group id bit' in mode */
   mode |= S_ISGID;
   if (chmod(filename, mode) < 0) [
      perror("program");
      exit(2);
   ]
      .
      .
      .
```

Files that are to be record locked should never have any type of execute permission set on them. This is because the operating system does not obey the record locking protocol when executing a file.

The **chmod**(1) command can also be easily used to set a file to have mandatory locking. This can be done with the command:

chmod +l *filename*

The **ls**(1) command was also changed to show this setting when you ask for the long listing format:

ls -l *filename*

causes the following to be printed:

```
-rw---l---  1 abc  other  1048576 Dec  3 11:44 filename
```

Caveat Emptor—Mandatory Locking

- Mandatory locking only protects those portions of a file that are locked. Other portions of the file that are not locked may be accessed according to normal UNIX system file permissions.
- If multiple reads or writes are necessary for an atomic transaction, the process should explicitly lock all such pieces before any I/O begins. Thus advisory enforcement is sufficient for all programs that perform in this way.
- As stated earlier, arbitrary programs should not have unrestricted access permission to files that are important enough to record lock.
- Advisory locking is more efficient because a record lock check does not have to be performed for every I/O request.

Record Locking and Future Releases of the UNIX System

Provisions have been made for file and record locking in a UNIX system environment. In such an environment the system on which the locking process resides may be remote from the system on which the file and record locks reside. In this way multiple processes on different systems may put locks upon a single file that resides on one of these or yet another system. The record locks for a file reside on the system that maintains the file. It is also important to note that deadlock detection/avoidance is only determined by the record locks being held by and for a single system. Therefore, it is necessary that a process only hold record locks on a single system at any given time for the deadlock mechanism to be effective. If a process needs to maintain locks over several systems, it is suggested that the process avoid the **sleep-when-blocked** features of **fcntl** or **lockf** and that the process maintain its own deadlock detection. If the process uses the **sleep-when-blocked** feature, then a timeout mechanism should be provided by the process so that it does not hang waiting for a lock to be cleared.

Chapter 14: Shared Libraries

Introduction

Shared libraries are most advantageous on small machines. On smaller systems, the memory and disk storage savings often justify the performance loss and increased maintenance complexity. However, on large high performance systems, such as MIPS machines, disk and memory are more plentiful. Consequently, shared libraries are less advantageous, and may even be a disadvantage in terms of performance and maintenance. On these systems, the standard system libraries do not provide much benefit in their shared form. This is why there are no shared libraries currently distributed in the UMIPS software releases.

In some instances, however, shared libraries can be very useful. When disk space and memory savings are important, shared libraries can be of benefit. For example, if constructed properly, a shared library can reduce **a.out** file (an executable object file) disk storage and process (an **a.out** file that is executing) memory space.

The first part of this chapter, "Using a Shared Library," is designed to help you use shared libraries. It describes what a shared library is and how to use one to build **a.out** files. It also offers advice about when and when not to use a shared library and how to determine whether an **a.out** uses a shared library.

The second part in this chapter, "Building a Shared Library," describes how to build a shared library. You do not need to read this part to use shared libraries. It addresses library developers, advanced programmers who are expected to build their own shared libraries. Specifically, this part describes how to use the UMIPS system tool **mkshlib**(1) and how to write C code for shared libraries on the UMIPS system. An example is included. Read this part of the chapter only if you have to build a shared library.

Using a Shared Library

If you are accustomed to using libraries to build your applications programs, shared libraries should blend into your work easily. This part of the chapter explains what shared libraries are and how and when to use them on the UMIPS system.

What is a Shared Library?

A shared library is a file containing object code that several **a.out** files may use simultaneously while executing. A shared library, like a library that is not shared, is an archive file. For simplicity, however, we refer to an archive file with shared library members as a shared library and one without as an archive library.

When a program is compiled or link edited with a shared library, the library code that defines the program's external references is not copied into the program's object file. Instead, a special section called **.lib** that identifies the library code is created in the object file. When the UMIPS system executes the resulting **a.out** file, it uses the information in this section to bring the required shared library code into the address space of the process.

A shared library offers several benefits by not copying code into **a.out** files. It can:

- save disk storage space

  Because shared library code is not copied into all the **a.out** files that use the code, these files are smaller and use less disk space.

- save memory

  By sharing library code at run time, the dynamic memory needs of processes are reduced.

- make executable files using library code easier to maintain

  As mentioned above, shared library code is brought into a process' address space at run time. Updating a shared library effectively updates all executable files that use the library, because the operating system brings the updated version into new processes. If an error in shared library code is fixed, all processes automatically use the corrected code.

  Archive libraries cannot, of course, offer this benefit: changes to archive libraries do not affect executable files, because code from the libraries is copied to the files during link editing, not during execution.

The section "Deciding Whether to Use a Shared Library" in this chapter describes shared libraries in more detail.

System Shared Library Conventions

UMIPS currently does not supply any of the system libraries in shared library form. There are, however, conventions established by AT&T for the names and storage locations of shared libraries.

All shared libraries are made up of two files called the *host library* and the *target library*. The host library is the file that the link editor searches when linking programs to create the **.lib** sections in **a.out** files. The target library is the file that the UNIX system uses when running those files. Naturally, the target library must be present for the **a.out** file to run. For example, consider a system library called **libfoo**; its shared version would adhere to the following conventions:

Shared Library	Host Library Command Line Option	Target Library Path Name
Library foo	**–lfoo_s**	**/shlib/libfoo_s**

Notice the **_s** suffix on the library name; it is used to identify both host and target shared libraries. For example, it distinguishes the relocatable foo library **libfoo.a** from the host shared foo library **libfoo_s.a**. The **-l** option passes the library name to the link editor.

Building an a.out File

You direct the link editor to search a shared library the same way you direct a search of an archive library on the UMIPS system:

cc *file*.**c** **–o** *file* ... **–l***library_file* ...

To direct a search of the shared foo library (as in the example in the previous section), you use the following command line.

cc *file*.**c** **–o** *file* ... **–lfoo_s** ...

Coding an Application

Application source code in C or assembly language is compatible with both archive libraries and shared libraries. As a result, you should not have to change the code in any applications you already have when you use a shared library with them. When coding a new application for use with a shared library, you should just observe your standard coding conventions.

However, do keep the following two points in mind, which apply when using either an archive or a shared library:

- Don't define symbols in your application with the same names as those in a library.

  Although there are exceptions, you should avoid redefining standard library routines, such as **printf**(3S) and **strcmp**(3C). Replacements that are incompatibly defined can cause any library, shared or unshared, to behave incorrectly.

- Don't use undocumented archive routines.

  Use only the functions and data mentioned on the manual pages describing the routines in Section 3 of the *Programmer's Reference Manual*. For example, don't try to outsmart the **ctype** design by manipulating the underlying implementation.

Deciding Whether to Use a Shared Library

You should base your decision to use a shared library on whether it saves space in disk storage and memory for your program and does not adversely effect your program's performance. A well-designed shared library almost always saves space.

To determine what savings are gained from using a shared library, you might build the same application with both an archive and a shared library, assuming both kinds of library are available. Remember, that you may do this because source code is compatible between shared libraries and archive libraries. (See the above section "Coding an Application.") Then compare the two versions of the application for size and performance. For example to compare the sizes,

```
% cc -o unshared bar.c -lfoo
% cc -o shared bar.c -lfoo_s
 % size -B unshared shared
text     data    bss     dec      hex
147456   24576   70240   242272   3b260   unshared
65535    24576   70240   160352   27260   shared
```

If the application calls only a few library members, it is possible that using a shared library could take more disk storage or memory. The following section gives a more detailed discussion about when a shared library does and does not save space.

When making your decision about using shared libraries, also remember that they are not implemented on UMIPS releases prior to Release 1.1. If your program must run on previous releases, you will need to use archive libraries.

More About Saving Space

This section is designed to help you better understand why your programs will usually benefit from using a shared library. It explains:

- how shared libraries save space that archive libraries cannot
- how shared libraries are implemented on the UNIX system
- how shared libraries might increase space usage

How Shared Libraries Save Space

To better understand how a shared library saves space, we need to compare it to an archive library.

A host shared library resembles an archive library in three ways. First, as noted earlier, both are archive files. Second, the object code in the library typically defines commonly used text symbols and data symbols. The symbols defined inside and made available outside the library are called exported symbols. Note that the library may also have imported symbols, symbols that it uses but usually does not define. Third, the link editor searches the library for these symbols when linking a program to resolve its external references. By resolving the references, the link editor produces an executable version of the program, the **a.out** file.

NOTE

Note that the link editor on the UMIPS system is a static linking tool; static linking requires that all symbolic references in a program be resolved before the program may be executed. The link editor uses static linking with both an archive library and a shared library.

Although these similarities exist, a shared library differs significantly from an archive library. The major differences relate to how the libraries are handled to resolve symbolic references, a topic already discussed briefly.

Consider how the UMIPS system handles both types of libraries during link editing. To produce an **a.out** file using an archive library, the link editor copies the library code that defines a program's unresolved external reference from the library into appropriate **.text** and **.data** sections in the program's object file. In contrast, to produce an **a.out** file using a shared library, the link editor does not copy any code from the library into the program's object file. Instead, it creates a special section called **.lib** in the file that identifies the library code needed at run time and resolves the external references to shared library symbols with their correct values. When the UMIPS system executes the resulting **a.out** file, it uses the information in the **.lib** section to bring the required shared library code into the address space of the process.

Figure 14-1 depicts the **a.out** files produced using a regular archive version and a shared version of the example foo library to compile the following program:

```
main()
{
    ...
    foo( "How do you like this manual?\n" );
    ...
    result = bar( "I do.", answer );
    ...
}
```

Notice that the shared version is smaller. Figure 14-2 depicts the process images in memory of these two files when they are executed.

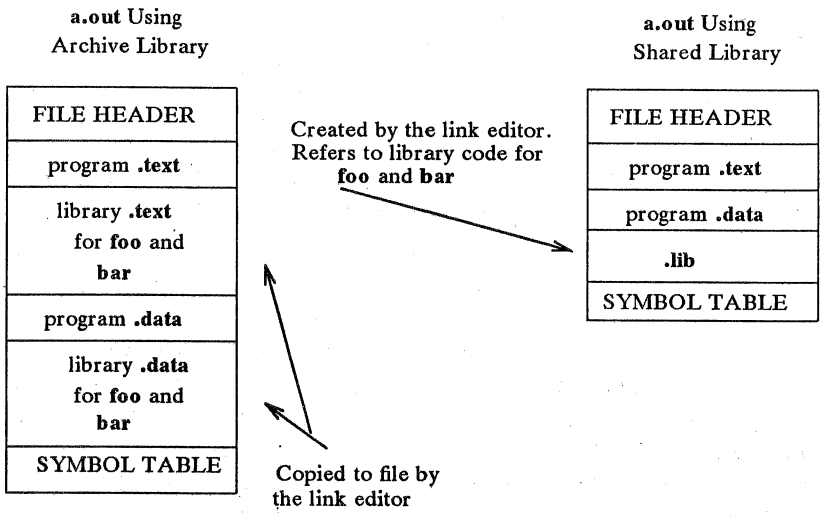


Figure 14-1: **a.out** Files Created Using an Archive Library and a Shared Library

Now consider what happens when several **a.out** files need the same code from a library. When using an archive library, each file gets its own copy of the code. This results in duplication of the same code on the disk and in memory when the **a.out** files are run as processes. In contrast, when a shared library is used, the library code remains separate from the code in the **a.out** files, as indicated in Figure 14-2. This separation enables all processes using the same shared library to reference a single copy of the code.

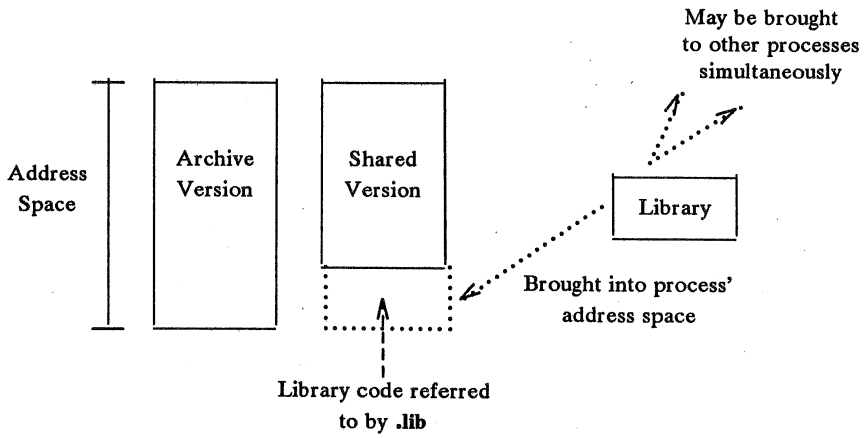


Figure 14-2: Processes Using an Archive and a Shared Library

How Shared Libraries Are Implemented

Now that you have a better understanding of how shared libraries save space, you need to consider their implementation on the UMIPS system to understand how they might increase space usage (this happens seldomly).

The Host Library and Target Library

As previously mentioned, every shared library has two parts: the host library used for linking that resides on the host machine and the target library used for execution that resides on the target machine. The host machine is the machine on which you build an **a.out** file; the target machine is the machine on which you run the file. Of course, the host and target may be the same machine, but they don't have to be.

The host library is just like an archive library. Each of its members (typically a complete object file) defines some text and data symbols in its symbol table. The link editor searches this file when a shared library is used during the compilation or link editing of a program.

The search is for definitions of symbols referenced in the program but not defined there. However, as mentioned earlier, the link editor does not copy the library code defining the symbols into the program's object file. Instead, it uses the library members to locate the definitions and then places symbols in the file that tell where the library code is. The result is the special section in the **a.out** file mentioned earlier (see the section "What is a Shared Library?") and shown in Figure 14-1 as **.lib**.

The target library used for execution resembles an **a.out** file. The UMIPS operating system reads this file during execution if a process needs a shared library. The special **.lib** section in the **a.out** file tells which shared libraries are needed. When the UMIPS system executes the **a.out** file, it uses this section to bring the appropriate library code into the address space of the process. In this way, before the process starts to run, all required library code has been made available.

Shared libraries enable the sharing of **.text** sections in the target library, which is where text symbols are defined. Although processes that use the shared library have their own virtual address spaces, they share a single physical copy of the library's text among them. That is, the UMIPS system uses the same physical code for each process that attaches a shared library's text.

The target library cannot share its **.data** sections. Each process using data from the library has its own private data region (contiguous area of virtual address space that mirrors the **.data** section of the target library). Processes that share text do not share data and stack area so that they do not interfere with one another.

As suggested above, the target library is a lot like an **a.out** file, which can also share its text, but not its data. Also, a process must have execute permission for a target library to execute an **a.out** file that uses the library.

The Branch Table

When the link editor resolves an external reference in a program, it gets the address of the referenced symbol from the host library. This is because a static linking loader like **ld** binds symbols to addresses during link editing. In this way, the **a.out** file for the program has an address for each referenced symbol.

What happens if library code is updated and the address of a symbol changes? Nothing happens to an **a.out** file built with an archive library, because that file already has a copy of the code defining the symbol. (Even though it isn't the updated copy, the **a.out** file will still run.) However, the change can adversely affect an **a.out** file built with a shared library. This file has only a symbol telling where the required library code is. If the library code were updated, the location of that code might

change. Therefore, if the **a.out** file ran after the change took place, the operating system could bring in the wrong code. To keep the **a.out** file current, you might have to recompile a program that uses a shared library after each library update.

To prevent the need to recompile, a shared library is implemented with a branch table on the UMIPS system. A branch table associates text symbols with an absolute address that does not change even when library code is changed. Each address labels a jump instruction to the address of the code that defines a symbol. Instead of being directly associated with the addresses of code, text symbols have addresses in the branch table.

Figure 14-3 shows two **a.out** files executing that make a call to **foo**. The process on the left was built using an archive library. It already has a copy of the library code defining the **foo** symbol. The process on the right was built using a shared library. This file references an absolute address (10) in the branch table of the shared library at run time; at this address, a jump instruction references the needed code.

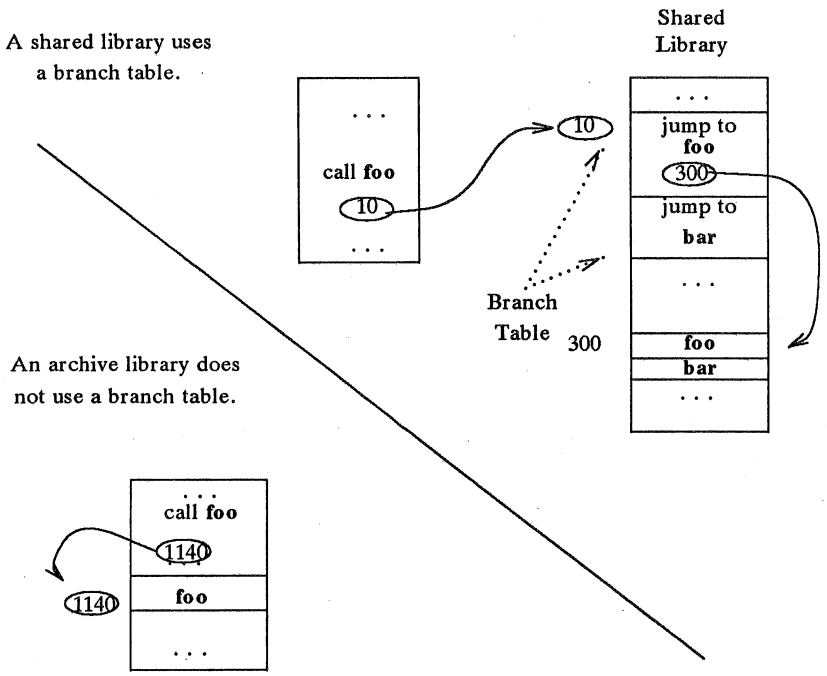


Figure 14-3: A Branch Table in a Shared Library

How Shared Libraries Might Increase Space Usage

A host library might add space to an **a.out** file. Recall that UMIPS uses static linking, which requires that all external references in a program be resolved before it is executed. Also recall that a shared library may have imported symbols, which are used but not defined by the library. These symbols might introduce unresolved references during the linking process. To resolve these references, the link editor has to add the **.text** and **.data** sections defining the referenced imported symbols to the **a.out** file. These sections increase the size of the **a.out** file.

A target library might also add space to a process. Recall that a shared library's target file may have both text and data regions connected to a process (see "How Shared Libraries are Implemented" in this chapter). While the text region is shared by all processes that use the library, the data region is not. Every process that uses the library gets its own private copy of the entire library data region. Naturally, this region adds to the process's memory requirements. As a result, if an application uses only a small part of a shared library's text and data, executing the application might require more memory with a shared library than without one. For example, it would be unwise to use the shared version of library to access only one small routine. Although sharing of the routine saves disk storage and memory, the memory cost for sharing all the shared library's private data region may outweigh the savings. In this case, using the archive version of the library would be more appropriate.

More About Performance Costs

This section is designed to help you better understand the performance costs in using a shared library on a the UMIPS system. These costs are explained in detail regarding the machine instructions generated by the UMIPS compilers. If the shared version of the library doesn't reduce these preformance costs to where the benefits outweigh the performance loss, the shared library shouldn't be used. This section explains:

- the performance cost of shared libraries using global or static data
- the performance cost of shared libraries using imported data
- the performance cost of the branch table

The Cost of Using Global or Static Data

The main performance occurs because the code in shared libraries can't use global pointer references to access its global or static data. Since there is only one global pointer covering a data area accessed with a 16 bit offset, only the user's program or one shared library can use it. For simplicity, the use of the global pointer has been reserved for the user's program. Because shared library references to global data use long references instead of global pointer references, this also doubles the number of reference instructions.

The Cost of Using Imported Data

If a shared library requires global data or routines declared outside the shared library, it must import their use via a pointer. This means that every access to imported data is through a pointer, the value of which must be fetched before the reference can be made. Compounding the problem is the fact that even fetching the pointer must be done with a long reference (two instructions) instead of relative to the global pointer (one instruction).

As with any indirect reference, a delay slot is introduced which waits for the loading of the pointer to complete before it can be used. If the assembler can move something to this delay slot, it costs nothing. Otherwise, it adds a no-op instruction, incurring one more instruction cost to our imported reference.

So, in the worse case, a reference to an imported data item can be four times as expensive. As shown in the following example which sets the global variable errno:

```
lui   $at,0x1000          # load the high 16 bits of
                          #_libfoo_errno
lw    $at(0x0124),$16     # load the value of the pointer
                          # _libfoo_errno
nop   # delay slot
sw    $16(0),$0           # store zero indirect into errno via
                          # _libfoo_errno
```

The Cost of the Branch Table

Since calling any routine in the shared library goes through the branch table, this adds two instructions to the call overhead (a jump instruction and a no-op for the delay slot that can't be filled). For routines that execute very few instructions, this could add significant overhead, especially if these routines are called frequently. Leaf routines are most likely to have this problem since they have a very lean call overhead. A routine of this nature is not a good candidate for inclusion in a shared library, and is best linked with the user's code from an archive library.

When the user's program is linked to the host shared library, the calls to those library routines appear as jumps to absolute locations. It is plausible that the text of the shared library routine could change after linking. Therefore, the link editor can't safely perform the optimization of filling delay slots (the **-jmpopt** option of **ld**(1)). This, of course, results in additional performance costs.

Identifying a.out Files that Use Shared Libraries

Suppose you have an executable file and you want to know whether it uses a shared library. You can use the **dump**(1) command to look at the section headers for the file:

dump -hv a.out

If the file has a **.lib** section, a shared library is required. If the **a.out** does not have a **.lib** section, it does not use a shared library.

With a little more work, you can even tell what libraries a file uses by looking at the **.lib** section contents:

dump -L a.out

Debugging a.out Files that Use Shared Libraries

Debugging support for shared libraries is currently limited. Shared library data are not dumped to core files, and **dbx**(1) does not read the symbol tables of the shared libraries. If you encounter an error that appears not to be in your application code, you may find debugging easier if you rebuild the application with the archive version of the library. The routines in a shared library do show up in stack traces produced by **dbx**(1). The debugger support for these routines is the same as for non-globally stripped object files (produced with the **-x** option of **ld**(1)).

Building a Shared Library

This part of the chapter explains how to build a shared library. It covers the major steps to the building process, the use of the UMIPS system tool **mkshlib**(1) which builds the host and target libraries, and some guidelines for writing shared library code.

This part assumes that you are an advanced C programmer faced with the task of building a shared library. It also assumes you are familiar with the archive library building process. You do not need to read this part of the chapter if you only plan to use existing shared libraries.

The Building Process

To build a shared library on the UMIPS system, you have to complete six major tasks:

- Choose region addresses.
- Choose the path name for the shared library target file.
- Select the library contents.
- Rewrite existing library code to be included in the shared library.
- Write the library specification file.
- Use the **mkshlib** tool to build the host and target libraries.

Here each of these tasks is discussed.

Step 1: Choosing Region Addresses

The first thing you need to do is choose region addresses for your shared library.

Shared library regions on the UMIPS system correspond to the virtual memory region code in the operating-system. Region addresses on the UMIPS system are on 2 MB (0x20000) boundaries, and their sizes are multiples of 2 MB. When choosing the region addresses for your shared library, you must ensure that none of your regions overlap (this includes overlapping the user's program regions).

The link editor will print a warning if any segments overlap, but only if they overlap within a page boundary. However, the operating system will refuse to run a program with segments overlapping within 2 MB boundaries. This is to allow the size of regions to be independent of the link editor.

All the text must be in the same 256 MB segment so jumps can access the entire text. This is because the jump instruction on MIPS machines uses the high 4 bits of the address as the high 4 bits of the jump target address. The link editor will print a jump relocation error if the jump target is not in the same 256 MB segment as the jump instruction.

Since regions are 2 MB in size, this limits the number of shared library text regions to 128 minus the number of 2 MB segments used by the program itself. The recommended starting address for allocating the first text segment is 254 MB. Each subsequent text segment should be allocated on a 2 MB boundary by -2 MB decrements. For example, the first text segment is allocated at 254 MB, the second at 252 MB, the third at 254, and so forth. It is recommended that the user's text segment should be loaded at the default 4 MB address. This allows for maximum growth of the user's program.

NOTE

Any number of libraries can use the same virtual addresses, even on the same machine. Conflicts occur only within a single process, not among separate processes. Thus two shared libraries can have the same region addresses without causing problems, as long as a single **a.out** file doesn't need to use both libraries.

The recommended starting address for allocating the first data segment is 256 MB. Each subsequent data segment should be allocated on a 2 MB boundary by +2 MB decrements. For example, the first data segment segment is allocated at 256 MB, the second at 258 MB, the third at 260 MB, and so forth. This will require the user to use the **-D** link editor flag to move his data past the last shared library data segment. This retains the maximum area for dynamically allocated items, and the maximum area for stack growth.

NOTE

If you plan to build a commercial shared library, you are strongly encouraged to provide a compatible, relocatable archive as well. Some of your customers might not find the shared library appropriate for their applications. Others might want their applications to run on versions of the UMIPS system without shared library support.

Step 2: Choosing the Target Library Path Name

After you select the region addresses for your shared library, you should choose the path name for the target library. The _s suffix is used by convention in the path names of shared libraries. To choose a path name for your shared library, consult the established list of names for your system, or see your system administrator. Also keep in mind that shared libraries required for booting a system should normally be located on the root file system partition. If your shared library is for personal use, you can choose any convenient path name for the target library.

Step 3: Selecting Library Contents

Selecting the contents for your shared library is the most important task in the building process. Some routines are prime candidates for sharing; others are not. For example, it's a good idea to include large, frequently used routines in a shared library but to exclude smaller routines that aren't used as much. What you include will depend on the individual needs of the programmers and other users for whom you are building the library. There are some general guidelines you should follow, however. They are discussed in the section "Choosing Library Members" in this chapter. Also see the guidelines in the following sections "Importing Symbols" and "Tuning the Shared Library Code."

Step 4: Rewriting Existing Library Code

If you choose to include some existing code from an archive library in a shared library, changing some of the code will make the shared code easier to maintain. See the section "Changing Existing Code for the Shared Library" in this chapter.

Step 5: Writing the Library Specification File

After you select and edit all the code for your shared library, you have to build the shared library specification file. The library specification file contains all the information that **mkshlib** needs to build both the host and target libraries. An example specification file is shown in the next section, "An Example." The contents and format of the specification file are given by the following directives (see also the **mkshlib**(1) manual page):

#address *segname address*

Specifies the start address, *address*, of the segment *segname* for the target. This directive is used to specify the start addresses of the text and data segments. Since the headers are part of the text segment of target shared libraries, they are given their own page. The actual text starts on the page after the text segment specification.

#target *pathname*

Specifies the path name, *pathname*, of the target shared library on the target machine. This is the location where the operating system looks for the shared library during execution. Normally, *pathname* will be an absolute path name, but it does not have to be.

This directive can be specified only once per shared library specification file.

#branch

Starts the branch table specifications. The lines following this directive are taken to be branch table specification lines.

Branch table specification lines have the following format:

funcname <white space> *position*

funcname is the name of the symbol given a branch table entry and *position* specifies the position of *funcname*'s branch table entry. *position* may be a single integer or a range of integers of the form *position1-position2*. The following rules apply:

- Each position must be greater than or equal to one.
- The same position cannot be specified more than once
- Position numbering must be sequential. For example, the starting position number following position 1-2 must be 3-x (where x is an optional range); the starting position number following position 3-x is either 4 or x + 1; and so forth. Dummy specification lines must be created for those positions that contain irrelevant data.

A symbol has the address of the highest associated branch table entry when you do either of the following:

- Give the symbol more than one branch table entry by associating a range of positions with the symbol or:
- specify the same symbol on more than one branch table specification line.

All other branch table entries for the symbol are empty slots that can be replaced by new entries in future versions of the shared library.

Finally, only functions should be given branch table entries, and those functions must be external.

This directive can be specified only once per shared library specification file.

#objects Specifies the names of the object files constituting the target shared library. The lines following this directive are taken to be the list of input object files in the order they are to be loaded into the target. The list simply consists of each filename followed by white space. This list of objects will be used to build the shared library.

This directive can be specified only once per shared library specification file.

#init *object* Specifies that the object file, *object*, requires initialization code. The lines following this directive are taken to be initialization specification lines.

Initialization specification lines have the following format:

pimport <white space> *import*

pimport is a pointer to the associated imported symbol, *import*, and must be defined in the current specified object file, *object*. The initialization code generated for each line resembles the C assignment statement:

pimport = &*import*;

The assignments set the pointers to default values. All initializations for a particular object file must be given at once and multiple specifications of the same object file are not allowed.

#ident "*string*" Specifies a string, *string*, to be included in the **.comment** section of the target shared library and the **.comment** sections of every member of the host shared library. Only one **#ident** directive is permitted per shared library specification file. This is ignored on UMIPS systems but allowed for compatibility.

Specifies a comment. The rest of the line is ignored.

All directives that are followed by multi-line specifications are valid until the next directive or the end of file.

Step 6: Using mkshlib to Build the Host and Target

The UMIPS system command **mkshlib**(1) builds both the host and target libraries. **mkshlib** invokes other tools such as the assembler, **as**(1), the archiver, **ar**(1), and link editor, **ld**(1). Tools are invoked through the use of **execvp** (see **exec**(2)) which searches directories in a user's $PATH environment variable. Also, suffixes to **mkshlib** are parsed in much the same manner as suffixes to the **cc**(1) command and invoked tools are given the suffix, where appropriate. For example, **mkshlib1.20** invokes **ld1.20**.

The user input to **mkshlib** consists of the library specification file and command line options. We just discussed the specification file; let's take a look at the options now. The shared library build tool has the following syntax:

mkshlib **−s** *specfil* **−t** *target* [**−h** *host*] [**−n**] [**−q**]

−s *specfil* Specifies the shared library specification file, *specfil*. This file contains all the information necessary to build a shared library, as described in Step 5. Its contents include the branch table specifications for the target, the path name in which the target should be installed, the start addresses of text and data for the target, the initialization specifications for the host, and the list of object files to be included in the shared library.

−t *target* Specifies the name, *target*, of the target shared library produced on the host machine. When *target* is moved to the target machine, it should be installed at the location given in the specification file (see the **#target** directive in the section "Writing the Library Specification File"). If the **−n** option is given, then a new target shared library will not be generated.

−h *host* Specifies the name of the host shared library, *host*. If this option is not given, then the host shared library will not be produced.

−n Prevents a new target shared library from being generated. This option is useful when producing only a new host shared library. The **−t** option must still be supplied since a version of the target shared library is needed to build the host shared library.

−q Suppresses the printing of certain warning messages.

−v Set the verbose option. This option prints the command lines as it executes them (as in the compiler drivers).

An Example

Follow each of the steps in the library building process to build a small example shared library. While building this library, appropriate guidelines will be displayed amidst text. Note that the example code is contrived to show samples of problem areas, not to do anything useful.

The name of our library will be **libexam**. Assume the original code was a single source file, as shown below.

```
/* Original exam.c */

#include <stdio.h>

extern int      strlen();
extern char     *malloc(), *strcpy();

int     count   = 0;
char    *Error;

char *
excopy(e)
        char    *e;
{
        char    *new;

        ++count;
        if ( (new = malloc(strlen(e)+1)) == 0 )
        {
                        Error = "no memory";
                        return 0;
        }
        return strcpy(new, e);
}

excount()
{
        fprintf(stderr, "excount %d\n", count);
        return count;
}
```

To begin, let's choose the region address spaces for the library's **text** and **data** segments. Note that the region addresses must be on a region boundary (2 MB):

```
.text    0x0ffe0000
.data    0x10000000
```

Also choose the path name for our target library:

```
/my/directory/libexam_s
```

Now you need to identify the imported symbols in the library code. (See the guidelines in the section about "Importing Symbols": **malloc, strcpy, strlen, fprintf,** and **_iob**.) A header file defines C preprocessor macros for these symbols; note that you don't use **_iob** directly except through the macro **stderr** from **<stdio.h>**. Also notice the **_libexam_** prefixes for the symbols. The pointers for imported symbols are exported and, therefore, might conflict with other symbols. Using the library name as a prefix reduces the chance of a conflict occurring.

```
/* New file import.h */

#define malloc    (*_libexam_malloc)
#define strcpy    (*_libexam_strcpy)
#define strlen    (*_libexam_strlen)
#define fprintf   (*_libexam_fprintf)
#define _iob      (*_libexam__iob)

extern char       *malloc();
extern char       *strcpy();
extern int        strlen();
extern int        fprintf();
```

The file **import.h** does not declare **_iob** as **extern**; it relies on the header file **<stdio.h>** for this information.

You will also need a new source file to hold definitions of the imported symbol pointers. Remember that all global data need to be initialized:

```
/* New file import.c */

#include <stdio.h>

char   *(*_libexam_malloc)()   = 0;
char   *(*_libexam_strcpy)()   = 0;
int    (*_libexam_strlen)()    = 0;
int    (*_libexam_fprintf)()   = 0;
FILE   (*_libexam__iob)[]      = 0;
```

Next, look at the library's global data to see what needs to be visible externally. (See the guideline "Minimize Global Data.") The variable **count** does not need to be external, because it is accessed through **excount()**. Make it static. (This should have been done for the relocatable version.)

Now the library's global data need to be moved into separate source files. (See the guideline "Define Text and Global Data in Separate Source Files.") The only global datum left is **Error**, and it needs to be initialized. (See the guideline "Initialize Global Data.") **Error** must remain global, because it passes information back to the calling routine:

```
/* New file global.c */

char   *Error   = 0;
```

Integrating these changes into the original source file, we get the following (notice that the symbol names must be declared as **externs**):

```
/* Modified exam.c */

#include "import.h"

#include <stdio.h>

extern int      strlen();
extern char     *malloc(), *strcpy();

static int      count = 0;
extern char     *Error;

char *
excopy(e)
        char    *e;
{
        char    *new;

        ++count;
        if ( (new = malloc(strlen(e)+1)) == 0 )
        {
                Error = "no memory";
                return 0;
        }
        return strcpy(new, e);
}

excount()
{
        fprintf(stderr, "excount %d\n", count);
        return count;
}
```

NOTE

The new header file **import.h** must be included before **<stdio.h>**.

Next, we must write the shared library specification file for **mkshlib**:

```
      /* New file libexam.sl */
1       #target /my/directory/libexam_s
2       #address .text 0x0ffe0000
3       #address .data 0x10000000

4       #branch
5             excopy        1
6             excount       2

7       #objects
8             import.o
9             global.o
10            exam.o

11      #init import.o
12            _libexam_malloc  malloc
13            _libexam_strcpy  strcpy
14            _libexam_strlen  strlen
15            _libexam_fprintf fprintf
16            _libexam__iob    _iob
```

Briefly, here is what the specification file does. Line 1 gives the path name of the shared library on the target machine. The target shared library must be installed there for **a.out** files that use it to work correctly. Lines 2 and 3 give the virtual addresses for the shared library text and data regions, respectively. Line 4 through 6 specify the branch table. Lines 5 and 6 assign the functions **excopy()** and **excount()** to branch table entries 1 and 2. Only external text symbols, such as C functions, should be placed in the branch table.

Lines 7 through 10 give the list of object files that will be used to construct the host and target shared libraries. When building the host shared library archive, each file listed here will reside in its own archive member. When building the target library, the order of object files will be preserved. The data files must be first. Otherwise, a change in the size of static data in **exam.o** would move external data symbols and break compatibility.

Lines 11 through 16 give imported symbol information for the object file **import.o**. You can imagine assignments of the symbol values on the right to the symbols on the left. Thus **_libexam_malloc** will hold a pointer to **malloc**, and so on.

Now, we have to compile the **.o** files as we would for all shared libraries (Note the use of **−G 0**):

```
cc −G 0 −c import.c global.c exam.c
```

Finally, we need to invoke **mkshlib** to build our host and target libraries:

```
mkshlib −s libexam.sl −t libexam_s −h libexam_s.a
```

Presuming all of the source files have been compiled appropriately, the **mkshlib** command line shown above will create both the host library, **libexam_s.a**, and the target library, **libexam_s**.

Guidelines for Writing Shared Library Code

Because the main advantage of a shared library over an archive library is sharing and the space it saves, these guidelines stress ways to increase sharing while avoiding the disadvantages of a shared library. The guidelines also stress upward compatibility.

We recommend that you read these guidelines once from beginning to end to get a perspective of the things you need to consider when building a shared library. Then use it as a checklist to guide your planning and decision-making.

Before we consider these guidelines, let's consider the restrictions to building a shared library common to all the guidelines. These restrictions involve static linking. Here's a summary of them, some of which are discussed in more detail later. Keep them in mind when reading the guidelines in this section:

- Exported symbols have fixed addresses.

  If an exported symbol moves, you have to re-link all **a.out** files that use the library. This restriction applies both to text and data symbols.

- If the library's text changes for one process at run time, it changes for all processes.

  Therefore, any library changes that apply only to a single process must occur in data, not in text, because only the data region is private. (Besides, the text region is read-only.)

- If the library uses a symbol directly, that symbol's run time value (address) must be known when the library is built.

- Imported symbols cannot be referenced directly.

  Their addresses are not known when you build the library, and they can be different for different processes. You can use imported symbols by adding an indirection through a pointer in the library's data.

Choosing Library Members

Include Large, Frequently Used Routines

These routines are prime candidates for sharing. Placing them in a shared library saves code space for individual **a.out** files and saves memory, too, when several concurrent processes need the same code.

Exclude Infrequently Used Routines

Putting these routines in a shared library can degrade performance, particularly on paging systems. Traditional **a.out** files contain all code they need at run time. By definition, the code in an **a.out** file is (at least distantly) related to the process. Therefore, if a process calls a function, it may already be in memory because of its proximity to other text in the process.

If the function is in the shared library, a page fault may be more likely to occur, because the surrounding library code may be unrelated to the calling process. Only rarely will any single **a.out** file use everything in the shared C library. If a shared library has unrelated functions, and unrelated processes make random calls to those functions, the locality of reference may be decreased. The decreased locality may cause more paging activity and, thereby, decrease performance. See also "Organize to Improve Locality."

After profiling your code in your shared library, you should remove small routines that were not often used. This is because functions used only by a few **a.out** files do not save much disk space by being in a shared library, and their inclusion can cause more paging and decrease performance.

Exclude Routines that Use Much Static Data

These modules increase the size of processes. As "How Shared Libraries are Implemented" and "Deciding Whether to Use a Shared Library" explain, every process that uses a shared library gets its own private copy of the library's data, regardless of how much of the data is needed. Library data is static: it is not shared and cannot be loaded selectively with the provision that unreferenced pages may be removed from the working set. This is one of the major performance costs in using a shared library, as explained in "More About Performance Costs".

Exclude Routines Excessively Import and Access Global Data

As explained in the section "The Cost of Using Imported Data" accessing imported data is expensive. The code generated to access such data is 3 to 4 times what it could be in an archive library. So great care must be taken to avoid placing routines in shared libraries that make heavy use of global data.

Exclude Routines that Complicate Maintenance

All exported symbols must remain at constant addresses. The branch table makes this easy for text symbols, but data symbols don't have an equivalent mechanism. The more data a library has, the more likely some of them will have to change size. Any change in the size of exported data may affect symbol addresses and break compatibility.

To take these instructions section literally, you might want to exclude using shared libraries all together. Binaries that use shared libraries are much harder to maintain than those that don't. Factors which normally could be changed from one release to the next, like changing the size of a global exported structure, now can't be altered without breaking compatibility. This cost of using shared libraries must be carefully considered. Using a bit more disk space may be well justified by the savings in maintenance complexities. Remember UMIPS machines tend to be much larger than those for which shared libraries were invented.

Include Routines the Library Itself Must Use

It usually pays to make the library self-contained. If there are a number of related routines and data structures, the shared library should contain all of them.

This guideline should not take priority over the others in this section. If you exclude some routine that the library itself needs based on a previous guideline, consider leaving the symbol out of the library and importing it.

Changing Existing Code for the Shared Library

All C code that works in a shared library will also work in an archive library. However, the reverse is not true because a shared library must explicitly handle imported symbols. The following guidelines are meant to help you produce shared library code that is still valid for archive libraries (although it may be slightly bigger and slower). The guidelines mostly explain how to structure data for ease of maintenance, since most compatibility problems involve restructuring data from a shared library to an archive library.

Minimize Global Data

In the current shared library implementation, all external data symbols are global; they are visible to applications. This can make maintenance difficult. You should try to reduce global data, as described below.

First, try to use automatic (stack) variables. Don't use permanent storage if automatic variables work. Using automatic variables saves static data space and reduces the number of symbols visible to application processes. This also reduces the performance cost to the equivalent cost of an archive library would be. This is because stack references can usually be made with one instruction.

Second, see whether variables really must be external. Static symbols are not visible outside the library, so they may change addresses between library versions. Only external variables must remain constant.

Third, allocate buffers at run time instead of defining them at compile time. This does two important things. It reduces the size of the library's data region for all processes and, therefore, saves memory; only the processes that actually need the buffers get them. It also allows the size of the buffer to change from one release to the next without affecting compatibility. Statically allocated buffers cannot change size without affecting the addresses of other symbols and, perhaps, breaking compatibility. Again, this reduces the performance cost of using a shared library. This is because a pointer to a dynamically allocated object will be cheaper to access than a statically allocated object.

Define Text and Global Data in Separate Source Files

Separating text from global data makes it easier to prevent data symbols from moving. If new exported variables are needed, they can be added at the end of the old definitions to preserve the old symbols' addresses.

Archive libraries let the link editor extract individual members. This sometimes encourages programmers to define related variables and text in the same source file. This works fine for relocatable files, but shared libraries have a different set of restrictions. Suppose exported variables were scattered throughout the library modules. Then visible and hidden data would be intermixed. Changing hidden data, such as a string, like **hello** in the following example, moves subsequent data symbols, even the exported symbols:

Before	Broken Successor
`...`	`...`
`int head = 0;`	`int head = 0;`
`...`	`...`
`func()`	`func()`
`{`	`{`
`...`	`...`
`p = "hello";`	`p = "hello, world";`
`...`	`...`
`}`	`}`
`...`	`...`
`int tail = 0;`	`int tail = 0;`
`...`	`...`

Assume the relative virtual address of **head** is 0 for both examples. The string literals will have the same address too, but they have different lengths. The old and new addresses of **tail** thus will be 12 and 20, respectively. If **tail** is supposed to be visible outside the library, the two versions will not be compatible.

Adding new exported variables to a shared library may change the addresses of static symbols, but this doesn't affect compatibility. An **a.out** file has no way to reference static library symbols directly, so it cannot depend on their values. Thus it pays to group all exported data symbols and place them at lower addresses than the static (hidden) data. You can write the specification file to control this. In the list of object files, make the global data files first.

```
#objects
   data1.o
   ...
   lastdata.o
   text1.o
   text2.o
   ...
```

If the data modules are not first, a seemingly harmless change (such as a new string literal) can break existing **a.out** files.

Shared library users get all library data at run time, regardless of the source file organization. Consequently, you can put all exported variables' definitions in a single source file without a penalty. You can also use several source files for data definitions.

Initialize Global Data

Initialize exported variables, including the pointers for imported symbols. Although this uses more disk space in the target shared library, the expansion is limited to a single file. Using initialized variables is another way to prevent address changes.

The C compilation system on UMIPS puts uninitialized variables in a common area, and the link editor assigns addresses to them in an unpredictable way. In other words, the order of uninitialized symbols may change from one link editor run to the next. However, the link editor will not change the order of initialized variables, thus allowing a library developer to preserve compatibility.

Preserve Branch Table Order

You should add new functions only at the end of the branch table. After you have a specification file for the library, try to maintain compatibility with previous versions. You may add new functions without breaking old **a.out** files as long as previous assignments are not changed. This lets you distribute a new library without having to re-link all of the **a.out** files that used a previous version of the library.

When using ranges of branch table entries, the symbol for that range uses the last entry of the range. So, if other entries are to be added in that range, they must be added before the original range symbol to maintain compatibility.

Importing Symbols

Shared library code cannot directly use symbols defined outside a library, but an escape hatch exists. You can define pointers in the data area and arrange for those pointers to be initialized to the addresses of imported symbols. Library code then accesses imported symbols indirectly, delaying symbol binding until run time. Libraries can import both text and data symbols. Moreover, imported symbols can come from the user's code, another library, or even the library itself. In Figure 14-4,

the symbols **_libfoo_ptr1** and **_libfoo_ptr2** are imported from user's code and the symbol **_libfoo_bar** from the library itself.

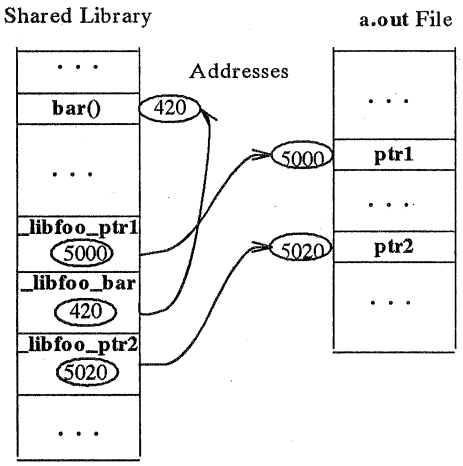


Figure 14-4: Imported Symbols in a Shared Library

The following guidelines describe when and how to use imported symbols.

Imported Symbols that the Library Does Not Define

Archive libraries typically contain relocatable files, which allow undefined references. Although the host shared library is an archive, too, that archive is constructed to mirror the target library, which more closely resembles an **a.out** file. Neither target shared libraries nor **a.out** files can have unresolved symbols.

Consequently, shared libraries must import any symbols they use but do not define. Some shared libraries will derive from existing archive libraries. For the reasons stated above, it may not be appropriate to include all the archive's modules in the target shared library. If you leave something out that the library calls, you have to make an imported symbol pointer for it.

Imported Symbols that Users Must Be Able to Redefine

Optionally, shared libraries can import their own symbols. At first this might appear to be an unnecessary complication, but consider the following. Suppose your shared library can benefit from using a custom set of **malloc** routines. You would rather have your shared library use your custom routines, but do not want to preclude using some other set of **malloc** routines, such as **libmalloc**.

Three possible strategies exist for your shared library. First, you could exclude your custom **malloc**. Library members requiring **malloc** would have to import another version, such as a user's version or the version from some other library. This is feasible, but it means less savings; it also wouldn't be importing your custom **malloc** which works best with your library.

Second, you can include your **malloc** family and not import it. This produces more savings, but at a price. Your other library routines then call your **malloc** directly, and those calls can not be overridden. If a user tries to redefine **malloc**, the library calls won't use the alternate version. Furthermore, the link editor will find multiple definitions of **malloc** while building the user program. To resolve this, you or the user must change the source code to remove the custom **malloc**, or the user must refrain from using the shared library.

Finally, you can include your **malloc** in the shared library, treating it as an imported symbol. Even though **malloc** is in the library, nothing else there refers to it directly. If the user does not redefine **malloc**, both the user's and your library calls are routed to your custom version in the shared library. All calls are mapped to the alternate, if the user defines one.

You might want to permit redefinition of all library symbols in some libraries. You can do this by importing all symbols the library defines, in addition to those it uses but does not define. Although this adds a little space and adds a performance cost to the library, the technique allows a shared library to be one hundred percent compatible with an existing archive at link time and run time.

Mechanics of Importing Symbols

Let's assume a shared library wants to import the symbol **malloc**. The original archive code and the shared library code appear below.

Archive Code	Shared Library Code
	`/* See pointers.c on next page */`
`extern char *malloc();`	`extern char *(*_libfoo_malloc)();`
`export()`	`export()`
`{`	`{`
`...`	`...`
`p = malloc(n);`	`p = (*_libfoo_malloc)(n);`
`...`	`...`
`}`	`}`

Making this transformation is straightforward, but two sets of source code would be necessary to support both an archive and a shared library. Some simple macro definitions can hide the transformations and allow source code compatibility. A header file defines the macros, and a different version of this header file would exist for each type of library. The **-I** flag to **cpp**(1) would direct the C preprocessor to look in the appropriate directory to find the desired file.

Archive **import.h**

```
/* empty */
```

Shared **import.h**

```
/*
 *  Macros for importing
 *  symbols.  One #define
 *  per symbol.
 */

...
#define malloc   (*_libfoo_malloc)
...
extern char *malloc();
...
```

These header files allow one source both to serve the original archive source and to serve a shared library, too, because they supply the indirections for imported symbols. The declaration of **malloc** in **import.h** actually declares the pointer **_libfoo_malloc**.

Common Source

```
#include "import.h"

extern char *malloc();

export()
{
    ...
    p = malloc(n);
    ...
}
```

Alternatively, one can hide the `#include` with `#ifdef`:

Common Source

```
#ifdef SHLIB
#     include "import.h"
#endif

extern char *malloc();

export()
{
    ...
    p = malloc(n);
    ...
}
```

Of course the transformation is not complete. You must define the pointer **_libfoo_malloc**.

File **pointers.c**

```
char *(*_libfoo_malloc)() = 0;
...
```

Note that **_libfoo_malloc** is initialized to zero, because it is an exported data symbol.

Special initialization code sets the pointers. Shared library code should not use the pointer before it contains the correct value. In the example the address of **malloc** must be assigned to **_libfoo_malloc**. Tools that build the shared library generate the initialization code according to the library specification file.

Pointer Initialization Fragments

A host shared library archive member can define one or many imported symbol pointers. Regardless of the number, every imported symbol pointer should have initialization code.

This code goes into the **a.out** file and does two things. First, it creates an unresolved reference to make sure the symbol being imported gets resolved. Second, initialization fragments set the imported symbol pointers to their values before the process reaches **main**. If the imported symbol pointer can be used at run time, the imported symbol will be present, and the imported symbol pointer will be set properly.

NOTE

Initialization fragments reside in the host, not the target, shared library. The link editor copies initialization code into **a.out** files to set imported pointers to their correct values.

Library specification files describe how to initialize the imported symbol pointers. For example, the following specification line would set **_libfoo_malloc** to the address of **malloc**:

```
...
#init pmalloc.o
_libfoo_malloc    malloc
...
```

When **mkshlib** builds the host library, it modifies the file **pmalloc.o**, adding relocatable code to perform the following assignment statement:

_libfoo_malloc = &malloc;

When the link editor extracts **pmalloc.o** from the host library, the relocatable code goes into the **a.out** file. As the link editor builds the final **a.out** file, it resolves the unresolved references and collects all initialization fragments. When the **a.out** file is executed, the run time startup (**crt1.o**) executes the initialization fragments to set the library pointers.

Selectively Loading Imported Symbols

Defining fewer pointers in each archive member increases the granularity of symbol selection and can prevent unnecessary objects from being linked into the **a.out** file. For example, if an archive member defines three pointers to imported symbols, the link editor will resolve all three, even though only one might be needed.

You can reduce unnecessary loading by writing C source files that define imported symbol pointers singly or in related groups. If an imported symbol must be individually selectable, put its pointer in its own source file (and archive member). This will give the link editor a finer granularity to use when it resolves the symbols.

Let's look at some examples. In the coarse method, a single source file might define all pointers to imported symbols:

Old **pointers.c**

```
int (*_libfoo_ptr1)() = 0;
char *(*_libfoo_malloc)() = 0;
int (*_libfoo_ptr2)() = 0;
...
```

Being able to use them individually requires multiple source files and archive members. Each of the new files defines a single pointer or a small group of related pointers:

File	Contents
ptr1.c	`int (*_libfoo_ptr1)() = 0;`
pmalloc.c	`char *(*_libfoo_malloc)() = 0;`
ptr2.c	`int (*_libfoo_ptr2)() = 0;`

Originally, a single object file, **pointers.o**, defines all pointers. Extracting it requires definitions for **ptr1, malloc,** and **ptr2**. The modified example lets one extract each pointer individually, thus avoiding the unresolved reference for unnecessary symbols.

Providing Archive Library Compatibility

Having compatible libraries makes it easy to substitute one for the other. In almost all cases, this can be done without makefile or source file changes.

The host shared library archive file be compatible with the relocatable archive library. However, you may not want the shared library target file to include all routines from the archive: including them all may hurt performance.

The goal of producing compatible archive and shared libraries is relatively easy to accomplish. You do so by building the host library in two steps. First, you should use the available shared library tools to create the host library to match the target exactly. However, the resulting host archive file will not be compatible with the archive library at this point. Second, you should add to the host library the set of relocatable objects residing in the archive library that were missing from the host library. Although this set is not in the shared library target, its inclusion in the host library makes the relocatable and shared libraries compatible.

Tuning the Shared Library Code

Some suggestions for how to organize shared library code to improve performance are presented here. They apply to paging systems, such as UMIPS systems. The suggestions come from the experience of building the shared C library.

A shared library should offer greater benefits for more homogeneous collections of code. For example, a data base library probably could be organized to reduce system paging substantially, if its static and dynamic calling dependencies are predictable.

Profile the Code

To begin, profile the code that might go into the shared library.

Choose Library Contents

Based on profiling information, make some decisions about what to include in the shared library. **a.out** file size is a static property, and paging is a dynamic property. These static and dynamic characteristics may conflict, so you have to decide whether the performance lost is worth the disk space gained. See "Choosing Library Members" in this chapter for more information.

Organize to Improve Locality

When a function is in **a.out** files, it probably resides in a page with other code that is used more often (see "Exclude Infrequently Used Routines"). Try to improve locality of reference by grouping dynamically related functions. If every call of **funcA** generates calls to **funcB** and **funcC**, try to put them in the same page. **cflow**(1) generates this static dependency information. Combine it with profiling to see what things actually are called, as opposed to what things might be called.

Align for Paging

The key is to arrange the shared library target's object files so that frequently used functions do not unnecessarily cross page boundaries. When arranging object files within the target library, be sure to keep the text and data files separate. You can reorder text object files without breaking compatibility; the same is not true for object files that define global data. Once again, an example might best explain this guideline:

The architecture of the MIPS R2000 processor uses 4 KB pages. Using name lists and disassemblies of the shared library target file, you can determine where the page boundaries fall.

After grouping related functions, you should break them into page-sized chunks. Although some object files and functions are larger than a single page, most of them are smaller. Use the infrequently called functions as "glue" between the chunks. Because these functions are referenced less frequently than the page contents, the probability of a page fault decreases.

After determining the branch table, you can rearrange the library's object files without breaking compatibility. Group frequently used, related functions together. Grouping frequently used, unrelated functions together is also useful, because they should be called randomly enough to keep the pages in memory. The following example shows how to change the order of the library's object files:

Before

```
#objects
   ...
   printf.o
   fopen.o
   malloc.o
   strcmp.o
   ...
```

After

```
#objects
   ...
   strcmp.o
   malloc.o
   printf.o
   fopen.o
   ...
```

Making A Shared Library Upward Compatible

The following guidelines explain how to build upward-compatible shared libraries. Note, however, that upward compatibility may not always be an issue. Consider the case in which a shared library is one piece of a larger system and is not delivered as a separate product. In this restricted case, you can identify all **a.out** files that use a particular library. As long as you rebuild all the **a.out** files every time the library changes, versions of the library may be incompatible with each other. This may complicate development, but it is possible.

Comparing Previous Versions of the Library

Shared library developers normally want newer versions of a library to be compatible with previous ones. As mentioned before, **a.out** files will not execute properly otherwise.

The following procedures let you check libraries for compatibility. In these tests, two libraries are said to be compatible if their exported symbols have the same addresses. Although this criterion usually works, it is not foolproof. For example, if a library developer changes the number of arguments a function requires, the new function may not be compatible with the old. This kind of change may not alter symbol addresses, but it will break old **a.out** files.

Let's assume we want to compare two target shared libraries: **new.libx_s** and **old.libx_s**. We use the **nm**(1) command to look at their symbols and **sed**(1) to delete everything except external symbols. A small **sed** program simplifies the job.

New file **cmplib.sed**

```
/^.* [^TDB] .*$/d
/^.*.bt.*$/d
/^.* etext$/d
/^.* edata$/d
/^.* end$/d
/^.* _ftext$/d
/^.* _fdata$/d
/^.* _fbss$/d
/^.* _gp$/d
/^.* _procedure_table$/d
/^.* _procedure_table_size$/d
```

The first line of the **sed** script deletes all lines except those for external symbols. The last lines delete special symbols that have no bearing on library compatibility; they are not visible to application programs. You will have to create your own file to hold the **sed** script.

Now we are ready to create lists of symbol names and values for the new and old libraries:

```
nm -B old.libx_s | sed -f cmplib.sed >old.nm
nm -B new.libx_s | sed -f cmplib.sed >new.nm
```

Next, we compare the symbol values to identify differences:

```
diff old.nm new.nm
```

If all symbols in the two libraries have the same values, the **diff**(1) command will produce no output, and the libraries are compatible. Otherwise, some symbols are different and the two libraries may be incompatible. **diff**(1), **nm**(1), and **sed**(1) are documented in the *User's Reference Manual*.

Dealing with Incompatible Libraries

When you determine that two libraries are incompatible, you have to deal with the incompatibility. You can deal with it in one of two ways. First, you can rebuild all the **a.out** files that use your library. If feasible, this is probably the best choice. Unfortunately, you might not be able to find those **a.out** files, let alone force their owners to rebuild them with your new library.

So your second choice is to give a different target path name to the new version of the library. The host and target path names are independent; so you don't have to change the host library path name. New **a.out** files will use your new target library, but old **a.out** files will continue to access the old library.

As the library developer, it is your responsibility to check for compatibility and, probably, to provide a new target library path name for a new version of a library that is incompatible with older versions. If you fail to resolve compatibility problems, **a.out** files that use your library will not work properly.

You should try to avoid multiple library versions. If too many copies of the same shared library exist, they might actually use more disk space and more memory than the equivalent relocatable version would have.

Summary

This chapter described the UMIPS shared libraries and explained how to use them. It also explained how to build your own shared libraries. Shared libraries are of most advantage on small machines in saving disk storage space and memory. On large high performance systems, such as the MIPS machine, memory savings is not as critical. The performance loss and increased maintenance complexities requires that the choice of using shared libraries be made very carefully on this class of machine.

Chapter 15: Interprocess Communication

Introduction

The UNIX system supports three types of Inter-Process Communication (IPC):

- messages
- semaphores
- shared memory

This chapter describes the system calls for each type of IPC.

Included in the chapter are several example programs that show the use of the IPC system calls.

Since there are many ways in the C Programming Language to accomplish the same task or requirement, keep in mind that the example programs were written for clarity and not for program efficiency. Usually, system calls are embedded within a larger user-written program that makes use of a particular function that the calls provide.

Messages

The message type of IPC allows processes (executing programs) to communicate through the exchange of data stored in buffers. This data is transmitted between processes in discrete portions called messages. Processes using this type of IPC can perform two operations:

- sending
- receiving

Before a message can be sent or received by a process, a process must have the UNIX operating system generate the necessary software mechanisms to handle these operations. A process does this by using the **msgget**(2) system call. While doing this, the process becomes the owner/creator of the message facility and specifies the initial operation permissions for all other processes, including itself. Subsequently, the owner/creator can relinquish ownership or change the operation permissions using the **msgctl**(2) system call. However, the creator remains the creator as long as the facility exists. Other processes with permission can use **msgctl**() to perform various other control functions.

Processes which have permission and are attempting to send or receive a message can suspend execution if they are unsuccessful at performing their operation. That is, a process which is attempting to send a message can wait until the process which is to receive the message is ready and vice versa. A process which specifies that execution is to be suspended is performing a "blocking message operation." A process which does not allow its execution to be suspended is performing a "nonblocking message operation."

A process performing a blocking message operation can be suspended until one of three conditions occurs:

- It is successful.
- It receives a signal.
- The facility is removed.

System calls make these message capabilities available to processes. The calling process passes arguments to a system call, and the system call either successfully or unsuccessfully performs its function. If the system call is successful, it performs its function and returns applicable information. Otherwise, a known error code (−1) is returned to the process, and an external error number variable **errno** is set accordingly.

Before a message can be sent or received, a uniquely identified message queue and data structure must be created. The unique identifier created is called the message queue identifier (**msqid**); it is used to identify or reference the associated message queue and data structure.

The message queue is used to store (header) information about each message that is being sent or received. This information includes the following for each message:

- pointer to the next message on queue
- message type
- message text size

- message text address

There is one associated data structure for the uniquely identified message queue. This data structure contains the following information related to the message queue:

- operation permissions data (operation permission structure)
- pointer to first message on the queue
- pointer to last message on the queue
- current number of bytes on the queue
- number of messages on the queue
- maximum number of bytes on the queue
- process identification (PID) of last message sender
- PID of last message receiver
- last message send time
- last message receive time
- last change time

All include files discussed in this chapter are located in the **/usr/include** or **/usr/include/sys** directories.

The C Programming Language data structure definition for the message information contained in the message queue is as follows:

```
struct msg
{
   struct msg  *msg_next;  /* ptr to next message on q */
   long     msg_type;   /* message type */
   short    msg_ts;     /* message text size */
   short    msg_spot;   /* message text map address */
};
```

It is located in the **/usr/include/sys/msg.h** header file.

Likewise, the structure definition for the associated data structure is as follows:

```
struct msqid_ds
{
   struct ipc_perm  msg_perm;    /* operation permission struct */
   struct msg  *msg_first;  /* ptr to first message on q */
   struct msg  *msg_last;   /* ptr to last message on q */
   ushort       msg_cbytes;  /* current # bytes on q */
   ushort       msg_qnum;    /* # of messages on q */
   ushort       msg_qbytes;  /* max # of bytes on q */
   ushort       msg_lspid;   /* pid of last msgsnd */
   ushort       msg_lrpid;   /* pid of last msgrcv */
   time_t       msg_stime;   /* last msgsnd time */
   time_t       msg_rtime;   /* last msgrcv time */
   time_t       msg_ctime;   /* last change time */
};
```

It is located in the **#include <sys/msg.h>** header file also. Note that the **msg_perm** member of this structure uses **ipc_perm** as a template. The breakout for the operation permissions data structure is shown in Figure 15-1.

The definition of the **ipc_perm** data structure is as follows:

```
struct ipc_perm
{
   ushort   uid;  /* owner's user id */
   ushort   gid;  /* owner's group id */
   ushort   cuid; /* creator's user id */
   ushort   cgid; /* creator's group id */
   ushort   mode; /* access modes */
   ushort   seq;  /* slot usage sequence number */
   key_t key;  /* key */
};
```

Figure 15-1: **ipc_perm** Data Structure

It is located in the **#include <sys/ipc.h>** header file; it is common for all IPC facilities.

The **msgget**(2) system call is used to perform two tasks when only the IPC_CREAT flag is set in the **msgflg** argument that it receives:

- to get a new **msqid** and create an associated message queue and data structure for it
- to return an existing **msqid** that already has an associated message queue and data structure

The task performed is determined by the value of the **key** argument passed to the **msgget**() system call. For the first task, if the **key** is not already in use for an existing **msqid**, a new **msqid** is returned with an associated message queue and data structure created for the **key**. This occurs provided no system tunable parameters would be exceeded.

There is also a provision for specifying a **key** of value zero which is known as the private **key** (IPC_PRIVATE = 0); when specified, a new **msqid** is always returned with an associated message queue and data structure created for it unless a system tunable parameter would be exceeded. When the **ipcs** command is performed, for security reasons the KEY field for the **msqid** is all zeros.

For the second task, if a **msqid** exists for the **key** specified, the value of the existing **msqid** is returned. If you do not desire to have an existing **msqid** returned, a control command (IPC_EXCL) can be specified (set) in the **msgflg** argument passed to the system call. The details of using this system call are discussed in the "Using **msgget**" section of this chapter.

When performing the first task, the process which calls **msgget** becomes the owner/creator, and the associated data structure is initialized accordingly. Remember, ownership can be changed but the creating process always remains the creator; see the "Controlling Message Queues" section in this chapter. The creator of the message queue also determines the initial operation permissions for it.

Once a uniquely identified message queue and data structure are created, message operations [**msgop**()] and message control [**msgctl**()] can be used.

Message operations, as mentioned previously, consist of sending and receiving messages. System calls are provided for each of these operations; they are **msgsnd**() and **msgrcv**(). Refer to the "Operations for Messages" section in this chapter for details of these system calls.

Message control is done by using the **msgctl**(2) system call. It permits you to control the message facility in the following ways:

- to determine the associated data structure status for a message queue identifier (**msqid**)
- to change operation permissions for a message queue
- to change the **size** (**msg_qbytes**) of the message queue for a particular **msqid**
- to remove a particular **msqid** from the UNIX operating system along with its associated message queue and data structure

Refer to the "Controlling Message Queues" section in this chapter for details of the **msgctl**() system call.

Getting Message Queues

This section gives a detailed description of using the **msgget**(2) system call along with an example program illustrating its use.

Using msgget

The synopsis found in the **msgget**(2) entry in the *Programmer's Reference Manual* is as follows:

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

int  msgget (key, msgflg)
key_t  key;
int msgflg;
```

All of these include files are located in the **/usr/include/sys** directory of the UNIX operating system.

The following line in the synopsis:

```
int msgget (key, msgflg)
```

informs you that **msgget**() is a function with two formal arguments that returns an integer type value, upon successful completion (**msqid**). The next two lines:

```
key_t  key;
int msgflg;
```

declare the types of the formal arguments. **key_t** is declared by a **typedef** in the **types.h** header file to be an integer.

The integer returned from this function upon successful completion is the message queue identifier (**msqid**) that was discussed earlier.

As declared, the process calling the **msgget()** system call must supply two arguments to be passed to the formal **key** and **msgflg** arguments.

A new **msqid** with an associated message queue and data structure is provided if either

- **key** is equal to IPC_PRIVATE,

or

- **key** is passed a unique hexadecimal integer, and **msgflg** ANDed with IPC_CREAT is TRUE.

The value passed to the **msgflg** argument must be an integer type octal value and it will specify the following:

- access permissions
- execution modes
- control fields (commands)

Access permissions determine the read/write attributes and execution modes determine the user/group/other attributes of the **msgflg** argument. They are collectively referred to as "operation permissions." Figure 15-2 reflects the numeric values (expressed in octal notation) for the valid operation permissions codes.

Operation Permissions	Octal Value
Read by User	00400
Write by User	00200
Read by Group	00040
Write by Group	00020
Read by Others	00004
Write by Others	00002

Figure 15-2: Operation Permissions Codes

A specific octal value is derived by adding the octal values for the operation permissions desired. That is, if read by user and read/write by others is desired, the code value would be 00406 (00400 plus 00006). There are constants located in the **msg.h** header file which can be used for the user (OWNER).

Control commands are predefined constants (represented by all uppercase letters). Figure 15-3 contains the names of the constants which apply to the **msgget()** system call along with their values. They are also referred to as flags and are defined in the **ipc.h** header file.

Control Command	Value
IPC_CREAT	0001000
IPC_EXCL	0002000

Figure 15-3: Control Commands (Flags)

The value for **msgflg** is therefore a combination of operation permissions and control commands. After determining the value for the operation permissions as previously described, the desired flag(s) can be specified. This is accomplished by bitwise ORing (|) them with the operation permissions; the bit positions and values for the control commands in relation to those of the operation permissions make this possible. It is illustrated as follows:

		Octal Value	Binary Value
IPC_CREAT	=	0 1 0 0 0	0 000 001 000 000 000
| ORed by User	=	0 0 4 0 0	0 000 000 100 000 000
msgflg	=	0 1 4 0 0	0 000 001 100 000 000

The **msgflg** value can be easily set by using the names of the flags in conjunction with the octal operation permissions value:

```
msqid = msgget (key, (IPC_CREAT | 0400));

msqid = msgget (key, (IPC_CREAT | IPC_EXCL | 0400));
```

As specified by the **msgget**(2) page in the *Programmer's Reference Manual*, success or failure of this system call depends upon the argument values for **key** and **msgflg** or system tunable parameters. The system call will attempt to return a new **msqid** if one of the following conditions is true:

- Key is equal to IPC_PRIVATE (0)
- Key does not already have a **msqid** associated with it, and (**msgflg** & IPC_CREAT) is "true" (not zero).

The **key** argument can be set to IPC_PRIVATE in the following ways:

```
msqid = msgget (IPC_PRIVATE, msgflg);
```

or

```
msqid = msgget ( 0 , msgflg);
```

This alone will cause the system call to be attempted because it satisfies the first condition specified. Exceeding the MSGMNI system tunable parameter always causes a failure. The MSGMNI system tunable parameter determines the maximum number of unique message queues (**msqid**'s) in the UNIX operating system.

The second condition is satisfied if the value for **key** is not already associated with a **msqid** and the bitwise ANDing of **msgflg** and IPC_CREAT is "true" (not zero). This means that the **key** is unique (not in use) within the UNIX operating system for this facility type and that the IPC_CREAT flag is set (**msgflg** | IPC_CREAT). The bitwise ANDing (&), which is the logical way of testing if a flag is set, is illustrated as follows:

msgflg	=	x 1 x x x	(x = immaterial)
& IPC_CREAT	=	0 1 0 0 0	
result	=	0 1 0 0 0	(not zero)

Since the result is not zero, the flag is set or "true."

IPC_EXCL is another control command used in conjunction with IPC_CREAT to exclusively have the system call fail if, and only if, a **msqid** exists for the specified **key** provided. This is necessary to prevent the process from thinking that it has received a new (unique) **msqid** when it has not. In other words, when both IPC_CREAT and IPC_EXCL are specified, a new **msqid** is returned if the system call is successful.

Refer to the **msgget**(2) page in the *Programmer's Reference Manual* for specific associated data structure initialization for successful completion. The specific failure conditions with error names are contained there also.

Example Program

The example program in this section (Figure 15-4) is a menu driven program which allows all possible combinations of using the **msgget**(2) system call to be exercised.

From studying this program, you can observe the method of passing arguments and receiving return values. The user-written program requirements are pointed out.

This program begins (lines 4-8) by including the required header files as specified by the **msgget**(2) entry in the *Programmer's Reference Manual*. Note that the **errno.h** header file is included as opposed to declaring **errno** as an external variable; either method will work.

Variable names have been chosen to be as close as possible to those in the synopsis for the system call. Their declarations are self-explanatory. These names make the program more readable, and it is perfectly legal since they are local to the program. The variables declared for this program and their purposes are as follows:

- **key**—used to pass the value for the desired **key**
- **opperm**—used to store the desired operation permissions
- **flags**—used to store the desired control commands (flags)
- **opperm_flags**—used to store the combination from the logical ORing of the **opperm** and **flags** variables; it is then used in the system call to pass the **msgflg** argument
- **msqid**—used for returning the message queue identification number for a successful system call or the error code (−1) for an unsuccessful one.

The program begins by prompting for a hexadecimal **key**, an octal operation permissions code, and finally for the control command combinations (flags) which are selected from a menu (lines 15-32). All possible combinations are allowed even though they might not be viable. This allows observing the errors for illegal combinations.

Next, the menu selection for the flags is combined with the operation permissions, and the result is stored at the address of the **opperm_flags** variable (lines 36-51).

The system call is made next, and the result is stored at the address of the **msqid** variable (line 53).

Since the **msqid** variable now contains a valid message queue identifier or the error code (−1), it is tested to see if an error occurred (line 55). If **msqid** equals −1, a message indicates that an error resulted, and the external **errno** variable is displayed (lines 57, 58).

If no error occurred, the returned message queue identifier is displayed (line 62).

The example program for the **msgget**(2) system call follows. It is suggested that the source program file be named **msgget.c** and that the executable file be named **msgget**. When compiling C programs that use floating point operations, the **-f** option should be used on the **cc** command line. If this option is not used, the program will compile successfully, but when the program is executed it will fail.

```
1    /*This is a program to illustrate
2    **the message get, msgget(),
3    **system call capabilities.*/

4    #include    <stdio.h>
5    #include    <sys/types.h>
6    #include    <sys/ipc.h>
7    #include    <sys/msg.h>
8    #include    <errno.h>

9    /*Start of main C language program*/
10    main()
11    {
12        key_t key;    /*declare as long integer*/
13        int opperm, flags;
14        int msqid, opperm_flags;
15        /*Enter the desired key*/
16        printf("Enter the desired key in hex = ");
17        scanf("%x", &key);


18        /*Enter the desired octal operation
19          permissions.*/
20        printf("\nEnter the operation\n");
21        printf("permissions in octal = ");
22        scanf("%o", &opperm);

23           /*Set the desired flags.*/
24           printf("\nEnter corresponding number to\n");
25           printf("set the desired flags:\n");
26           printf("No flags                = 0\n");
27           printf("IPC_CREAT               = 1\n");
28           printf("IPC_EXCL                = 2\n");
29           printf("IPC_CREAT and IPC_EXCL = 3\n");
30           printf("            Flags       = ");

31           /*Get the flag(s) to be set.*/
32           scanf("%d", &flags);

33           /*Check the values.*/
34           printf ("\nkey =0x%x, opperm = 0%o, flags = 0%o\n",
35              key, opperm, flags);
```

Figure 15-4: **msgget**() System Call Example (Sheet 1 of 2)

```
36  /*Incorporate the control fields (flags) with
37    the operation permissions*/
38  switch (flags)
39  {
40  case 0:     /*No flags are to be set.*/
41    opperm_flags = (opperm | 0);
42    break;
43  case 1:     /*Set the IPC_CREAT flag.*/
44    opperm_flags = (opperm | IPC_CREAT);
45    break;
46  case 2:     /*Set the IPC_EXCL flag.*/
47    opperm_flags = (opperm | IPC_EXCL);
48    break;
49  case 3:     /*Set the IPC_CREAT and IPC_EXCL flags.*/
50    opperm_flags = (opperm | IPC_CREAT | IPC_EXCL);
51  }

52  /*Call the msgget system call.*/
53  msqid = msgget (key, opperm_flags);

54  /*Perform the following if the call is unsuccessful.*/
55  if(msqid == -1)
56  {
57    printf ("\nThe msgget system call failed!\n");
58    printf ("The error number = %d\n", errno);
59  }

60  /*Return the msqid upon successful completion.*/
61  else
62    printf ("\nThe msqid = %d\n", msqid);
63  exit(0);
64    }
```

Figure 15-4: **msgget**() System Call Example (Sheet 2 of 2)

Controlling Message Queues

This section gives a detailed description of using the **msgctl** system call along with an example program which allows all of its capabilities to be exercised.

Using msgctl

The synopsis found in the **msgctl**(2) entry in the *Programmer's Reference Manual* is as follows:

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

int msgctl (msqid, cmd, buf)
int msqid, cmd;
struct msqid_ds *buf;
```

The **msgctl()** system call requires three arguments to be passed to it, and it returns an integer value.

Upon successful completion, a zero value is returned; and when unsuccessful, it returns a −1.

The **msqid** variable must be a valid, non-negative, integer value. In other words, it must have already been created by using the **msgget()** system call.

The **cmd** argument can be replaced by one of the following control commands (flags):

IPC_STAT return the status information contained in the associated data structure for the specified **msqid**, and place it in the data structure pointed to by the ***buf** pointer in the user memory area.

IPC_SET for the specified **msqid**, set the effective user and group identification, operation permissions, and the number of bytes for the message queue.

IPC_RMID remove the specified **msqid** along with its associated message queue and data structure.

A process must have an effective user identification of OWNER/CREATOR or super-user to perform an IPC_SET or IPC_RMID control command. Read permission is required to perform the IPC_STAT control command.

The details of this system call are discussed in the example program for it. If you have problems understanding the logic manipulations in this program, read the "Using **msgget**" section of this chapter; it goes into more detail than what would be practical to do for every system call.

Example Program

The example program in this section (Figure 15-5) is a menu driven program which allows all possible combinations of using the **msgctl**(2) system call to be exercised.

From studying this program, you can observe the method of passing arguments and receiving return values. The user-written program requirements are pointed out.

This program begins (lines 5-9) by including the required header files as specified by the **msgctl**(2) entry in the *Programmer's Reference Manual*. Note in this program that **errno** is declared as an external variable, and therefore, the **errno.h** header file does not have to be included.

Variable and structure names have been chosen to be as close as possible to those in the synopsis for the system call. Their declarations are self-explanatory. These names make the program more readable, and it is perfectly legal since they are local to the program. The variables declared for this program and their purpose are as follows:

uid	used to store the IPC_SET value for the effective user identification
gid	used to store the IPC_SET value for the effective group identification
mode	used to store the IPC_SET value for the operation permissions
bytes	used to store the IPC_SET value for the number of bytes in the message queue (**msg_qbytes**)
rtrn	used to store the return integer value from the system call
msqid	used to store and pass the message queue identifier to the system call
command	used to store the code for the desired control command so that subsequent processing can be performed on it
choice	used to determine which member is to be changed for the IPC_SET control command
msqid_ds	used to receive the specified message queue indentifier's data structure when an IPC_STAT control command is performed
***buf**	a pointer passed to the system call which locates the data structure in the user memory area where the IPC_STAT control command is to place its return values or where the IPC_SET command gets the values to set

Note that the **msqid_ds** data structure in this program (line 16) uses the data structure located in the **msg.h** header file of the same name as a template for its declaration. This is a perfect example of the advantage of local variables.

The next important thing to observe is that although the ***buf** pointer is declared to be a pointer to a data structure of the **msqid_ds** type, it must also be initialized to contain the address of the user memory area data structure (line 17). Now that all of the required declarations have been explained for this program, this is how it works.

First, the program prompts for a valid message queue identifier which is stored at the address of the **msqid** variable (lines 19, 20). This is required for every **msgctl** system call.

Then the code for the desired control command must be entered (lines 21-27), and it is stored at the address of the command variable. The code is tested to determine the control command for subsequent processing.

If the IPC_STAT control command is selected (code 1), the system call is performed (lines 37, 38) and the status information returned is printed out (lines 39-46); only the members that can be set are printed out in this program. Note that if the system call is unsuccessful (line 106), the status information of the last successful call is printed out. In addition, an error message is displayed and the **errno** variable is printed out (lines 108, 109). If the system call is successful, a message indicates this along with the message queue identifier used (lines 111-114).

If the IPC_SET control command is selected (code 2), the first thing done is to get the current status information for the message queue identifier specified (lines 50-52). This is necessary because this example program provides for changing only one member at a time, and the system call changes all of them. Also, if an invalid value happened to be stored in the user memory area for one of these members, it would cause repetitive failures for this control command until corrected. The next thing the program does is to prompt for a code corresponding to the member to be changed (lines 53-59). This code is stored at the address of the choice variable (line 60). Now, depending upon the member picked, the program prompts for the new value

(lines 66-95). The value is placed at the address of the appropriate member in the user memory area data structure, and the system call is made (lines 96-98). Depending upon success or failure, the program returns the same messages as for IPC_STAT above.

If the IPC_RMID control command (code 3) is selected, the system call is performed (lines 100-103), and the **msqid** along with its associated message queue and data structure are removed from the UNIX operating system. Note that the ***buf** pointer is not required as an argument to perform this control command, and its value can be zero or NULL. Depending upon the success or failure, the program returns the same messages as for the other control commands.

The example program for the **msgctl()** system call follows. It is suggested that the source program file be named **msgctl.c** and that the executable file be named **msgctl**. When compiling C programs that use floating point operations, the **-f** option should be used on the **cc** command line. If this option is not used, the program will compile successfully, but when the program is executed it will fail.

```
1    /*This is a program to illustrate
2    **the message control, msgctl(),
3    **system call capabilities.
4    */

5    /*Include necessary header files.*/
6    #include    <stdio.h>
7    #include    <sys/types.h>
8    #include    <sys/ipc.h>
9    #include    <sys/msg.h>

10    /*Start of main C language program*/
11    main()
12    {
13        extern int errno;
14        int uid, gid, mode, bytes;
15        int rtrn, msqid, command, choice;
16        struct msqid_ds msqid_ds, *buf;
17        buf = &msqid_ds;

18        /*Get the msqid, and command.*/
19        printf("Enter the msqid = ");
20        scanf("%d", &msqid);
21        printf("\nEnter the number for\n");
22        printf("the desired command:\n");
23        printf("IPC_STAT    =  1\n");
24        printf("IPC_SET     =  2\n");
25        printf("IPC_RMID    =  3\n");
26        printf("Entry       =  ");
27        scanf("%d", &command);

28 /*Check the values.*/
29 printf ("\nmsqid =%d, command = %d\n",
30    msqid, command);

31 switch (command)
32 {
33 case 1:  /*Use msgctl() to duplicate
34      the data structure for
35      msqid in the msqid_ds area pointed
36      to by buf and then print it out.*/
37    rtrn = msgctl(msqid, IPC_STAT,
38        buf);
39    printf ("\nThe USER ID = %d\n",
40        buf->msg_perm.uid);
41    printf ("The GROUP ID = %d\n",
42        buf->msg_perm.gid);
43    printf ("The operation permissions = 0%o\n",
44        buf->msg_perm.mode);
```

Figure 15-5: **msgctl**() System Call Example (Sheet 1 of 3)

```
45     printf ("The msg_qbytes = %d\n",
46         buf->msg_qbytes);
47     break;

48 case 2:  /*Select and change the desired
49           member(s) of the data structure.*/
50     /*Get the original data for this msqid
51       data structure first.*/
52     rtrn = msgctl(msqid, IPC_STAT, buf);
53     printf("\nEnter the number for the\n");
54     printf("member to be changed:\n");
55     printf("msg_perm.uid    = 1\n");
56     printf("msg_perm.gid    = 2\n");
57     printf("msg_perm.mode   = 3\n");
58     printf("msg_qbytes      = 4\n");
59     printf("Entry           = ");
60             scanf("%d", &choice);
61             /*Only one choice is allowed per
62               pass as an illegal entry will
63                   cause repetitive failures until
64               msqid_ds is updated with
65                   IPC_STAT.*/

66             switch(choice){
67             case 1:
68                 printf("\nEnter USER ID = ");
69                 scanf ("%d", &uid);
70                 buf->msg_perm.uid = uid;
71                 printf("\nUSER ID = %d\n",
72                     buf->msg_perm.uid);
73                 break;
74             case 2:
75                 printf("\nEnter GROUP ID = ");
76                 scanf("%d", &gid);
77                 buf->msg_perm.gid = gid;
78                 printf("\nGROUP ID = %d\n",
79                     buf->msg_perm.gid);
80                 break;
81             case 3:
82                 printf("\nEnter MODE = ");
83                 scanf("%o", &mode);
84                 buf->msg_perm.mode = mode;
85                 printf("\nMODE = 0%o\n",
86                     buf->msg_perm.mode);
87                 break;
```

Figure 15-5: **msgctl**() System Call Example (Sheet 2 of 3)

```
88          case 4:
89             printf("\nEnter msq_bytes = ");
90             scanf("%d", &bytes);
91             buf->msg_qbytes = bytes;
92             printf("\nmsg_qbytes = %d\n",
93                buf->msg_qbytes);
94             break;
95          }

96          /*Do the change.*/
97          rtrn = msgctl(msqid, IPC_SET,
98             buf);
99          break;

100         case 3:    /*Remove the msqid along with its
101                    associated message queue
102                    and data structure.*/
103            rtrn = msgctl(msqid, IPC_RMID, NULL);
104         }
105         /*Perform the following if the call is unsuccessful.*/
106         if(rtrn == -1)
107         {
108            printf ("\nThe msgctl system call failed!\n");
109            printf ("The error number = %d\n", errno);
110         }
111         /*Return the msqid upon successful completion.*/
112         else
113            printf ("\nMsgctl was successful for msqid = %d\n",
114               msqid);
115         exit (0);
116      }
```

Figure 15-5: **msgctl**() System Call Example (Sheet 3 of 3)

Operations for Messages

This section gives a detailed description of using the **msgsnd**(2) and **msgrcv**(2) system calls, along with an example program which allows all of their capabilities to be exercised.

Using msgop

The synopsis found in the **msgop**(2) entry in the *Programmer's Reference Manual* is as follows:

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

int msgsnd (msqid, msgp, msgsz, msgflg)
int msqid;
struct msgbuf *msgp;
int msgsz, msgflg;

int msgrcv (msqid, msgp, msgsz, msgtyp, msgflg)
int msqid;
struct msgbuf *msgp;
int msgsz;
long msgtyp;
int msgflg;
```

Sending a Message

The **msgsnd** system call requires four arguments to be passed to it. It returns an integer value.

Upon successful completion, a zero value is returned; and when unsuccessful, **msgsnd**() returns a −1.

The **msqid** argument must be a valid, non-negative, integer value. In other words, it must have already been created by using the **msgget**() system call.

The **msgp** argument is a pointer to a structure in the user memory area that contains the type of the message and the message to be sent.

The **msgsz** argument specifies the length of the character array in the data structure pointed to by the **msgp** argument. This is the length of the message. The maximum **size** of this array is determined by the MSGMAX system tunable parameter.

The **msg_qbytes** data structure member can be lowered from MSGMNB by using the **msgctl**() IPC_SET control command, but only the super-user can raise it afterwards.

The **msgflg** argument allows the "blocking message operation" to be performed if the IPC_NOWAIT flag is not set (**msgflg** & IPC_NOWAIT = 0); this would occur if the total number of bytes allowed on the specified message queue are in use (**msg_qbytes** or MSGMNB), or the total system-wide number of messages on all queues is equal to the system imposed limit (MSGTQL). If the IPC_NOWAIT flag is set, the system call will fail and return a −1.

Further details of this system call are discussed in the example program for it. If you have problems understanding the logic manipulations in this program, read the "Using **msgget**" section of this chapter; it goes into more detail than what would be practical to do for every system call.

Receiving Messages

The **msgrcv**() system call requires five arguments to be passed to it, and it returns an integer value.

Upon successful completion, a value equal to the number of bytes received is returned and when unsuccessful it returns a −1.

The **msqid** argument must be a valid, non-negative, integer value. In other words, it must have already been created by using the **msgget**() system call.

The **msgp** argument is a pointer to a structure in the user memory area that will receive the message type and the message text.

The **msgsz** argument specifies the length of the message to be received. If its value is less than the message in the array, an error can be returned if desired; see the **msgflg** argument.

The **msgtyp** argument is used to pick the first message on the message queue of the particular type specified. If it is equal to zero, the first message on the queue is received; if it is greater than zero, the first message of the same type is received; if it is less than zero, the lowest type that is less than or equal to its absolute value is received.

The **msgflg** argument allows the "blocking message operation" to be performed if the IPC_NOWAIT flag is not set (**msgflg** & IPC_NOWAIT = 0); this would occur if there is not a message on the message queue of the desired type (**msgtyp**) to be received. If the IPC_NOWAIT flag is set, the system call will fail immediately when there is not a message of the desired type on the queue. Msgflg can also specify that the system call fail if the message is longer than the **size** to be received; this is done by not setting the MSG_NOERROR flag in the **msgflg** argument (**msgflg** & MSG_NOERROR =. 0). If the MSG_NOERROR flag is set, the message is truncated to the length specified by the **msgsz** argument of **msgrcv**().

Further details of this system call are discussed in the example program for it. If you have problems understanding the logic manipulations in this program, read the "Using **msgget**" section of this chapter; it goes into more detail than what would be practical to do for every system call.

Example Program

The example program in this section (Figure 15-6) is a menu driven program which allows all possible combinations of using the **msgsnd**() and **msgrcv**(2) system calls to be exercised.

From studying this program, you can observe the method of passing arguments and receiving return values. The user-written program requirements are pointed out.

This program begins (lines 5-9) by including the required header files as specified by the **msgop**(2) entry in the *Programmer's Reference Manual*. Note that in this program **errno** is declared as an external variable, and therefore, the **errno.h** header file does not have to be included.

Variable and structure names have been chosen to be as close as possible to those in the synopsis. Their declarations are self-explanatory. These names make the program more readable, and this is perfectly legal since they are local to the program. The variables declared for this program and their purposes are as follows:

sndbuf used as a buffer to contain a message to be sent (line 13); it uses the **msgbuf1** data structure as a template (lines 10-13) The **msgbuf1** structure (lines 10-13) is almost an exact duplicate of the **msgbuf** structure contained in the **msg.h** header file. The only difference is that the character array for **msgbuf1** contains the maximum message **size** (MSGMAX) for the 3B2 Computer where in **msgbuf** it is set to one (1) to satisfy the compiler. For this reason **msgbuf** cannot be used directly as a template for the user-written program. It is there so you can determine its members.

rcvbuf	used as a buffer to receive a message (line 13); it uses the **msgbuf1** data structure as a template (lines 10-13)
***msgp**	used as a pointer (line 13) to both the **sndbuf** and **rcvbuf** buffers
i	used as a counter for inputting characters from the keyboard, storing them in the array, and keeping track of the message length for the **msgsnd**() system call; it is also used as a counter to output the received message for the **msgrcv**() system call
c	used to receive the input character from the **getchar**() function (line 50)
flag	used to store the code of IPC_NOWAIT for the **msgsnd**() system call (line 61)
flags	used to store the code of the IPC_NOWAIT or MSG_NOERROR flags for the **msgrcv**() system call (line 117)
choice	used to store the code for sending or receiving (line 30)
rtrn	used to store the return values from all system calls
msqid	used to store and pass the desired message queue identifier for both system calls
msgsz	used to store and pass the **size** of the message to be sent or received
msgflg	used to pass the value of flag for sending or the value of flags for receiving
msgtyp	used for specifying the message type for sending, or used to pick a message type for receiving.

Note that a **msqid_ds** data structure is set up in the program (line 21) with a pointer which is initialized to point to it (line 22); this will allow the data structure members that are affected by message operations to be observed. They are observed by using the **msgctl**() (IPC_STAT) system call to get them for the program to print them out (lines 80-92 and lines 161-168).

The first thing the program prompts for is whether to send or receive a message. A corresponding code must be entered for the desired operation, and it is stored at the address of the choice variable (lines 23-30). Depending upon the code, the program proceeds as in the following **msgsnd** or **msgrcv** sections.

msgsnd

When the code is to send a message, the **msgp** pointer is initialized (line 33) to the address of the send data structure, **sndbuf**. Next, a message type must be entered for the message; it is stored at the address of the variable **msgtyp** (line 42), and then (line 43) it is put into the mtype member of the data structure pointed to by **msgp**.

The program now prompts for a message to be entered from the keyboard and enters a loop of getting and storing into the mtext array of the data structure (lines 48-51). This will continue until an end of file is recognized which for the **getchar**() function is a control-d (CTRL-D) immediately following a carriage return (<CR>). When this happens, the **size** of the message is determined by adding one to the **i** counter (lines 52, 53) as it stored the message beginning in the zero array element of mtext. Keep in mind that the message also contains the terminating characters, and the message will therefore appear to be three characters short of **msgsz**.

The message is immediately echoed from the mtext array of the **sndbuf** data structure to provide feedback (lines 54-56).

The next and final thing that must be decided is whether to set the IPC_NOWAIT flag. The program does this by requesting that a code of a 1 be entered for yes or anything else for no (lines 57-65). It is stored at the address of the flag variable. If a 1 is entered, IPC_NOWAIT is logically ORed with **msgflg**; otherwise, **msgflg** is set to zero.

The **msgsnd**() system call is performed (line 69). If it is unsuccessful, a failure message is displayed along with the error number (lines 70-72). If it is successful, the returned value is printed which should be zero (lines 73-76).

Every time a message is successfully sent, there are three members of the associated data structure which are updated. They are described as follows:

msg_qnum represents the total number of messages on the message queue; it is incremented by one.

msg_lspid contains the Process Identification (PID) number of the last process sending a message; it is set accordingly.

msg_stime contains the time in seconds since January 1, 1970, Greenwich Mean Time (GMT) of the last message sent; it is set accordingly.

These members are displayed after every successful message send operation (lines 79-92).

msgrcv

If the code specifies that a message is to be received, the program continues execution as in the following paragraphs.

The **msgp** pointer is initialized to the **rcvbuf** data structure (line 99).

Next, the message queue identifier of the message queue from which to receive the message is requested, and it is stored at the address of **msqid** (lines 100-103).

The message type is requested, and it is stored at the address of **msgtyp** (lines 104-107).

The code for the desired combination of control flags is requested next, and it is stored at the address of flags (lines 108-117). Depending upon the selected combination, **msgflg** is set accordingly (lines 118-133).

Finally, the number of bytes to be received is requested, and it is stored at the address of **msgsz** (lines 134-137).

The **msgrcv**() system call is performed (line 144). If it is unsuccessful, a message and error number is displayed (lines 145-148). If successful, a message indicates so, and the number of bytes returned is displayed followed by the received message (lines 153-159).

When a message is successfully received, there are three members of the associated data structure which are updated; they are described as follows:

msg_qnum contains the number of messages on the message queue; it is decremented by one.

msg_lrpid contains the process identification (PID) of the last process receiving a message; it is set accordingly.

msg_rtime contains the time in seconds since January 1, 1970, Greenwich Mean Time (GMT) that the last process received a message; it is set accordingly.

The example program for the **msgop()** system calls follows. It is suggested that the program be put into a source file called **msgop.c** and then into an executable file called **msgop**.

When compiling C programs that use floating point operations, the **-f** option should be used on the **cc** command line. If this option is not used, the program will compile successfully, but when the program is executed it will fail.

```
 1    /*This is a program to illustrate
 2    **the message operations, msgop(),
 3    **system call capabilities.
 4    */

 5    /*Include necessary header files.*/
 6    #include    <stdio.h>
 7    #include    <sys/types.h>
 8    #include    <sys/ipc.h>
 9    #include    <sys/msg.h>

10    struct msgbuf1 {
11        long    mtype;
12        char    mtext[8192];
13    } sndbuf, rcvbuf, *msgp;

14    /*Start of main C language program*/
15    main()
16    {
17        extern int errno;
18        int i, c, flag, flags, choice;
19        int rtrn, msqid, msgsz, msgflg;
20        long mtype, msgtyp;
21        struct msqid_ds msqid_ds, *buf;
22        buf = &msqid_ds;

23    /*Select the desired operation.*/
24    printf("Enter the corresponding\n");
25    printf("code to send or\n");
26    printf("receive a message:\n");
27    printf("Send    =  1\n");
28    printf("Receive=  2\n");
29    printf("Entry   =  ");
30    scanf("%d", &choice);
```

Figure 15-6: **msgop()** System Call Example (Sheet 1 of 5)

```
31    if(choice == 1) /*Send a message.*/
32    {
33       msgp = &sndbuf; /*Point to user send structure.*/

34       printf("\nEnter the msqid of\n");
35       printf("the message queue to\n");
36       printf("handle the message = ");
37       scanf("%d", &msqid);

38       /*Set the message type.*/
39       printf("\nEnter a positive integer\n");
40       printf("message type (long) for the\n");
41       printf("message = ");
42       scanf("%d", &msgtyp);
43       msgp->mtype = msgtyp;

44       /*Enter the message to send.*/
45       printf("\nEnter a message: \n");

46       /*A control-d (^d) terminates as
47         EOF.*/

48       /*Get each character of the message
49         and put it in the mtext array.*/
50       for(i = 0; ((c = getchar()) != EOF); i++)
51          sndbuf.mtext[i] = c;

52       /*Determine the message size.*/
53       msgsz = i + 1;

54       /*Echo the message to send.*/
55       for(i = 0; i < msgsz; i++)
56          putchar(sndbuf.mtext[i]);

57       /*Set the IPC_NOWAIT flag if
58         desired.*/
59       printf("\nEnter a 1 if you want the\n");
60       printf("the IPC_NOWAIT flag set:  ");
61       scanf("%d", &flag);
62       if(flag == 1)
63          msgflg |= IPC_NOWAIT;
64       else
65          msgflg = 0;
```

Figure 15-6: **msgop**() System Call Example (Sheet 2 of 5)

```
66        /*Check the msgflg.*/
67        printf("\nmsgflg = 0%o\n", msgflg);

68        /*Send the message.*/
69        rtrn = msgsnd(msqid, msgp, msgsz, msgflg);
70        if(rtrn == -1)
71        printf("\nMsgsnd failed.  Error = %d\n",
72           errno);
73        else {
74           /*Print the value of test which
75             should be zero for successful.*/
76           printf("\nValue returned = %d\n", rtrn);

77           /*Print the size of the message
78             sent.*/
79           printf("\nMsgsz = %d\n", msgsz);

80           /*Check the data structure update.*/
81           msgctl(msqid, IPC_STAT, buf);

82           /*Print out the affected members.*/

83           /*Print the incremented number of
84             messages on the queue.*/
85           printf("\nThe msg_qnum = %d\n",
86           buf->msg_qnum);
87           /*Print the process id of the last sender.*/
88           printf("The msg_lspid = %d\n",
89           buf->msg_lspid);
90           /*Print the last send time.*/
91           printf("The msg_stime = %d\n",
92           buf->msg_stime);
93        }
94     }

95     if(choice == 2)  /*Receive a message.*/
96     {
97        /*Initialize the message pointer
98          to the receive buffer.*/
99        msgp = &rcvbuf;

100          /*Specify the message queue which contains
101            the desired message.*/
102          printf("\nEnter the msqid = ");
103          scanf("%d", &msqid);
```

Figure 15-6: **msgop**() System Call Example (Sheet 3 of 5)

```
104         /*Specify the specific message on the queue
105           by using its type.*/
106         printf("\nEnter the msgtyp = ");
107         scanf("%d", &msgtyp);

108         /*Configure the control flags for the
109           desired actions.*/
110         printf("\nEnter the corresponding code\n");
111         printf("to select the desired flags: \n");
112         printf("No flags                    =  0\n");
113         printf("MSG_NOERROR                 =  1\n");
114         printf("IPC_NOWAIT                  =  2\n");
115         printf("MSG_NOERROR and IPC_NOWAIT =  3\n");
116         printf("                      Flags  =  ");
117         scanf("%d", &flags);

118         switch(flags) {
119            /*Set msgflg by ORing it with the appropriate
120              flags (constants).*/
121         case 0:
122            msgflg = 0;
123            break;
124         case 1:
125            msgflg |= MSG_NOERROR;
126            break;
127         case 2:
128            msgflg |= IPC_NOWAIT;
129            break;
130         case 3:
131            msgflg |= MSG_NOERROR | IPC_NOWAIT;
132            break;
133         }

134         /*Specify the number of bytes to receive.*/
135         printf("\nEnter the number of bytes\n");
136         printf("to receive (msgsz) = ");
137         scanf("%d", &msgsz);

138         /*Check the values for the arguments.*/
139         printf("\nmsqid =%d\n", msqid);
140         printf("\nmsgtyp = %d\n", msgtyp);
141         printf("\nmsgsz = %d\n", msgsz);
142         printf("\nmsgflg = 0%o\n", msgflg);

143         /*Call msgrcv to receive the message.*/
144         rtrn = msgrcv(msqid, msgp, msgsz, msgtyp, msgflg);
```

Figure 15-6: **msgop**() System Call Example (Sheet 4 of 5)

```
145         if(rtrn == -1)  {
146            printf("\nMsgrcv failed.  ");
147            printf("Error = %d\n", errno);
148         }
149         else {
150            printf ("\nMsgctl was successful\n");
151            printf("for msqid = %d\n",
152               msqid);

153            /*Print the number of bytes received,
154              it is equal to the return
155              value.*/
156            printf("Bytes received = %d\n", rtrn);

157            /*Print the received message.*/
158            for(i = 0; i<=rtrn; i++)
159               putchar(rcvbuf.mtext[i]);
160         }
161         /*Check the associated data structure.*/
162         msgctl(msqid, IPC_STAT, buf);
163         /*Print the decremented number of messages.*/
164         printf("\nThe msg_qnum = %d\n", buf->msg_qnum);
165         /*Print the process id of the last receiver.*/
166         printf("The msg_lrpid = %d\n", buf->msg_lrpid);
167         /*Print the last message receive time*/
168         printf("The msg_rtime = %d\n", buf->msg_rtime);
169      }
170   }
```

Figure 15-6: **msgop**() System Call Example (Sheet 5 of 5)

Semaphores

The semaphore type of IPC allows processes to communicate through the exchange of semaphore values. A semaphore is a positive integer (0 through 32,767). Since many applications require the use of more than one semaphore, the UNIX operating system has the ability to create sets or arrays of semaphores. A semaphore set can contain one or more semaphores up to a limit set by the system administrator. The tunable parameter, SEMMSL has a default value of 25. Semaphore sets are created by using the **semget**(2) system call.

The process performing the **semget**(2) system call becomes the owner/creator, determines how many semaphores are in the set, and sets the operation permissions for the set, including itself. This process can subsequently relinquish ownership of the set or change the operation permissions using the **semctl**(), semaphore control, system call. The creating process always remains the creator as long as the facility exists. Other processes with permission can use **semctl**() to perform other control functions.

Provided a process has alter permission, it can manipulate the semaphore(s). Each semaphore within a set can be manipulated in two ways with the **semop**(2) system call (which is documented in the *Programmer's Reference Manual*):

- incremented
- decremented

To increment a semaphore, an integer value of the desired magnitude is passed to the **semop**(2) system call. To decrement a semaphore, a minus (−) value of the desired magnitude is passed.

The UNIX operating system ensures that only one process can manipulate a semaphore set at any given time. Simultaneous requests are performed sequentially in an arbitrary manner.

A process can test for a semaphore value to be greater than a certain value by attempting to decrement the semaphore by one more than that value. If the process is successful, then the semaphore value is greater than that certain value. Otherwise, the semaphore value is not. While doing this, the process can have its execution suspended (IPC_NOWAIT flag not set) until the semaphore value would permit the operation (other processes increment the semaphore), or the semaphore facility is removed.

The ability to suspend execution is called a "blocking semaphore operation." This ability is also available for a process which is testing for a semaphore to become zero or equal to zero; only read permission is required for this test, and it is accomplished by passing a value of zero to the **semop**(2) system call.

On the other hand, if the process is not successful and the process does not request to have its execution suspended, it is called a "nonblocking semaphore operation." In this case, the process is returned a known error code (−1), and the external **errno** variable is set accordingly.

The blocking semaphore operation allows processes to communicate based on the values of semaphores at different points in time. Remember also that IPC facilities remain in the UNIX operating system until removed by a permitted process or until the system is reinitialized.

Operating on a semaphore set is done by using the **semop**(2), semaphore operation, system call.

When a set of semaphores is created, the first semaphore in the set is semaphore number zero. The last semaphore number in the set is one less than the total in the set.

An array of these "blocking/nonblocking operations" can be performed on a set containing more than one semaphore. When performing an array of operations, the "blocking/nonblocking operations" can be applied to any or all of the semaphores in the set. Also, the operations can be applied in any order of semaphore number. However, no operations are done until they can all be done successfully. This requirement means that preceding changes made to semaphore values in the set must be undone when a "blocking semaphore operation" on a semaphore in the set cannot be completed successfully; no changes are made until they can all be made. For example, if a process has successfully completed three of six operations on a set of ten semaphores but is "blocked" from performing the fourth operation, no changes are made to the set until the fourth and remaining operations are successfully performed. Additionally, any operation preceding or succeeding the "blocked" operation, including the blocked operation, can specify that at such time that all operations can be performed successfully, that the operation be undone. Otherwise, the operations are performed and the semaphores are changed or one "nonblocking operation" is unsuccessful and none are changed. All of this is commonly referred to as being "atomically performed."

The ability to undo operations requires the UNIX operating system to maintain an array of "undo structures" corresponding to the array of semaphore operations to be performed. Each semaphore operation which is to be undone has an associated adjust variable used for undoing the operation, if necessary.

Remember, any unsuccessful "nonblocking operation" for a single semaphore or a set of semaphores causes immediate return with no operations performed at all. When this occurs, a known error code (−1) is returned to the process, and the external variable **errno** is set accordingly.

System calls make these semaphore capabilities available to processes.The calling process passes arguments to a system call, and the system call either successfully or unsuccessfully performs its function. If the system call is successful, it performs its function and returns the appropriate information. Otherwise, a known error code (-1) is returned to the process, and the external variable **errno** is set accordingly.

Using Semaphores

Before semaphores can be used (operated on or controlled) a uniquely identified **data structure** and **semaphore set** (array) must be created. The unique identifier is called the semaphore identifier (**semid**); it is used to identify or reference a particular data structure and semaphore set.

The semaphore set contains a predefined number of structures in an array, one structure for each semaphore in the set. The number of semaphores (**nsems**) in a semaphore set is user selectable. The following members are in each structure within a semaphore set:

- semaphore text map address

- process identification (PID) performing last operation
- number of processes awaiting the semaphore value to become greater than its current value
- number of processes awaiting the semaphore value to equal zero

There is one associated data structure for the uniquely identified semaphore set. This data structure contains information related to the semaphore set as follows:

- operation permissions data (operation permissions structure)
- pointer to first semaphore in the set (array)
- number of semaphores in the set
- last semaphore operation time
- last semaphore change time

The C Programming Language data structure definition for the semaphore set (array member) is as follows:

```
struct sem
{
   ushort  semval;    /* semaphore text map address */
   short   sempid;    /* pid of last operation */
   ushort  semncnt;  /* # awaiting semval > cval */
   ushort  semzcnt;  /* # awaiting semval = 0 */
};
```

It is located in the **#include <sys/sem.h>** header file.

Likewise, the structure definition for the associated semaphore data structure is as follows:

```
struct semid_ds
{
  struct ipc_perm sem_perm;  /* operation permission struct */
  struct sem      *sem_base; /* ptr to first semaphore in set */
  ushort          sem_nsems; /* # of semaphores in set */
  time_t          sem_otime; /* last semop time */
  time_t          sem_ctime; /* last change time */
};
```

It is also located in the **#include <sys/sem.h>** header file. Note that the **sem_perm** member of this structure uses **ipc_perm** as a template. The breakout for the operation permissions data structure is shown in Figure 15-1.

The **ipc_perm** data structure is the same for all IPC facilities, and it is located in the **#include <sys/ipc.h>** header file. It is shown in the "Messages" section.

The **semget**(2) system call is used to perform two tasks when only the IPC_CREAT flag is set in the **semflg** argument that it receives:

- to get a new **semid** and create an associated data structure and semaphore set for it
- to return an existing **semid** that already has an associated data structure and semaphore set

The task performed is determined by the value of the **key** argument passed to the **semget**(2) system call. For the first task, if the **key** is not already in use for an existing **semid**, a new **semid** is returned with an associated data structure and semaphore set created for it provided no system tunable parameter would be exceeded.

There is also a provision for specifying a **key** of value zero (0) which is known as the private **key** (IPC_PRIVATE = 0); when specified, a new **semid** is always returned with an associated data structure and semaphore set created for it unless a system tunable parameter would be exceeded. When the **ipcs** command is performed, the KEY field for the **semid** is all zeros.

When performing the first task, the process which calls **semget()** becomes the owner/creator, and the associated data structure is initialized accordingly. Remember, ownership can be changed, but the creating process always remains the creator; see the "Controlling Semaphores" section in this chapter. The creator of the semaphore set also determines the initial operation permissions for the facility.

For the second task, if a **semid** exists for the **key** specified, the value of the existing **semid** is returned. If it is not desired to have an existing **semid** returned, a control command (IPC_EXCL) can be specified (set) in the **semflg** argument passed to the system call. The system call will fail if it is passed a value for the number of semaphores (**nsems**) that is greater than the number actually in the set; if you do not know how many semaphores are in the set, use 0 for **nsems**. The details of using this system call are discussed in the "Using **semget**" section of this chapter.

Once a uniquely identified semaphore set and data structure are created, semaphore operations [**semop**(2)] and semaphore control [**semctl**()] can be used.

Semaphore operations consist of incrementing, decrementing, and testing for zero. A single system call is used to perform these operations. It is called **semop**(). Refer to the "Operations on Semaphores" section in this chapter for details of this system call.

Semaphore control is done by using the **semctl**(2) system call. These control operations permit you to control the semaphore facility in the following ways:

- to return the value of a semaphore
- to set the value of a semaphore
- to return the process identification (PID) of the last process performing an operation on a semaphore set
- to return the number of processes waiting for a semaphore value to become greater than its current value
- to return the number of processes waiting for a semaphore value to equal zero
- to get all semaphore values in a set and place them in an array in user memory
- to set all semaphore values in a semaphore set from an array of values in user memory
- to place all data structure member values, status, of a semaphore set into user memory area
- to change operation permissions for a semaphore set
- to remove a particular **semid** from the UNIX operating system along with its associated data structure and semaphore set

Refer to the "Controlling Semaphores" section in this chapter for details of the **semctl**(2) system call.

Getting Semaphores

This section contains a detailed description of using the **semget**(2) system call along with an example program illustrating its use.

Using semget

The synopsis found in the **semget**(2) entry in the *Programmer's Reference Manual* is as follows:

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

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

The following line in the synopsis:

```
int semget (key, nsems, semflg)
```

informs you that **semget()** is a function with three formal arguments that returns an integer type value, upon successful completion (**semid**). The next two lines:

```
key_t  key;
int nsems, semflg;
```

declare the types of the formal arguments. **key_t** is declared by a **typedef** in the **types.h** header file to be an integer.

The integer returned from this system call upon successful completion is the semaphore set identifier (**semid**) that was discussed above.

As declared, the process calling the **semget()** system call must supply three actual arguments to be passed to the formal **key**, **nsems**, and **semflg** arguments.

A new **semid** with an associated semaphore set and data structure is provided if either

- **key** is equal to IPC_PRIVATE,

or

- **key** is passed a unique hexadecimal integer, and **semflg** ANDed with IPC_CREAT is TRUE.

The value passed to the **semflg** argument must be an integer type octal value and will specify the following:

- access permissions
- execution modes
- control fields (commands)

Access permissions determine the read/alter attributes and execution modes determine the user/group/other attributes of the **semflg** argument. They are collectively referred to as "operation permissions." Figure 15-7 reflects the numeric values (expressed in octal notation) for the valid operation permissions codes.

Operation Permissions	Octal Value
Read by User	00400
Alter by User	00200
Read by Group	00040
Alter by Group	00020
Read by Others	00004
Alter by Others	00002

Figure 15-7: Operation Permissions Codes

A specific octal value is derived by adding the octal values for the operation permissions desired. That is, if read by user and read/alter by others is desired, the code value would be 00406 (00400 plus 00006). There are constants **#define**'d in the **sem.h** header file which can be used for the user (OWNER). They are as follows:

```
SEM_A    0200    /* alter permission by owner */
SEM_R    0400    /* read permission by owner */
```

Control commands are predefined constants (represented by all uppercase letters). Figure 15-8 contains the names of the constants which apply to the **semget**(2) system call along with their values. They are also referred to as flags and are defined in the **ipc.h** header file.

Control Command	Value
IPC_CREAT	0001000
IPC_EXCL	0002000

Figure 15-8: Control Commands (Flags)

The value for **semflg** is, therefore, a combination of operation permissions and control commands. After determining the value for the operation permissions as previously described, the desired flag(s) can be specified. This specification is accomplished by bitwise ORing (|) them with the operation permissions; the bit positions and values for the control commands in relation to those of the operation permissions make this possible. It is illustrated as follows:

		Octal Value	Binary Value
IPC_CREAT	=	0 1 0 0 0	0 000 001 000 000 000
CW| ORed by User	=	0 0 4 0 0	0 000 000 100 000 000
semflg	=	0 1 4 0 0	0 000 001 100 000 000

The **semflg** value can be easily set by using the names of the flags in conjunction with the octal operation permissions value:

```
semid = semget (key, nsems, (IPC_CREAT | 0400));

semid = semget (key, nsems, (IPC_CREAT | IPC_EXCL | 0400));
```

As specified by the **semget**(2) entry in the *Programmer's Reference Manual*, success or failure of this system call depends upon the actual argument values for **key**, **nsems**, **semflg** or system tunable parameters. The system call will attempt to return a new **semid** if one of the following conditions is true:

- Key is equal to IPC_PRIVATE (0)
- Key does not already have a **semid** associated with it, and (**semflg** & IPC_CREAT) is "true" (not zero).

The **key** argument can be set to IPC_PRIVATE in the following ways:

```
semid = semget (IPC_PRIVATE, nsems, semflg);
```

or

```
semid = semget ( 0, nsems, semflg);
```

This alone will cause the system call to be attempted because it satisfies the first condition specified.

Exceeding the SEMMNI, SEMMNS, or SEMMSL system tunable parameters will always cause a failure. The SEMMNI system tunable parameter determines the maximum number of unique semaphore sets (**semid**'s) in the UNIX operating system. The SEMMNS system tunable parameter determines the maximum number of semaphores in all semaphore sets system wide. The SEMMSL system tunable parameter determines the maximum number of semaphores in each semaphore set.

The second condition is satisfied if the value for **key** is not already associated with a **semid**, and the bitwise ANDing of **semflg** and IPC_CREAT is "true" (not zero). This means that the **key** is unique (not in use) within the UNIX operating system for this facility type and that the IPC_CREAT flag is set (**semflg** | IPC_CREAT). The bitwise ANDing (&), which is the logical way of testing if a flag is set, is illustrated as follows:

```
semflg = x 1 x x x   (x = immaterial)
              & IPC_CREAT = 0 1 0 0 0

 result = 0 1 0 0 0   (not zero)
```

Since the result is not zero, the flag is set or "true." SEMMNI, SEMMNS, and SEMMSL apply here also, just as for condition one.

IPC_EXCL is another control command used in conjunction with IPC_CREAT to exclusively have the system call fail if, and only if, a **semid** exists for the specified key provided. This is necessary to prevent the process from thinking that it has received a new (unique) **semid** when it has not. In other words, when both IPC_CREAT and IPC_EXCL are specified, a new **semid** is returned if the system call is successful. Any value for **semflg** returns a new **semid** if the key equals zero (IPC_PRIVATE) and no system tunable parameters are exceeded.

Refer to the **semget**(2) manual page for specific associated data structure initialization for successful completion.

Example Program

The example program in this section (Figure 15-9) is a menu driven program which allows all possible combinations of using the **semget**(2) system call to be exercised.

From studying this program, you can observe the method of passing arguments and receiving return values. The user-written program requirements are pointed out.

This program begins (lines 4-8) by including the required header files as specified by the **semget**(2) entry in the *Programmer's Reference Manual*. Note that the **errno.h** header file is included as opposed to declaring **errno** as an external variable; either method will work.

Variable names have been chosen to be as close as possible to those in the synopsis. Their declarations are self-explanatory. These names make the program more readable, and this is perfectly legal since they are local to the program. The variables declared for this program and their purpose are as follows:

- **key**—used to pass the value for the desired key
- **opperm**—used to store the desired operation permissions
- **flags**—used to store the desired control commands (flags)
- **opperm_flags**—used to store the combination from the logical ORing of the **opperm** and **flags** variables; it is then used in the system call to pass the **semflg** argument
- **semid**—used for returning the semaphore set identification number for a successful system call or the error code (−1) for an unsuccessful one.

The program begins by prompting for a hexadecimal **key**, an octal operation permissions code, and the control command combinations (flags) which are selected from a menu (lines 15-32). All possible combinations are allowed even though they might not be viable. This allows observing the errors for illegal combinations.

Next, the menu selection for the flags is combined with the operation permissions, and the result is stored at the address of the **opperm_flags** variable (lines 36-52).

Then, the number of semaphores for the set is requested (lines 53-57), and its value is stored at the address of **nsems**.

The system call is made next, and the result is stored at the address of the **semid** variable (lines 60, 61).

Since the **semid** variable now contains a valid semaphore set identifier or the error code (−1), it is tested to see if an error occurred (line 63). If **semid** equals −1, a message indicates that an error resulted and the external **errno** variable is displayed (lines 65, 66). Remember that the external **errno** variable is only set when a system call fails; it should only be tested immediately following system calls.

If no error occurred, the returned semaphore set identifier is displayed (line 70).

The example program for the **semget**(2) system call follows. It is suggested that the source program file be named **semget.c** and that the executable file be named **semget**.

```
1    /*This is a program to illustrate
2    **the semaphore get, semget(),
3    **system call capabilities.*/

4    #include    <stdio.h>
5    #include    <sys/types.h>
6    #include    <sys/ipc.h>
7    #include    <sys/sem.h>
8    #include    <errno.h>

9    /*Start of main C language program*/
10    main()
11    {
12        key_t key;      /*declare as long integer*/
13        int opperm, flags, nsems;
14        int semid, opperm_flags;

15        /*Enter the desired key*/
16        printf("\nEnter the desired key in hex = ");
17        scanf("%x", &key);

18        /*Enter the desired octal operation
19              permissions.*/
20        printf("\nEnter the operation\n");
21        printf("permissions in octal = ");
22        scanf("%o", &opperm);

23                      /*Set the desired flags.*/
24                      printf("\nEnter corresponding number to\n");
25                      printf("set the desired flags:\n");
26                      printf("No flags                = 0\n");
27                      printf("IPC_CREAT               = 1\n");
28                      printf("IPC_EXCL                = 2\n");
29                      printf("IPC_CREAT and IPC_EXCL  = 3\n");
30                      printf("             Flags      = ");
31                      /*Get the flags to be set.*/
32                      scanf("%d", &flags);
```

Figure 15-9: **semget**() System Call Example (Sheet 1 of 2)

```
33    /*Error checking (debugging)*/
34    printf ("\nkey =0x%x, opperm = 0%o, flags = 0%o\n",
35       key, opperm, flags);
36    /*Incorporate the control fields (flags) with
37      the operation permissions.*/
38    switch (flags)
39    {
40    case 0:  /*No flags are to be set.*/
41       opperm_flags = (opperm | 0);
42       break;
43    case 1:   /*Set the IPC_CREAT flag.*/
44       opperm_flags = (opperm | IPC_CREAT);
45       break;
46    case 2:   /*Set the IPC_EXCL flag.*/
47       opperm_flags = (opperm | IPC_EXCL);
48       break;
49    case 3:  /*Set the IPC_CREAT and IPC_EXCL
50          flags.*/
51       opperm_flags = (opperm | IPC_CREAT | IPC_EXCL);
52    }

53    /*Get the number of semaphores for this set.*/
54    printf("\nEnter the number of\n");
55    printf("desired semaphores for\n");
56    printf("this set (25 max) = ");
57    scanf("%d", &nsems);

58    /*Check the entry.*/
59    printf("\nNsems = %d\n", nsems);

60    /*Call the semget system call.*/
61    semid = semget(key, nsems, opperm_flags);

62    /*Perform the following if the call is unsuccessful.*/
63    if(semid == -1)
64    {
65       printf("The semget system call failed!\n");
66       printf("The error number = %d\n", errno);
67    }
68    /*Return the semid upon successful completion.*/
69    else
70       printf("\nThe semid = %d\n", semid);
71    exit(0);
72 }
```

Figure 15-9: **semget**() System Call Example (Sheet 2 of 2)

Controlling Semaphores

This section contains a detailed description of using the **semctl**(2) system call along with an example program which allows all of its capabilities to be exercised.

Using semctl

The synopsis found in the **semctl**(2) entry in the *Programmer's Reference Manual* is as follows:

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

int semctl (semid, semnum, cmd, arg)
int semid, cmd;
int semnum;
union semun
{
        int val;
        struct semid_ds *bu;
        ushort array[];
} arg;
```

The **semctl**(2) system call requires four arguments to be passed to it, and it returns an integer value.

The **semid** argument must be a valid, non-negative, integer value that has already been created by using the **semget**(2) system call.

The **semnum** argument is used to select a semaphore by its number. This relates to array (atomically performed) operations on the set. When a set of semaphores is created, the first semaphore is number 0, and the last semaphore has the number of one less than the total in the set.

The **cmd** argument can be replaced by one of the following control commands (flags):

- GETVAL—return the value of a single semaphore within a semaphore set
- SETVAL—set the value of a single semaphore within a semaphore set
- GETPID—return the Process Identifier (PID) of the process that performed the last operation on the semaphore within a semaphore set
- GETNCNT—return the number of processes waiting for the value of a particular semaphore to become greater than its current value
- GETZCNT—return the number of processes waiting for the value of a particular semaphore to be equal to zero
- GETALL—return the values for all semaphores in a semaphore set
- SETALL—set all semaphore values in a semaphore set
- IPC_STAT—return the status information contained in the associated data structure for the specified **semid**, and place it in the data structure pointed to by the ***buf** pointer in the user memory area; **arg.buf** is the union member that contains the value of **buf**

- IPC_SET—for the specified semaphore set (**semid**), set the effective user/group identification and operation permissions
- IPC_RMID—remove the specified (**semid**) semaphore set along with its associated data structure.

A process must have an effective user identification of OWNER/CREATOR or super-user to perform an IPC_SET or IPC_RMID control command. Read/alter permission is required as applicable for the other control commands.

The **arg** argument is used to pass the system call the appropriate union member for the control command to be performed:

- **arg.val**
- **arg.buf**
- **arg.array**

The details of this system call are discussed in the example program for it. If you have problems understanding the logic manipulations in this program, read the "Using **semget**" section of this chapter; it goes into more detail than what would be practical to do for every system call.

Example Program

The example program in this section (Figure 15-10) is a menu driven program which allows all possible combinations of using the **semctl**(2) system call to be exercised.

From studying this program, you can observe the method of passing arguments and receiving return values. The user-written program requirements are pointed out.

This program begins (lines 5-9) by including the required header files as specified by the **semctl**(2) entry in the *Programmer's Reference Manual* Note that in this program **errno** is declared as an external variable, and therefore the **errno.h** header file does not have to be included.

Variable, structure, and union names have been chosen to be as close as possible to those in the synopsis. Their declarations are self-explanatory. These names make the program more readable, and this is perfectly legal since they are local to the program. Those declared for this program and their purpose are as follows:

- **semid_ds**—used to receive the specified semaphore set identifier's data structure when an IPC_STAT control command is performed
- **c**—used to receive the input values from the **scanf**(3S) function, (line 117) when performing a SETALL control command
- **i**—used as a counter to increment through the union **arg.array** when displaying the semaphore values for a GETALL (lines 97-99) control command, and when initializing the **arg.array** when performing a SETALL (lines 115-119) control command
- **length**—used as a variable to test for the number of semaphores in a set against the **i** counter variable (lines 97, 115)
- **uid**—used to store the IPC_SET value for the effective user identification
- **gid**—used to store the IPC_SET value for the effective group identification

- **mode**—used to store the IPC_SET value for the operation permissions
- **rtrn**—used to store the return integer from the system call which depends upon the control command or a −1 when unsuccessful
- **semid**—used to store and pass the semaphore set identifier to the system call
- **semnum**—used to store and pass the semaphore number to the system call
- **cmd**—used to store the code for the desired control command so that subsequent processing can be performed on it
- **choice**—used to determine which member (**uid, gid, mode**) for the IPC_SET control command that is to be changed
- **arg.val**—used to pass the system call a value to set (SETVAL) or to store (GETVAL) a value returned from the system call for a single semaphore (union member)
- **arg.buf**—a pointer passed to the system call which locates the data structure in the user memory area where the IPC_STAT control command is to place its return values, or where the IPC_SET command gets the values to set (union member)
- **arg.array**—used to store the set of semaphore values when getting (GETALL) or initializing (SETALL) (union member).

Note that the **semid_ds** data structure in this program (line 14) uses the data structure located in the **sem.h** header file of the same name as a template for its declaration. This is a perfect example of the advantage of local variables.

The **arg** union (lines 18-22) serves three purposes in one. The compiler allocates enough storage to hold its largest member. The program can then use the union as any member by referencing union members as if they were regular structure members. Note that the array is declared to have 25 elements (0 through 24).This number corresponds to the maximum number of semaphores allowed per set (SEMMSL), a system tunable parameter.

The next important program aspect to observe is that although the ***buf** pointer member (**arg.buf**) of the union is declared to be a pointer to a data structure of the **semid_ds** type, it must also be initialized to contain the address of the user memory area data structure (line 24). Because of the way this program is written, the pointer does not need to be reinitialized later. If it was used to increment through the array, it would need to be reinitialized just before calling the system call.

Now that all of the required declarations have been presented for this program, this is how it works.

First, the program prompts for a valid semaphore set identifier, which is stored at the address of the **semid** variable (lines 25-27). This is required for all **semctl**(2) system calls.

Then, the code for the desired control command must be entered (lines 28-42), and the code is stored at the address of the **cmd** variable. The code is tested to determine the control command for subsequent processing.

If the GETVAL control command is selected (code 1), a message prompting for a semaphore number is displayed (lines 49, 50). When it is entered, it is stored at the address of the **semnum** variable (line 51). Then, the system call is performed, and the semaphore value is displayed (lines 52-55). If the system call is successful, a message indicates this along with the semaphore set identifier used (lines 195, 196); if the

system call is unsuccessful, an error message is displayed along with the value of the external **errno** variable (lines 191-193).

If the SETVAL control command is selected (code 2), a message prompting for a semaphore number is displayed (lines 56, 57). When it is entered, it is stored at the address of the **semnum** variable (line 58). Next, a message prompts for the value to which the semaphore is to be set, and it is stored as the **arg.val** member of the union (lines 59, 60). Then, the system call is performed (lines 61, 63). Depending upon success or failure, the program returns the same messages as for GETVAL above.

If the GETPID control command is selected (code 3), the system call is made immediately since all required arguments are known (lines 64-67), and the PID of the process performing the last operation is displayed. Depending upon success or failure, the program returns the same messages as for GETVAL above.

If the GETNCNT control command is selected (code 4), a message prompting for a semaphore number is displayed (lines 68-72). When entered, it is stored at the address of the **semnum** variable (line 73). Then, the system call is performed, and the number of processes waiting for the semaphore to become greater than its current value is displayed (lines 74-77). Depending upon success or failure, the program returns the same messages as for GETVAL above.

If the GETZCNT control command is selected (code 5), a message prompting for a semaphore number is displayed (lines 78-81). When it is entered, it is stored at the address of the **semnum** variable (line 82). Then the system call is performed, and the number of processes waiting for the semaphore value to become equal to zero is displayed (lines 83, 86). Depending upon success or failure, the program returns the same messages as for GETVAL above.

If the GETALL control command is selected (code 6), the program first performs an IPC_STAT control command to determine the number of semaphores in the set (lines 88-93). The length variable is set to the number of semaphores in the set (line 91). Next, the system call is made and, upon success, the **arg.array** union member contains the values of the semaphore set (line 96). Now, a loop is entered which displays each element of the **arg.array** from zero to one less than the value of length (lines 97-103). The semaphores in the set are displayed on a single line, separated by a space. Depending upon success or failure, the program returns the same messages as for GETVAL above.

If the SETALL control command is selected (code 7), the program first performs an IPC_STAT control command to determine the number of semaphores in the set (lines 106-108). The length variable is set to the number of semaphores in the set (line 109). Next, the program prompts for the values to be set and enters a loop which takes values from the keyboard and initializes the **arg.array** union member to contain the desired values of the semaphore set (lines 113-119). The loop puts the first entry into the array position for semaphore number zero and ends when the semaphore number that is filled in the array equals one less than the value of length. The system call is then made (lines 120-122). Depending upon success or failure, the program returns the same messages as for GETVAL above.

If the IPC_STAT control command is selected (code 8), the system call is performed (line 127), and the status information returned is printed out (lines 128-139); only the members that can be set are printed out in this program. Note that if the system call is unsuccessful, the status information of the last successful one is printed out. In addition, an error message is displayed, and the **errno** variable is printed out (lines 191, 192).

If the IPC_SET control command is selected (code 9), the program gets the current status information for the semaphore set identifier specified (lines 143-146). This is necessary because this example program provides for changing only one member at a time, and the **semctl**(2) system call changes all of them. Also, if an invalid value happened to be stored in the user memory area for one of these members, it would cause repetitive failures for this control command until corrected. The next thing the program does is to prompt for a code corresponding to the member to be changed (lines 147-153). This code is stored at the address of the choice variable (line 154). Now, depending upon the member picked, the program prompts for the new value (lines 155-178). The value is placed at the address of the appropriate member in the user memory area data structure, and the system call is made (line 181). Depending upon success or failure, the program returns the same messages as for GETVAL above.

If the IPC_RMID control command (code 10) is selected, the system call is performed (lines 183-185). The **semid** along with its associated data structure and semaphore set is removed from the UNIX operating system. Depending upon success or failure, the program returns the same messages as for the other control commands.

The example program for the **semctl**(2) system call follows. It is suggested that the source program file be named **semctl.c** and that the executable file be named **semctl**.

```
1    /*This is a program to illustrate
2    **the semaphore control, semctl(),
3    **system call capabilities.
4    */

5    /*Include necessary header files.*/
6    #include    <stdio.h>
7    #include    <sys/types.h>
8    #include    <sys/ipc.h>
9    #include    <sys/sem.h>

10    /*Start of main C language program*/
11    main()
12    {
13        extern int errno;
14        struct semid_ds semid_ds;
15        int c, i, length;
16        int uid, gid, mode;
17        int retrn, semid, semnum, cmd, choice;
18        union semun  {
19            int val;
20            struct semid_ds *buf;
21            ushort array[25];
22        } arg;

23        /*Initialize the data structure pointer.*/
24        arg.buf = &semid_ds;

25        /*Enter the semaphore ID.*/
26        printf("Enter the semid = ");
27        scanf("%d", &semid);

28        /*Choose the desired command.*/
29        printf("\nEnter the number for\n");
30        printf("the desired cmd:\n");
31        printf("GETVAL      =  1\n");
32        printf("SETVAL      =  2\n");
33        printf("GETPID      =  3\n");
34        printf("GETNCNT     =  4\n");
35        printf("GETZCNT     =  5\n");
36        printf("GETALL      =  6\n");
37        printf("SETALL      =  7\n");
38        printf("IPC_STAT    =  8\n");
39        printf("IPC_SET     =  9\n");
40        printf("IPC_RMID    =  10\n");
41        printf("Entry       =  ");
42        scanf("%d", &cmd);
```

Figure 15-10: **semctl**() System Call Example (Sheet 1 of 5)

```
43          /*Check entries.*/
44          printf ("\nsemid =%d, cmd = %d\n\n",
45              semid, cmd);

46          /*Set the command and do the call.*/
47          switch (cmd)
48          {

49          case 1: /*Get a specified value.*/
50              printf("\nEnter the semnum = ");
51              scanf("%d", &semnum);
52              /*Do the system call.*/
53              retrn = semctl(semid, semnum, GETVAL, 0);
54              printf("\nThe semval = %d\n", retrn);
55              break;
56          case 2: /*Set a specified value.*/
57              printf("\nEnter the semnum = ");
58              scanf("%d", &semnum);
59              printf("\nEnter the value = ");
60              scanf("%d", &arg.val);
61              /*Do the system call.*/
62              retrn = semctl(semid, semnum, SETVAL, arg.val);
63              break;
64          case 3: /*Get the process ID.*/
65              retrn = semctl(semid, 0, GETPID, 0);
66              printf("\nThe sempid = %d\n", retrn);
67              break;
68          case 4: /*Get the number of processes
69                    waiting for the semaphore to
70                    become greater than its current
71                            value.*/
72              printf("\nEnter the semnum = ");
73              scanf("%d", &semnum);
74              /*Do the system call.*/
75              retrn = semctl(semid, semnum, GETNCNT, 0);
76              printf("\nThe semncnt = %d", retrn);
77              break;

78          case 5: /*Get the number of processes
79                       waiting for the semaphore
80                       value to become zero.*/
81              printf("\nEnter the semnum = ");
82              scanf("%d", &semnum);
83              /*Do the system call.*/
84              retrn = semctl(semid, semnum, GETZCNT, 0);
85              printf("\nThe semzcnt = %d", retrn);
86              break;
```

Figure 15-10: **semctl**() System Call Example (Sheet 2 of 5)

```
87          case 6: /*Get all of the semaphores.*/
88              /*Get the number of semaphores in
89                the semaphore set.*/
90              retrn = semctl(semid, 0, IPC_STAT, arg.buf);
91              length = arg.buf->sem_nsems;
92              if(retrn == -1)
93                  goto ERROR;
94              /*Get and print all semaphores in the
95                specified set.*/
96              retrn = semctl(semid, 0, GETALL, arg.array);
97              for (i = 0; i < length; i++)
98              {
99                  printf("%d", arg.array[i]);
100                 /*Seperate each
101                   semaphore.*/
102                 printf("%c", ' ');
103             }
104             break;

105         case 7: /*Set all semaphores in the set.*/
106             /*Get the number of semaphores in
107               the set.*/
108             retrn = semctl(semid, 0, IPC_STAT, arg.buf);
109             length = arg.buf->sem_nsems;
110             printf("Length = %d\n", length);
111             if(retrn == -1)
112                goto ERROR;
113             /*Set the semaphore set values.*/
114             printf("\nEnter each value:\n");
115             for(i = 0; i < length ; i++)
116             {
117                scanf("%d", &c);
118                arg.array[i] = c;
119             }
120             /*Do the system call.*/
121             retrn = semctl(semid, 0, SETALL, arg.array);
122             break;
```

Figure 15-10: **semctl**() System Call Example (Sheet 3 of 5)

```
123       case 8: /*Get the status for the semaphore set.*/
125          /*Get and print the current status values.*/
127          retrn = semctl(semid, 0, IPC_STAT, arg.buf);
128          printf ("\nThe USER ID = %d\n",
129             arg.buf->sem_perm.uid);
130          printf ("The GROUP ID = %d\n",
131             arg.buf->sem_perm.gid);
132          printf ("The operation permissions = 0%o\n",
133             arg.buf->sem_perm.mode);
134          printf ("The number of semaphores in set = %d\n",
135             arg.buf->sem_nsems);
136          printf ("The last semop time = %d\n",
137             arg.buf->sem_otime);

138             printf ("The last change time  = %d\n",
139                 arg.buf->sem_ctime);
140             break;

141        case 9:    /*Select and change the desired
142                      member of the data structure.*/
143             /*Get the current status values.*/
144             retrn = semctl(semid, 0, IPC_STAT, arg.buf);
145             if(retrn == -1)
146                 goto ERROR;
147             /*Select the member to change.*/
148             printf("\nEnter the number for the\n");
149             printf("member to be changed:\n");
150             printf("sem_perm.uid   = 1\n");
151             printf("sem_perm.gid   = 2\n");
152             printf("sem_perm.mode  = 3\n");
153             printf("Entry          = ");
154             scanf("%d", &choice);
155             switch(choice){

156             case 1: /*Change the user ID.*/
157                 printf("\nEnter USER ID = ");
158                 scanf ("%d", &uid);
159                 arg.buf->sem_perm.uid = uid;
160                 printf("\nUSER ID = %d\n",
161                     arg.buf->sem_perm.uid);
162                 break;

163             case 2: /*Change the group ID.*/
164                 printf("\nEnter GROUP ID = ");
165                 scanf("%d", &gid);
166                 arg.buf->sem_perm.gid = gid;
167                 printf("\nGROUP ID = %d\n",
168                     arg.buf->sem_perm.gid);
169                 break;
```

Figure 15-10: **semctl()** System Call Example (Sheet 4 of 5)

```
170         case 3: /*Change the mode portion of
171                   the operation
172                         permissions.*/
173            printf("\nEnter MODE = ");
174            scanf("%o", &mode);
175            arg.buf->sem_perm.mode = mode;
176            printf("\nMODE = 0%o\n",
177            arg.buf->sem_perm.mode);
178            break;
179         }
180         /*Do the change.*/
181         retrn = semctl(semid, 0, IPC_SET, arg.buf);
182         break;
183      case 10: /*Remove the semid along with its
184                 data structure.*/
185         retrn = semctl(semid, 0, IPC_RMID, 0);
186      }
187      /*Perform the following if the call is unsuccessful.*/
188      if(retrn == -1)
189      {
190   ERROR:
191         printf ("\n\nThe semctl system call failed!\n");
192         printf ("The error number = %d\n", errno);
193         exit(0);
194      }
195      printf ("\n\nThe semctl system call was successful\n");
196      printf ("for semid = %d\n", semid);
197      exit (0);
198   }
```

Figure 15-10: **semctl**() System Call Example (Sheet 5 of 5)

Operations on Semaphores

This section contains a detailed description of using the **semop**(2) system call along with an example program which allows all of its capabilities to be exercised.

Using semop

The synopsis found in the **semop**(2) entry in the *Programmer's Reference Manual* is as follows:

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

int semop (semid, sops, nsops)
int semid;
struct sembuf **sops;
unsigned nsops;
```

The **semop**(2) system call requires three arguments to be passed to it, and it returns an integer value.

Upon successful completion, a zero value is returned and when unsuccessful it returns a −1.

The **semid** argument must be a valid, non-negative, integer value. In other words, it must have already been created by using the **semget**(2) system call.

The **sops** argument is a pointer to an array of structures in the user memory area that contains the following for each semaphore to be changed:

- the semaphore number
- the operation to be performed
- the control command (flags)

The ***sops** declaration means that a pointer can be initialized to the address of the array, or the array name can be used since it is the address of the first element of the array. **Sembuf** is the *tag* name of the data structure used as the template for the structure members in the array; it is located in the **#include <sys/sem.h>** header file.

The **nsops** argument specifies the length of the array (the number of structures in the array). The maximum **size** of this array is determined by the SEMOPM system tunable parameter. Therefore, a maximum of SEMOPM operations can be performed for each **semop**(2) system call.

The semaphore number determines the particular semaphore within the set on which the operation is to be performed.

The operation to be performed is determined by the following:

- a positive integer value means to increment the semaphore value by its value
- a negative integer value means to decrement the semaphore value by its value
- a value of zero means to test if the semaphore is equal to zero

The following operation commands (flags) can be used:

- IPC_NOWAIT—this operation command can be set for any operations in the array. The system call will return unsuccessfully without changing any semaphore values at all if any operation for which IPC_NOWAIT is set cannot be performed successfully. The system call will be unsuccessful when trying to decrement a semaphore more than its current value, or when testing for a semaphore to be equal to zero when it is not.
- SEM_UNDO—this operation command allows any operations in the array to be undone when any operation in the array is unsuccessful and does not have the IPC_NOWAIT flag set. That is, the blocked operation waits until it can

perform its operation; and when it and all succeeding operations are successful, all operations with the SEM_UNDO flag set are undone. Remember, no operations are performed on any semaphores in a set until all operations are successful. Undoing is accomplished by using an array of adjust values for the operations that are to be undone when the blocked operation and all subsequent operations are successful.

Example Program

The example program in this section (Figure 15-11) is a menu driven program which allows all possible combinations of using the **semop**(2) system call to be exercised.

From studying this program, you can observe the method of passing arguments and receiving return values. The user-written program requirements are pointed out.

This program begins (lines 5-9) by including the required header files as specified by the **shmop**(2) entry in the *Programmer's Reference Manual* Note that in this program **errno** is declared as an external variable, and therefore, the **errno.h** header file does not have to be included.

Variable and structure names have been chosen to be as close as possible to those in the synopsis. Their declarations are self-explanatory. These names make the program more readable, and this is perfectly legal since the declarations are local to the program. The variables declared for this program and their purpose are as follows:

- **sembuf**[10]—used as an array buffer (line 14) to contain a maximum of ten **sembuf** type structures; ten equals SEMOPM, the maximum number of operations on a semaphore set for each **semop**(2) system call
- ***sops**—used as a pointer (line 14) to **sembuf**[10] for the system call and for accessing the structure members within the array
- **rtrn**—used to store the return values from the system call
- **flags**—used to store the code of the IPC_NOWAIT or SEM_UNDO flags for the **semop**(2) system call (line 60)
- **i**—used as a counter (line 32) for initializing the structure members in the array, and used to print out each structure in the array (line 79)
- **nsops**—used to specify the number of semaphore operations for the system call—must be less than or equal to SEMOPM
- **semid**—used to store the desired semaphore set identifier for the system call

First, the program prompts for a semaphore set identifier that the system call is to perform operations on (lines 19-22). Semid is stored at the address of the **semid** variable (line 23).

A message is displayed requesting the number of operations to be performed on this set (lines 25-27). The number of operations is stored at the address of the **nsops** variable (line 28).

Next, a loop is entered to initialize the array of structures (lines 30-77). The semaphore number, operation, and operation command (flags) are entered for each structure in the array. The number of structures equals the number of semaphore operations (**nsops**) to be performed for the system call, so **nsops** is tested against the **i** counter for loop control. Note that **sops** is used as a pointer to each element (structure) in the array, and **sops** is incremented just like **i**. **sops** is then used to point to each member in the structure for setting them.

After the array is initialized, all of its elements are printed out for feedback (lines 78-85).

The **sops** pointer is set to the address of the array (lines 86, 87). Sem**buf** could be used directly, if desired, instead of **sops** in the system call.

The system call is made (line 89), and depending upon success or failure, a corresponding message is displayed. The results of the operation(s) can be viewed by using the **semctl**() GETALL control command.

The example program for the **semop**(2) system call follows. It is suggested that the source program file be named **semop.c** and that the executable file be named **semop**.

```
1    /*This is a program to illustrate
2    **the semaphore operations, semop(),
3    **system call capabilities.
4    */

5    /*Include necessary header files.*/
6    #include    <stdio.h>
7    #include    <sys/types.h>
8    #include    <sys/ipc.h>
9    #include    <sys/sem.h>
10   /*Start of main C language program*/
11   main()
12   {
13       extern int errno;
14       struct sembuf sembuf[10], *sops;
15       char string[];
16       int retrn, flags, sem_num, i, semid;
17       unsigned nsops;
18       sops = sembuf; /*Pointer to array sembuf.*/

19       /*Enter the semaphore ID.*/
20       printf("\nEnter the semid of\n");
21       printf("the semaphore set to\n");
22       printf("be operated on = ");
23       scanf("%d", &semid);
24       printf("\nsemid = %d", semid);

25   /*Enter the number of operations.*/
26   printf("\nEnter the number of semaphore\n");
27   printf("operations for this set = ");
28   scanf("%d", &nsops);
29   printf("\nnosops = %d", nsops);

30   /*Initialize the array for the
31     number of operations to be performed.*/
32   for(i = 0; i < nsops; i++, sops++)
33   {

34       /*This determines the semaphore in
35         the semaphore set.*/
36       printf("\nEnter the semaphore\n");
37       printf("number (sem_num) = ");
38       scanf("%d", &sem_num);
39       sops->sem_num = sem_num;
40       printf("\nThe sem_num = %d", sops->sem_num);
```

Figure 15-11: **semop**(2) System Call Example (Sheet 1 of 3)

```
41        /*Enter a (-)number to decrement,
42          an unsigned number (no +) to increment,
43          or zero to test for zero.  These values
44          are entered into a string and converted
45          to integer values.*/
46        printf("\nEnter the operation for\n");
47        printf("the semaphore (sem_op) = ");
48        scanf("%s", string);
49        sops->sem_op = atoi(string);
50        printf("\nsem_op = %d\n", sops->sem_op);

51        /*Specify the desired flags.*/
52        printf("\nEnter the corresponding\n");
53        printf("number for the desired\n");
54        printf("flags:\n");
55        printf("No flags                    = 0\n");
56        printf("IPC_NOWAIT                  = 1\n");
57        printf("SEM_UNDO                    = 2\n");
58        printf("IPC_NOWAIT and SEM_UNDO     = 3\n");
59        printf("                Flags       = ");
60        scanf("%d", &flags);

61        switch(flags)
62        {
63        case 0:
64           sops->sem_flg = 0;
65           break;
66        case 1:
67           sops->sem_flg = IPC_NOWAIT;
68           break;
69        case 2:
70           sops->sem_flg = SEM_UNDO;
71           break;
72        case 3:
73           sops->sem_flg = IPC_NOWAIT | SEM_UNDO;
74           break;
75        }
76        printf("\nFlags = 0%o\n", sops->sem_flg);
77    }
```

Figure 15-11: **semop**(2) System Call Example (Sheet 2 of 3)

```
78     /*Print out each structure in the array.*/
79     for(i = 0; i < nsops; i++)
80     {
81        printf("\nsem_num = %d\n", sembuf[i].sem_num);
82        printf("sem_op = %d\n", sembuf[i].sem_op);
83        printf("sem_flg = %o\n", sembuf[i].sem_flg);
84        printf("%c", ' ');
85     }

86     sops = sembuf; /*Reset the pointer to
87                       sembuf[0].*/

88     /*Do the semop system call.*/
89     retrn = semop(semid, sops, nsops);
90     if(retrn == -1)  {
91        printf("\nSemop failed.  ");
92        printf("Error = %d\n", errno);
93     }
94     else {
95        printf ("\nSemop was successful\n");
96        printf("for semid = %d\n", semid);

97        printf("Value returned = %d\n", retrn);
98     }
99  }
```

Figure 15-11: **semop**(2) System Call Example (Sheet 3 of 3)

Shared Memory

The shared memory type of IPC allows two or more processes (executing programs) to share memory and consequently the data contained there. This is done by allowing processes to set up access to a common virtual memory address space. This sharing occurs on a segment basis, which is memory management hardware dependent.

This sharing of memory provides the fastest means of exchanging data between processes.

A process initially creates a shared memory segment facility using the **shmget**(2) system call. Upon creation, this process sets the overall operation permissions for the shared memory segment facility, sets its size in bytes, and can specify that the shared memory segment is for reference only (read-only) upon attachment. If the memory segment is not specified to be for reference only, all other processes with appropriate operation permissions can read from or write to the memory segment.

There are two operations that can be performed on a shared memory segment:

- **shmat**(2) — shared memory attach
- **shmdt**(2) — shared memory detach

Shared memory attach allows processes to associate themselves with the shared memory segment if they have permission. They can then read or write as allowed.

Shared memory detach allows processes to disassociate themselves from a shared memory segment. Therefore, they lose the ability to read from or write to the shared memory segment.

The original owner/creator of a shared memory segment can relinquish ownership to another process using the **shmctl**(2) system call. However, the creating process remains the creator until the facility is removed or the system is reinitialized. Other processes with permission can perform other functions on the shared memory segment using the **shmctl**(2) system call.

System calls, which are documented in the *Programmer's Reference Manual*, make these shared memory capabilities available to processes. The calling process passes arguments to a system call, and the system call either successfully or unsuccessfully performs its function. If the system call is successful, it performs its function and returns the appropriate information. Otherwise, a known error code (−1) is returned to the process, and the external variable **errno** is set accordingly.

Using Shared Memory

The sharing of memory between processes occurs on a virtual segment basis. There is one and only one instance of an individual shared memory segment existing in the UNIX operating system at any point in time.

Before sharing of memory can be realized, a uniquely identified shared memory segment and data structure must be created. The unique identifier created is called the shared memory identifier (**shmid**); it is used to identify or reference the associated data structure. The data structure includes the following for each shared memory segment:

- operation permissions

- segment size
- segment descriptor
- process identification performing last operation
- process identification of creator
- current number of processes attached
- in memory number of processes attached
- last attach time
- last detach time
- last change time

The C Programming Language data structure definition for the shared memory segment data structure is located in the **/usr/include/sys/shm.h** header file. It is as follows:

```
/*
**      There is a shared mem id data structure for
**      each segment in the system.
*/

struct shmid_ds {
  struct ipc_perm shm_perm; /* operation permission struct */
  int     shm_segsz;        /* segment size */
  struct region   *shm_reg;   /* ptr to region structure */
  char    pad[4];       /* for swap compatibility */
  ushort shm_lpid;          /* pid of last shmop */
  ushort shm_cpid;          /* pid of creator */
  ushort shm_nattch;        /* used only for shminfo */
  ushort shm_cnattch;       /* used only for shminfo */
  time_t shm_atime;         /* last shmat time */
  time_t shm_dtime;         /* last shmdt time */
  time_t shm_ctime;         /* last change time */
};
```

Note that the **shm_perm** member of this structure uses **ipc_perm** as a template. The breakout for the operation permissions data structure is shown in Figure 15-1.

The **ipc_perm** data structure is the same for all IPC facilities, and it is located in the **#include <sys/ipc.h>** header file. It is shown in the introduction section of "Messages."

Figure 15-12 is a table that shows the shared memory state information.

Shared Memory States

Lock Bit	Swap Bit	Allocated Bit	Implied State
0	0	0	Unallocated Segment
0	0	1	Incore
0	1	0	Unused
0	1	1	On Disk
1	0	1	Locked Incore
1	1	0	Unused
1	0	0	Unused
1	1	1	Unused

Figure 15-12: Shared Memory State Information

The implied states of Figure 15-12 are as follows:

- **Unallocated Segment**—the segment associated with this segment descriptor has not been allocated for use.
- **Incore**—the shared segment associated with this descriptor has been allocated for use. Therefore, the segment does exist and is currently resident in memory.
- **On Disk**—the shared segment associated with this segment descriptor is currently resident on the swap device.
- **Locked Incore**—the shared segment associated with this segment descriptor is currently locked in memory and will not be a candidate for swapping until the segment is unlocked. Only the super-user may lock and unlock a shared segment.
- **Unused**—this state is currently unused and should never be encountered by the normal user in shared memory handling.

The **shmget**(2) system call is used to perform two tasks when only the IPC_CREAT flag is set in the **shmflg** argument that it receives:

- to get a new **shmid** and create an associated shared memory segment data structure for it
- to return an existing **shmid** that already has an associated shared memory segment data structure

The task performed is determined by the value of the **key** argument passed to the **shmget**(2) system call. For the first task, if the **key** is not already in use for an existing **shmid**, a new **shmid** is returned with an associated shared memory segment data structure created for it provided no system tunable parameters would be exceeded.

There is also a provision for specifying a **key** of value zero which is known as the private **key** (IPC_PRIVATE = 0); when specified, a new **shmid** is always returned with an associated shared memory segment data structure created for it unless a system tunable parameter would be exceeded. When the **ipcs** command is performed, the KEY field for the **shmid** is all zeros.

For the second task, if a **shmid** exists for the **key** specified, the value of the existing **shmid** is returned. If it is not desired to have an existing **shmid** returned, a control command (IPC_EXCL) can be specified (set) in the **shmflg** argument passed to the system call. The details of using this system call are discussed in the "Using **shmget**" section of this chapter.

When performing the first task, the process that calls **shmget** becomes the owner/creator, and the associated data structure is initialized accordingly. Remember, ownership can be changed, but the creating process always remains the creator; see the "Controlling Shared Memory" section in this chapter. The creator of the shared memory segment also determines the initial operation permissions for it.

Once a uniquely identified shared memory segment data structure is created, shared memory segment operations [**shmop**()] and control [**shmctl**(2)] can be used.

Shared memory segment operations consist of attaching and detaching shared memory segments. System calls are provided for each of these operations; they are **shmat**(2) and **shmdt**(2). Refer to the "Operations for Shared Memory" section in this chapter for details of these system calls.

Shared memory segment control is done by using the **shmctl**(2) system call. It permits you to control the shared memory facility in the following ways:

- to determine the associated data structure status for a shared memory segment (**shmid**)
- to change operation permissions for a shared memory segment
- to remove a particular **shmid** from the UNIX operating system along with its associated shared memory segment data structure
- to lock a shared memory segment in memory
- to unlock a shared memory segment

Refer to the "Controlling Shared Memory" section in this chapter for details of the **shmctl**(2) system call.

Getting Shared Memory Segments

This section gives a detailed description of using the **shmget**(2) system call along with an example program illustrating its use.

Using shmget

The synopsis found in the **shmget**(2) entry in the *Programmer's Reference Manual* is as follows:

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

int  shmget (key, size, shmflg)
key_t  key;
int size, shmflg;
```

All of these include files are located in the **/usr/include/sys** directory of the UNIX operating system. The following line in the synopsis:

```
int shmget (key, size, shmflg)
```

informs you that **shmget**(2) is a function with three formal arguments that returns an integer type value, upon successful completion (**shmid**). The next two lines:

```
key_t  key;
int size, shmflg;
```

declare the types of the formal arguments. The variable **key_t** is declared by a **typedef** in the **types.h** header file to be an integer.

The integer returned from this function upon successful completion is the shared memory identifier (**shmid**) that was discussed earlier.

As declared, the process calling the **shmget**(2) system call must supply three arguments to be passed to the formal **key**, **size**, and **shmflg** arguments.

A new **shmid** with an associated shared memory data structure is provided if either

- **key** is equal to IPC_PRIVATE,

or

- **key** is passed a unique hexadecimal integer, and **shmflg** ANDed with IPC_CREAT is TRUE.

The value passed to the **shmflg** argument must be an integer type octal value and will specify the following:

- access permissions
- execution modes
- control fields (commands)

Access permissions determine the read/write attributes and execution modes determine the user/group/other attributes of the **shmflg** argument. They are collectively referred to as "operation permissions." Figure 15-13 reflects the numeric values (expressed in octal notation) for the valid operation permissions codes.

Operation Permissions	Octal Value
Read by User	00400
Write by User	00200
Read by Group	00040
Write by Group	00020
Read by Others	00004
Write by Others	00002

Figure 15-13: Operation Permissions Codes

A specific octal value is derived by adding the octal values for the operation permissions desired. That is, if read by user and read/write by others is desired, the code value would be 00406 (00400 plus 00006). There are constants located in the **shm.h** header file which can be used for the user (OWNER). They are as follows:

```
SHM_R       0400
SHM_W       0200
```

Control commands are predefined constants (represented by all uppercase letters). Figure 15-14 contains the names of the constants that apply to the **shmget()** system call along with their values. They are also referred to as flags and are defined in the **ipc.h** header file.

Control Command	Value
IPC_CREAT	0001000
IPC_EXCL	0002000

Figure 15-14: Control Commands (Flags)

The value for **shmflg** is, therefore, a combination of operation permissions and control commands. After determining the value for the operation permissions as previously described, the desired flag(s) can be specified. This is accomplished by bitwise ORing (|) them with the operation permissions; the bit positions and values for the control commands in relation to those of the operation permissions make this possible. It is illustrated as follows:

		Octal Value	Binary Value
IPC_CREAT	=	0 1 0 0 0	0 000 001 000 000 000
| ORed by User	=	0 0 4 0 0	0 000 000 100 000 000
shmflg	=	0 1 4 0 0	0 000 001 100 000 000

The **shmflg** value can be easily set by using the names of the flags in conjunction with the octal operation permissions value:

```
shmid = shmget (key, size, (IPC_CREAT | 0400));

shmid = shmget (key, size, (IPC_CREAT | IPC_EXCL | 0400));
```

As specified by the **shmget**(2) entry in the *Programmer's Reference Manual*, success or failure of this system call depends upon the argument values for **key**, **size**, and **shmflg** or system tunable parameters. The system call will attempt to return a new **shmid** if one of the following conditions is true:

- Key is equal to IPC_PRIVATE (0).
- Key does not already have a **shmid** associated with it, and (**shmflg** & IPC_CREAT) is "true" (not zero).

The **key** argument can be set to IPC_PRIVATE in the following ways:

```
shmid = shmget (IPC_PRIVATE, size, shmflg);
```

or

```
shmid = shmget ( 0 , size, shmflg);
```

This alone will cause the system call to be attempted because it satisfies the first condition specified. Exceeding the SHMMNI system tunable parameter always causes a failure. The SHMMNI system tunable parameter determines the maximum number of unique shared memory segments (**shmids**) in the UNIX operating system.

The second condition is satisfied if the value for **key** is not already associated with a **shmid** and the bitwise ANDing of **shmflg** and IPC_CREAT is "true" (not zero). This means that the **key** is unique (not in use) within the UNIX operating system for this facility type and that the IPC_CREAT flag is set (**shmflg** | IPC_CREAT). The bitwise ANDing (&), which is the logical way of testing if a flag is set, is illustrated as follows:

```
shmflg = x 1 x x x   (x = immaterial)
         & IPC_CREAT = 0 1 0 0 0

result = 0 1 0 0 0   (not zero)
```

Because the result is not zero, the flag is set or "true." SHMMNI applies here also, just as for condition one.

IPC_EXCL is another control command used in conjunction with IPC_CREAT to exclusively have the system call fail if, and only if, a **shmid** exists for the specified **key** provided. This is necessary to prevent the process from thinking that it has received a new (unique) **shmid** when it has not. In other words, when both IPC_CREAT and IPC_EXCL are specified, a unique **shmid** is returned if the system call is successful. Any value for **shmflg** returns a new **shmid** if the **key** equals zero (IPC_PRIVATE).

The system call will fail if the value for the **size** argument is less than SHMMIN or greater than SHMMAX. These tunable parameters specify the minimum and maximum shared memory segment **sizes**.

Refer to the **shmget**(2) manual page for specific associated data structure initialization for successful completion. The specific failure conditions with error names are contained there also.

Example Program

The example program in this section (Figure 15-15) is a menu driven program which allows all possible combinations of using the **shmget**(2) system call to be exercised.

From studying this program, you can observe the method of passing arguments and receiving return values. The user-written program requirements are pointed out.

This program begins (lines 4-7) by including the required header files as specified by the **shmget**(2) entry in the *Programmer's Reference Manual*. Note that the **errno.h** header file is included as opposed to declaring **errno** as an external variable; either method will work.

Variable names have been chosen to be as close as possible to those in the synopsis for the system call. Their declarations are self-explanatory. These names make the program more readable, and this is perfectly legal since they are local to the program. The variables declared for this program and their purposes are as follows:

- **key**—used to pass the value for the desired **key**
- **opperm**—used to store the desired operation permissions
- **flags**—used to store the desired control commands (flags)
- **opperm_flags**—used to store the combination from the logical ORing of the **opperm** and **flags** variables; it is then used in the system call to pass the **shmflg** argument
- **shmid**—used for returning the message queue identification number for a successful system call or the error code (−1) for an unsuccessful one
- **size**—used to specify the shared memory segment size.

The program begins by prompting for a hexadecimal **key**, an octal operation permissions code, and finally for the control command combinations (flags) which are selected from a menu (lines 14-31). All possible combinations are allowed even though they might not be viable. This allows observing the errors for illegal combinations.

Next, the menu selection for the flags is combined with the operation permissions, and the result is stored at the address of the **opperm_flags** variable (lines 35-50).

A display then prompts for the **size** of the shared memory segment, and it is stored at the address of the **size** variable (lines 51-54).

The system call is made next, and the result is stored at the address of the **shmid** variable (line 56).

Since the **shmid** variable now contains a valid message queue identifier or the error code (−1), it is tested to see if an error occurred (line 58). If **shmid** equals −1, a message indicates that an error resulted and the external **errno** variable is displayed (lines 60, 61).

If no error occurred, the returned shared memory segment identifier is displayed (line 65).

The example program for the **shmget**(2) system call follows. It is suggested that the source program file be named **shmget.c** and that the executable file be named **shmget**.

When compiling C programs that use floating point operations, the **-f** option should be used on the **cc** command line. If this option is not used, the program will compile successfully, but when the program is executed it will fail.

```
 1    /*This is a program to illustrate
 2    **the shared memory get, shmget(),
 3    **system call capabilities.*/

 4    #include    <sys/types.h>
 5    #include    <sys/ipc.h>
 6    #include    <sys/shm.h>
 7    #include    <errno.h>

 8    /*Start of main C language program*/
 9    main()
10    {
11        key_t key;      /*declare as long integer*/
12        int opperm, flags;
13        int shmid, size, opperm_flags;
14        /*Enter the desired key*/
15        printf("Enter the desired key in hex = ");
16        scanf("%x", &key);

17        /*Enter the desired octal operation
18          permissions.*/
19        printf("\nEnter the operation\n");
20        printf("permissions in octal = ");
21        scanf("%o", &opperm);

22    /*Set the desired flags.*/
23    printf("\nEnter corresponding number to\n");
24    printf("set the desired flags:\n");
25    printf("No flags                  = 0\n");
26    printf("IPC_CREAT                 = 1\n");
27    printf("IPC_EXCL                  = 2\n");
28    printf("IPC_CREAT and IPC_EXCL    = 3\n");
29    printf("              Flags       = ");
30    /*Get the flag(s) to be set.*/
31    scanf("%d", &flags);

32    /*Check the values.*/
33    printf ("\nkey =0x%x, opperm = 0%o, flags = 0%o\n",
34       key, opperm, flags);
```

Figure 15-15: **shmget**(2) System Call Example (Sheet 1 of 2)

```
35    /*Incorporate the control fields (flags) with
36      the operation permissions*/
37    switch (flags)
38    {
39    case 0:  /*No flags are to be set.*/
40       opperm_flags = (opperm | 0);
41       break;
42    case 1:  /*Set the IPC_CREAT flag.*/
43       opperm_flags = (opperm | IPC_CREAT);
44       break;
45    case 2:  /*Set the IPC_EXCL flag.*/
46       opperm_flags = (opperm | IPC_EXCL);
47       break;
48    case 3:  /*Set the IPC_CREAT and IPC_EXCL flags.*/
49       opperm_flags = (opperm | IPC_CREAT | IPC_EXCL);
50    }

51    /*Get the size of the segment in bytes.*/
52    printf ("\nEnter the segment");
53    printf ("\nsize in bytes = ");
54    scanf ("%d", &size);

55    /*Call the shmget system call.*/
56    shmid = shmget (key, size, opperm_flags);

57    /*Perform the following if the call is unsuccessful.*/
58    if(shmid == -1)
59    {
60       printf ("\nThe shmget system call failed!\n");
61       printf ("The error number = %d\n", errno);
62    }
63    /*Return the shmid upon successful completion.*/
64    else
65       printf ("\nThe shmid = %d\n", shmid);
66    exit(0);
67 }
```

Figure 15-15: **shmget**(2) System Call Example (Sheet 2 of 2)

Controlling Shared Memory

This section gives a detailed description of using the **shmctl**(2) system call along with an example program which allows all of its capabilities to be exercised.

Using shmctl

The synopsis found in the **shmctl**(2) entry in the *Programmer's Reference Manual* is as follows:

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

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

The **shmctl**(2) system call requires three arguments to be passed to it, and **shmctl**(2) returns an integer value.

Upon successful completion, a zero value is returned; and when unsuccessful, **shmctl**() returns a −1.

The **shmid** variable must be a valid, non-negative, integer value. In other words, it must have already been created by using the **shmget**(2) system call.

The **cmd** argument can be replaced by one of following control commands (flags):

- IPC_STAT—return the status information contained in the associated data structure for the specified **shmid** and place it in the data structure pointed to by the ***buf** pointer in the user memory area
- IPC_SET—for the specified **shmid**, set the effective user and group identification, and operation permissions
- IPC_RMID—remove the specified **shmid** along with its associated shared memory segment data structure
- SHM_LOCK—lock the specified shared memory segment in memory, must be super-user
- SHM_UNLOCK—unlock the shared memory segment from memory, must be super-user.

A process must have an effective user identification of OWNER/CREATOR or super-user to perform an IPC_SET or IPC_RMID control command. Only the super-user can perform a SHM_LOCK or SHM_UNLOCK control command. A process must have read permission to perform the IPC_STAT control command.

The details of this system call are discussed in the example program for it. If you have problems understanding the logic manipulations in this program, read the "Using **shmget**" section of this chapter; it goes into more detail than what would be practical to do for every system call.

Example Program

The example program in this section (Figure 15-16) is a menu driven program which allows all possible combinations of using the **shmctl**(2) system call to be exercised.

From studying this program, you can observe the method of passing arguments and receiving return values. The user-written program requirements are pointed out.

This program begins (lines 5-9) by including the required header files as specified by the **shmctl**(2) entry in the *Programmer's Reference Manual*. Note in this program that **errno** is declared as an external variable, and therefore, the **errno.h** header file does not have to be included.

Variable and structure names have been chosen to be as close as possible to those in the synopsis for the system call. Their declarations are self-explanatory. These names make the program more readable, and it is perfectly legal since they are local to the program. The variables declared for this program and their purposes are as follows:

- **uid**—used to store the IPC_SET value for the effective user identification
- **gid**—used to store the IPC_SET value for the effective group identification
- **mode**—used to store the IPC_SET value for the operation permissions
- **rtrn**—used to store the return integer value from the system call
- **shmid**—used to store and pass the shared memory segment identifier to the system call
- **command**—used to store the code for the desired control command so that subsequent processing can be performed on it
- **choice**—used to determine which member for the IPC_SET control command that is to be changed
- **shmid_ds**—used to receive the specified shared memory segment identifier's data structure when an IPC_STAT control command is performed
- ***buf**—a pointer passed to the system call which locates the data structure in the user memory area where the IPC_STAT control command is to place its return values or where the IPC_SET command gets the values to set.

Note that the **shmid_ds** data structure in this program (line 16) uses the data structure located in the **shm.h** header file of the same name as a template for its declaration. This is a perfect example of the advantage of local variables.

The next important thing to observe is that although the ***buf** pointer is declared to be a pointer to a data structure of the **shmid_ds** type, it must also be initialized to contain the address of the user memory area data structure (line 17).

Now that all of the required declarations have been explained for this program, this is how it works.

First, the program prompts for a valid shared memory segment identifier which is stored at the address of the **shmid** variable (lines 18-20). This is required for every **shmctl**(2) system call.

Then, the code for the desired control command must be entered (lines 21-29), and it is stored at the address of the command variable. The code is tested to determine the control command for subsequent processing.

If the IPC_STAT control command is selected (code 1), the system call is performed (lines 39, 40) and the status information returned is printed out (lines 41-71). Note that if the system call is unsuccessful (line 146), the status information of the last successful call is printed out. In addition, an error message is displayed and the **errno** variable is printed out (lines 148, 149). If the system call is successful, a message indicates this along with the shared memory segment identifier used (lines 151-154).

If the IPC_SET control command is selected (code 2), the first thing done is to get the current status information for the message queue identifier specified (lines 90-92). This is necessary because this example program provides for changing only one member at a time, and the system call changes all of them. Also, if an invalid value happened to be stored in the user memory area for one of these members, it would cause repetitive failures for this control command until corrected. The next thing the program does is to prompt for a code corresponding to the member to be changed (lines 93-98). This code is stored at the address of the choice variable (line 99). Now, depending upon the member picked, the program prompts for the new value (lines 105-127). The value is placed at the address of the appropriate member in the user memory area data structure, and the system call is made (lines 128-130). Depending upon success or failure, the program returns the same messages as for IPC_STAT above.

If the IPC_RMID control command (code 3) is selected, the system call is performed (lines 132-135), and the **shmid** along with its associated message queue and data structure are removed from the UNIX operating system. Note that the ***buf** pointer is not required as an argument to perform this control command and its value can be zero or NULL. Depending upon the success or failure, the program returns the same messages as for the other control commands.

If the SHM_LOCK control command (code 4) is selected, the system call is performed (lines 137,138). Depending upon the success or failure, the program returns the same messages as for the other control commands.

If the SHM_UNLOCK control command (code 5) is selected, the system call is performed (lines 140-142). Depending upon the success or failure, the program returns the same messages as for the other control commands.

The example program for the **shmctl**(2) system call follows. It is suggested that the source program file be named **shmctl.c** and that the executable file be named **shmctl**.

When compiling C programs that use floating point operations, the **-f** option should be used on the **cc** command line. If this option is not used, the program will compile successfully, but when the program is executed it will fail.

```
1    /*This is a program to illustrate
2    **the shared memory control, shmctl(),
3    **system call capabilities.
4    */

5    /*Include necessary header files.*/
6    #include    <stdio.h>
7    #include    <sys/types.h>
8    #include    <sys/ipc.h>
9    #include    <sys/shm.h>

10   /*Start of main C language program*/
11   main()
12   {
13       extern int errno;
14       int uid, gid, mode;
15       int rtrn, shmid, command, choice;
16       struct shmid_ds shmid_ds, *buf;
17       buf = &shmid_ds;
                . . .
18       /*Get the shmid, and command.*/
19       printf("Enter the shmid = ");
20       scanf("%d", &shmid);
21       printf("\nEnter the number for\n");
22       printf("the desired command:\n");

23       printf("IPC_STAT    =  1\n");
24       printf("IPC_SET     =  2\n");
25       printf("IPC_RMID    =  3\n");
26       printf("SHM_LOCK    =  4\n");
27       printf("SHM_UNLOCK  =  5\n");
28       printf("Entry       =  ");
29       scanf("%d", &command);

30       /*Check the values.*/
31       printf ("\nshmid =%d, command = %d\n",
32           shmid, command);
```

Figure 15-16: **shmctl**(2) System Call Example (Sheet 1 of 4)

```
33          switch (command)
34          {
35          case 1:     /*Use shmctl() to duplicate·
36                the data structure for
37                shmid in the shmid_ds area pointed
38                to by buf and then print it out.*/
39              rtrn = shmctl(shmid, IPC_STAT,
40                  buf);
41              printf ("\nThe USER ID = %d\n",
42                  buf->shm_perm.uid);
43              printf ("The GROUP ID = %d\n",
44                  buf->shm_perm.gid);
45              printf ("The creator's ID = %d\n",
46                  buf->shm_perm.cuid);
47              printf ("The creator's group ID = %d\n",
48                  buf->shm_perm.cgid);
49              printf ("The operation permissions = 0%o\n",
50                  buf->shm_perm.mode);
51              printf ("The slot usage sequence\n");

52              printf ("number = 0%x\n",
53                  buf->shm_perm.seq);
54              printf ("The key= 0%x\n",
55                  buf->shm_perm.key);
56              printf ("The segment size = %d\n",
57                  buf->shm_segsz);
58              printf ("The pid of last shmop = %d\n",
59                  buf->shm_lpid);
60              printf ("The pid of creator = %d\n",
61                  buf->shm_cpid);
62              printf ("The current # attached = %d\n",
63                  buf->shm_nattch);
64              printf("The in memory # attached = %d\n",
65                  buf->shm_cnattach);
66              printf("The last shmat time = %d\n",
67                  buf->shm_atime);
68              printf("The last shmdt time = %d\n",
69                  buf->shm_dtime);
70              printf("The last change time = %d\n",
71                  buf->shm_ctime);
72              break;

                    /* Lines 73 - 87 deleted */
```

Figure 15-16: **shmctl**(2) System Call Example (Sheet 2 of 4)

```
88          case 2:  /*Select and change the desired
89                       member(s) of the data structure.*/

90                  /*Get the original data for this shmid
91                       data structure first.*/
92                  rtrn = shmctl(shmid, IPC_STAT, buf);

93                  printf("\nEnter the number for the\n");
94                  printf("member to be changed:\n");
95                  printf("shm_perm.uid    = 1\n");
96                  printf("shm_perm.gid    = 2\n");
97                  printf("shm_perm.mode   = 3\n");
98                  printf("Entry           = ");
99                  scanf("%d", &choice);
100                  /*Only one choice is allowed per
101                    pass as an illegal entry will
102                    cause repetitive failures until
103                    shmid_ds is updated with
104                        IPC_STAT.*/

105                 switch(choice){
106                 case 1:
107                     printf("\nEnter USER ID = ");
108                     scanf ("%d", &uid);
109                     buf->shm_perm.uid = uid;
110                     printf("\nUSER ID = %d\n",
111                         buf->shm_perm.uid);
112                     break;

113                 case 2:
114                     printf("\nEnter GROUP ID = ");
115                     scanf("%d", &gid);
116                     buf->shm_perm.gid = gid;
117                     printf("\nGROUP ID = %d\n",
118                         buf->shm_perm.gid);
119                     break;

120                 case 3:
121                     printf("\nEnter MODE = ");
122                     scanf("%o", &mode);
123                     buf->shm_perm.mode = mode;
124                     printf("\nMODE = 0%o\n",
125                         buf->shm_perm.mode);
126                     break;
127                 }
128                 /*Do the change.*/
129                 rtrn = shmctl(shmid, IPC_SET,
130                     buf);
131                 break;
```

Figure 15-16: **shmctl**(2) System Call Example (Sheet 3 of 4)

```
132       case 3:  /*Remove the shmid along with its
133                   associated
134                   data structure.*/
135          rtrn = shmctl(shmid, IPC_RMID, NULL);
136          break;

137       case 4: /*Lock the shared memory segment*/
138          rtrn = shmctl(shmid, SHM_LOCK, NULL);
139          break;
140       case 5: /*Unlock the shared memory
141                   segment.*/
142          rtrn = shmctl(shmid, SHM_UNLOCK, NULL);
143          break;
144       }
145       /*Perform the following if the call is unsuccessful.*/
146       if(rtrn == -1)
147       {
148          printf ("\nThe shmctl system call failed!\n");
149          printf ("The error number = %d\n", errno);
150       }
151       /*Return the shmid upon successful completion.*/
152       else
153          printf ("\nShmctl was successful for shmid = %d\n",
154             shmid);
155       exit (0);
156    }
```

Figure 15-16: **shmctl**(2) System Call Example (Sheet 4 of 4)

Operations for Shared Memory

This section gives a detailed description of using the **shmat**(2) and **shmdt**(2) system calls, along with an example program which allows all of their capabilities to be exercised.

Using shmop

The synopsis found in the **shmop**(2) entry in the *Programmer's Reference Manual* is as follows:

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

char *shmat (shmid, shmaddr, shmflg)
int shmid;
char *shmaddr;
int shmflg;

int shmdt (shmaddr)
char *shmaddr;
```

Attaching a Shared Memory Segment

The **shmat**(2) system call requires three arguments to be passed to it, and it returns a character pointer value.

The system call can be cast to return an integer value. Upon successful completion, this value will be the address in core memory where the process is attached to the shared memory segment and when unsuccessful it will be a −1.

The **shmid** argument must be a valid, non-negative, integer value. In other words, it must have already been created by using the **shmget**(2) system call.

The shm**addr** argument can be zero or user supplied when passed to the **shmat**(2) system call. If it is zero, the UNIX operating system picks the address of where the shared memory segment will be attached. If it is user supplied, the address must be a valid address that the UNIX operating system would pick. Letting the operating system pick addresses improves portability.

The **shmflg** argument is used to pass the SHM_RND and SHM_RDONLY flags to the **shmat**() system call.

Further details are discussed in the example program for **shmop**(). If you have problems understanding the logic manipulations in this program, read the "Using **shmget**" section of this chapter; it goes into more detail than what would be practical to do for every system call.

Detaching Shared Memory Segments

The **shmdt**(2) system call requires one argument to be passed to it, and **shmdt**(2) returns an integer value.

Upon successful completion, zero is returned; and when unsuccessful, **shmdt**(2) returns a −1.

Further details of this system call are discussed in the example program. If you have problems understanding the logic manipulations in this program, read the "Using **shmget**" section of this chapter; it goes into more detail than what would be practical to do for every system call.

Example Program

The example program in this section (Figure 15-17) is a menu driven program which allows all possible combinations of using the **shmat**(2) and **shmdt**(2) system calls to be exercised.

From studying this program, you can observe the method of passing arguments and receiving return values. The user-written program requirements are pointed out.

This program begins (lines 5-9) by including the required header files as specified by the **shmop**(2) entry in the *Programmer's Reference Manual*. Note that in this program that **errno** is declared as an external variable, and therefore, the **errno.h** header file does not have to be included.

Variable and structure names have been chosen to be as close as possible to those in the synopsis. Their declarations are self-explanatory. These names make the program more readable, and this is perfectly legal since they are local to the program. The variables declared for this program and their purposes are as follows:

- **flags**—used to store the codes of SHM_RND or SHM_RDONLY for the **shmat**(2) system call
- **addr**—used to store the address of the shared memory segment for the **shmat**(2) and **shmdt**(2) system calls
- **i**—used as a loop counter for attaching and detaching
- **attach**—used to store the desired number of attach operations
- **shmid**—used to store and pass the desired shared memory segment identifier
- **shmflg**—used to pass the value of flags to the **shmat**(2) system call
- **retrn**—used to store the return values from both system calls
- **detach**—used to store the desired number of detach operations

This example program combines both the **shmat**(2) and **shmdt**(2) system calls. The program prompts for the number of attachments and enters a loop until they are done for the specified shared memory identifiers. Then, the program prompts for the number of detachments to be performed and enters a loop until they are done for the specified shared memory segment addresses.

shmat

The program prompts for the number of attachments to be performed, and the value is stored at the address of the attach variable (lines 17-21).

A loop is entered using the attach variable and the i counter (lines 23-70) to perform the specified number of attachments.

In this loop, the program prompts for a shared memory segment identifier (lines 24-27) and it is stored at the address of the **shmid** variable (line 28). Next, the program prompts for the address where the segment is to be attached (lines 30-34), and it is stored at the address of the **addr** variable (line 35). Then, the program prompts for the desired flags to be used for the attachment (lines 37-44), and the code representing the flags is stored at the address of the flags variable (line 45). The flags variable is tested to determine the code to be stored for the **shmflg** variable used to pass them to the **shmat**(2) system call (lines 46-57). The system call is made (line 60). If successful, a message stating so is displayed along with the attach address (lines 66-68). If unsuccessful, a message stating so is displayed and the error code is displayed (lines 62, 63). The loop then continues until it finishes.

shmdt

After the attach loop completes, the program prompts for the number of detach operations to be performed (lines 71-75), and the value is stored at the address of the detach variable (line 76).

A loop is entered using the detach variable and the i counter (lines 78-95) to perform the specified number of detachments.

In this loop, the program prompts for the address of the shared memory segment to be detached (lines 79-83), and it is stored at the address of the **addr** variable (line 84). Then, the **shmdt**(2) system call is performed (line 87). If successful, a message stating so is displayed along with the address that the segment was detached from (lines 92,93). If unsuccessful, the error number is displayed (line 89). The loop continues until it finishes.

The example program for the **shmop**(2) system calls follows. It is suggested that the program be put into a source file called **shmop.c** and then into an executable file called **shmop**.

When compiling C programs that use floating point operations, the **-f** option should be used on the **cc** command line. If this option is not used, the program will compile successfully, but when the program is executed it will fail.

```
1    /*This is a program to illustrate
2    **the shared memory operations, shmop(),
3    **system call capabilities.
4    */

5    /*Include necessary header files.*/
6    #include    <stdio.h>
7    #include    <sys/types.h>
8    #include    <sys/ipc.h>
9    #include    <sys/shm.h>
10   /*Start of main C language program*/
11   main()
12   {
13       extern int errno;
14       int flags, addr, i, attach;
15       int shmid, shmflg, retrn, detach;

16       /*Loop for attachments by this process.*/
17       printf("Enter the number of\n");
18       printf("attachments for this\n");
19       printf("process (1-4).\n");
20       printf("       Attachments = ");

21       scanf("%d", &attach);
22       printf("Number of attaches = %d\n", attach);

23       for(i = 1; i <= attach; i++) {
24           /*Enter the shared memory ID.*/
25           printf("\nEnter the shmid of\n");
26           printf("the shared memory segment to\n");
27           printf("be operated on = ");
28           scanf("%d", &shmid);
29           printf("\nshmid = %d\n", shmid);

30           /*Enter the value for shmaddr.*/
31           printf("\nEnter the value for\n");
32           printf("the shared memory address\n");
33           printf("in hexadecimal:\n");
34           printf("            Shmaddr = ");
35           scanf("%x", &addr);
36           printf("The desired address = 0x%x\n", addr);
```

Figure 15-17: **shmop**() System Call Example (Sheet 1 of 3)

```
37          /*Specify the desired flags.*/
38          printf("\nEnter the corresponding\n");
39          printf("number for the desired\n");
40          printf("flags:\n");
41          printf("SHM_RND                 = 1\n");
42          printf("SHM_RDONLY              = 2\n");
43          printf("SHM_RND and SHM_RDONLY = 3\n");
44          printf("            Flags       = ");
45          scanf("%d", &flags);

46          switch(flags)
47          {
48          case 1:
49              shmflg = SHM_RND;
50              break;
51          case 2:
52              shmflg = SHM_RDONLY;
53              break;
54          case 3:
55              shmflg = SHM_RND | SHM_RDONLY;
56              break;
57          }
58          printf("\nFlags = 0%o\n", shmflg);

59          /*Do the shmat system call.*/
60          retrn = (int)shmat(shmid, addr, shmflg);
61          if(retrn == -1)  {
62              printf("\nShmat failed.  ");
63              printf("Error = %d\n", errno);
64          }
65          else {
66              printf ("\nShmat was successful\n");
67              printf("for shmid = %d\n", shmid);
68              printf("The address = 0x%x\n", retrn);
69          }
70      }

71      /*Loop for detachments by this process.*/
72      printf("Enter the number of\n");
73      printf("detachments for this\n");
74      printf("process (1-4).\n");
75      printf("       Detachments = ");
```

Figure 15-17: **shmop**() System Call Example (Sheet 2 of 3)

```
76          scanf("%d", &detach);
77          printf("Number of attaches = %d\n", detach);
78          for(i = 1; i <= detach; i++) {

79              /*Enter the value for shmaddr.*/
80              printf("\nEnter the value for\n");
81              printf("the shared memory address\n");
82              printf("in hexadecimal:\n");
83              printf("            Shmaddr = ");
84              scanf("%x", &addr);
85              printf("The desired address = 0x%x\n", addr);

86              /*Do the shmdt system call.*/
87              retrn = (int)shmdt(addr);
88              if(retrn == -1)  {
89                  printf("Error = %d\n", errno);
90              }
91              else {
92                  printf ("\nShmdt was successful\n");
93                  printf("for address  = 0%x\n", addr);

94              }
95          }
96      }
```

Figure 15-17: **shmop**() System Call Example (Sheet 3 of 3)

Chapter 16: Interprocess Communication Tutorial

Abstract

This document was created by Stuart Sechrest at the following site:

Computer Science Research Group
Computer Science Division
Department of Electrical Engineering and Computer Science
University of California, Berkeley

UMIPS offers several choices for interprocess communication. To aid the programmer in developing programs which are comprised of cooperating processes, the different choices are discussed and a series of example programs are presented. These programs demonstrate in a simple way the use of pipes, socketpairs, sockets and the use of datagram and stream communication. The intent of this document is to present a few simple example programs, not to describe the networking system in full.

Goals

Facilities for interprocess communication (IPC) and networking were originally a major addition to UNIX in the Berkeley UNIX 4.2BSD release. These facilities required major additions and some changes to the system interface. The basic idea of this interface is to make IPC similar to file I/O. In UNIX a process has a set of I/O descriptors, from which one reads and to which one writes. Descriptors may refer to normal files, to devices (including terminals), or to communication channels. The use of a descriptor has three phases: its creation, its use for reading and writing, and its destruction. By using descriptors to write files, rather than simply naming the target file in the write call, one gains a surprising amount of flexibility. Often, the program that creates a descriptor will be different from the program that uses the descriptor. For example the shell can create a descriptor for the output of the 'ls' command that will cause the listing to appear in a file rather than on a terminal. Pipes are another form of descriptor that have been used in UNIX for some time. Pipes allow one-way data transmission from one process to another; the two processes and the pipe must be set up by a common ancestor.

The use of descriptors is not the only communication interface provided by UNIX. The signal mechanism sends a tiny amount of information from one process to another. The signaled process receives only the signal type, not the identity of the sender, and the number of possible signals is small. The signal semantics limit the flexibility of the signaling mechanism as a means of interprocess communication.

The identification of IPC with I/O is quite longstanding in UNIX and has proved quite successful. At first, however, IPC was limited to processes communicating within a single machine. With Berkeley UNIX 4.2BSD this expanded to include IPC between machines. This expansion has necessitated some change in the way that descriptors are created. Additionally, new possibilities for the meaning of read and write have been admitted. Originally the meanings, or semantics, of these terms were fairly simple. When you wrote something it was delivered. When you read something, you were blocked until the data arrived. Other possibilities exist, however. One can write without full assurance of delivery if one can check later to catch occasional failures. Messages can be kept as discrete units or merged into a stream. One can ask to read, but insist on not waiting if nothing is immediately available. These new possibilities are allowed in the UMIPS IPC interface.

Thus UMIPS offers several choices for IPC. This chapter presents simple examples that illustrate some of the choices. The reader is presumed to be familiar with the C programming language [Kernighan & Ritchie 1978], but not necessarily with the system calls of the UNIX system or with processes and interprocess communication. The paper reviews the notion of a process and the types of communication that are supported by UMIPS. A series of examples are presented that create processes that communicate with one another. The programs show different ways of establishing channels of communication. Finally, the calls that actually transfer data are reviewed. To clearly present how communication can take place, the example programs have been cleared of anything that might be construed as useful work. They can, therefore, serve as models for the programmer trying to construct programs which are comprised of cooperating processes.

Processes

A *program* is both a sequence of statements and a rough way of referring to the computation that occurs when the compiled statements are run. A *process* can be thought of as a single line of control in a program. Most programs execute some statements, go through a few loops, branch in various directions and then end. These are single process programs. Programs can also have a point where control splits into two independent lines, an action called *forking*. In UNIX these lines can never join again. A call to the system routine *fork()* causes a process to split in this way. The result of this call is that two independent processes will be running, executing exactly the same code. Memory values will be the same for all values set before the fork, but, subsequently, each version will be able to change only the value of its own copy of each variable. Initially, the only difference between the two will be the value returned by *fork()*. The parent will receive a process id for the child, the child will receive a zero. Calls to *fork(),* therefore, typically precede, or are included in, an if-statement.

A process views the rest of the system through a private table of descriptors. The descriptors can represent open files or sockets (sockets are communication objects that will be discussed below). Descriptors are referred to by their index numbers in the table. The first three descriptors are often known by special names, *stdin, stdout* and *stderr*. These are the standard input, output and error. When a process forks, its descriptor table is copied to the child. Thus, if the parent's standard input is being taken from a terminal (devices are also treated as files in UNIX), the child's input will be taken from the same terminal. Whoever reads first will get the input. If, before forking, the parent changes its standard input so that it is reading from a new file, the child will take its input from the new file. It is also possible to take input from a socket, rather than from a file.

Pipes

Most users of UNIX know that they can pipe the output of a program "prog1" to the input of another, "prog2," by typing the command

```
prog1 | prog2
```

This is called "piping" the output of one program to another because the mechanism used to transfer the output is called a pipe. When the user types a command, the command is read by the shell, which decides how to execute it. If the command is simple, for example, *"prog1,"* the shell forks a process, which executes the program, prog1, and then dies. The shell waits for this termination and then prompts for the next command. If the command is a compound command,

```
prog1 | prog2
```

the shell creates two processes connected by a pipe. One process runs the program, prog1, the other runs prog2. The pipe is an I/O mechanism with two ends, or sockets. Data that is written into one socket can be read from the other.

```
#include <stdio.h>
#define DATA "Bright star, would I were steadfast
#     as thou art ..."
/*
 * This program creates a pipe, then forks.  The child
 * communicates to the parent over the pipe.  Notice
 * that a pipe is a one-way communications device.  I
 * can write to the output socket (sockets[1], the
 * second socket of the array returned by pipe()) and
 * read from the input socket (sockets[0]), but not vice
 * versa.
 */
main()
{
      int sockets[2], child;
      /* Create a pipe */
      if (pipe(sockets) < 0) {
            perror("opening stream socket pair");
            exit(10);
      }

      if ((child = fork()) == -1)
            perror("fork");
      else if (child) {
            char buf[1024];

            /* This is still the parent. */
            /* It reads the child's message. */
            close(sockets[1]);
            if (read(sockets[0], buf, 1024) < 0)
                  perror("reading message");
            printf("-->%s\n", buf);
            close(sockets[0]);
      } else {
            /* This is the child. */
            /* It writes a message to its parent. */
            close(sockets[0]);
            if (write(sockets[1], DATA, sizeof(DATA)) < 0)
                  perror("writing message");
            close(sockets[1]);
      }
}
```

Figure 16-1: Use of a pipe

Since a program specifies its input and output only by the descriptor table indices, which appear as variables or constants, the input source and output destination can be changed without changing the text of the program. It is in this way that the shell is able to set up pipes. Before executing prog1, the process can close whatever is at *stdout* and replace it with one end of a pipe. Similarly, the process that will execute prog2 can substitute the opposite end of the pipe for *stdin*.

Let us now examine a program that creates a pipe for communication between its child and itself (Figure 16-1). A pipe is created by a parent process, which then forks. When a process forks, the parent's descriptor table is copied into the child's.

In Figure 16-1, the parent process makes a call to the system routine *pipe()*. This routine creates a pipe and places descriptors for the sockets for the two ends of the pipe in the process's descriptor table. *Pipe()* is passed an array into which it places the index numbers of the sockets it created. The two ends are not equivalent. The socket whose index is returned in the low word of the array is opened for reading only, while the socket in the high end is opened only for writing. This corresponds to the fact that the standard input is the first descriptor of a process's descriptor table and the standard output is the second. After creating the pipe, the parent creates the child with which it will share the pipe by calling *fork()*. Figure 16-2 illustrates the effect of a fork. The parent process's descriptor table points to both ends of the pipe. After the fork, both parent's and child's descriptor tables point to the pipe. The child can then use the pipe to send a message to the parent.

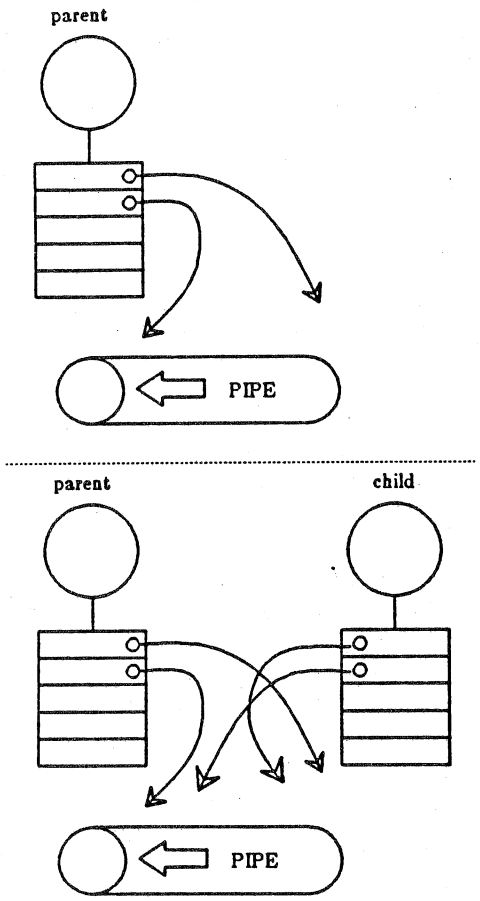


Figure 16-2: Sharing a pipe between parent and child

Just what is a pipe? It is a one-way communication mechanism, with one end opened for reading and the other end for writing. Therefore, parent and child need to agree on which way to turn the pipe, from parent to child or the other way around. Using the same pipe for communication both from parent to child and from child to

parent would be possible (since both processes have references to both ends), but very complicated. If the parent and child are to have a two-way conversation, the parent creates two pipes, one for use in each direction. (In accordance with their plans, both parent and child in the example above close the socket that they will not use. It is not required that unused descriptors be closed, but it is good practice.) A pipe is also a *stream* communication mechanism; that is, all messages sent through the pipe are placed in order and reliably delivered. When the reader asks for a certain number of bytes from this stream, he is given as many bytes as are available, up to the amount of the request. Note that these bytes may have come from the same call to *write()* or from several calls to *write()* which were concatenated.

Socketpairs

UMIPS provides a slight generalization of pipes. A pipe is a pair of connected sockets for one-way stream communication. One may obtain a pair of connected sockets for two-way stream communication by calling the routine *socketpair()*. The program in Figure 16-3 calls *socketpair()* to create such a connection. The program uses the link for communication in both directions. Since socketpairs are an extension of pipes, their use resembles that of pipes. Figure 16-4 illustrates the result of a fork following a call to *socketpair()*.

Socketpair() takes as arguments a specification of a domain, a style of communication, and a protocol. These are the parameters shown in the example. Domains and protocols will be discussed in the next section. Briefly, a domain is a space of names that may be bound to sockets and implies certain other conventions. Currently, socketpairs have only been implemented for one domain, called the UNIX domain. The UNIX domain uses UNIX path names for naming sockets. It only allows communication between sockets on the same machine.

Note that the header files *<sys/socket.h>* and *<sys/types.h>*. are required in this program. The constants AF_UNIX and SOCK_STREAM are defined in *<sys/socket.h>*, which in turn requires the file *<sys/types.h>* for some of its definitions.

UMIPS 3.0 does not support the AF_UNIX domain. Use AF-INET instead.

```
#include <sys/types.h>
#include <sys/socket.h>
#include <stdio.h>
#define DATA1 "In Xanadu, did Kublai Khan . . ."
#define DATA2 "A stately pleasure dome decree . . ."
/*
 * This program creates a pair of connected sockets then
 * forks and communicates over them.  This is very
 * similar to communication with pipes, however,
 * socketpairs are two-way communications objects.
 * Therefore I can send messages in both directions.
 */
main()
{
      int sockets[2], child;
      char buf[1024];
      if (socketpair(AF_UNIX, SOCK_STREAM, 0, sockets) < 0) {
            perror("opening stream socket pair");
            exit(1);
      }
      if ((child = fork()) == -1)
            perror("fork");
      else if (child) { /* This is the parent. */
            close(sockets[0]);
            if (read(sockets[1], buf, 1024, 0) < 0)
                  perror("reading stream message");
            printf("-->%s\n", buf);
            if (write(sockets[1], DATA2, sizeof(DATA2)) < 0)
```

```
                perror("writing stream message");
            close(sockets[1]);
        } else {          /* This is the child. */
            close(sockets[1]);
            if (write(sockets[0], DATA1, sizeof(DATA1)) < 0)
                perror("writing stream message");
            if (read(sockets[0], buf, 1024, 0) < 0)
                perror("reading stream message");
            printf("-->%s\n", buf);
            close(sockets[0]);
        }
}
```

Figure 16-3: Use of a socketpair

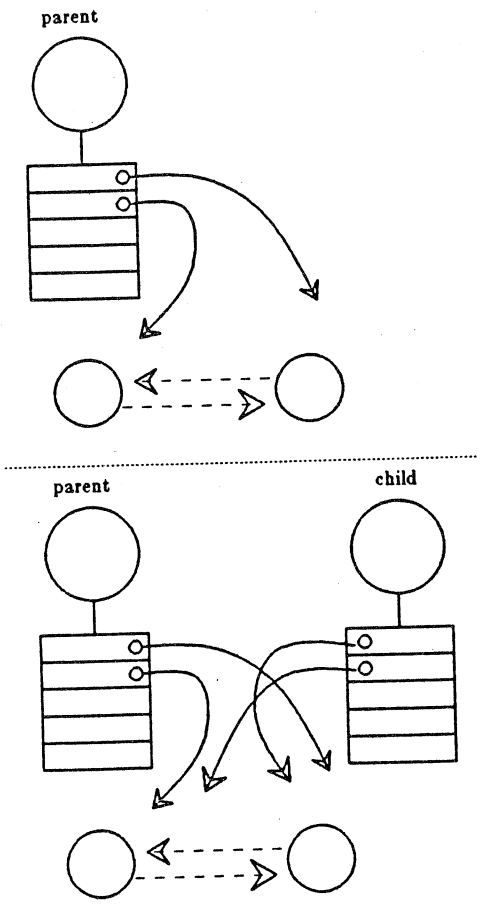


Figure 16-4: Sharing a socketpair between parent and child

Domains and Protocols

Pipes and socketpairs are a simple solution for communicating between a parent and child or between child processes. What if we wanted to have processes that have no common ancestor with whom to set up communication? Neither standard UNIX pipes nor socketpairs are the answer here, since both mechanisms require a common ancestor to set up the communication. We would like to have two processes separately create sockets and then have messages sent between them. This is often the case when providing or using a service in the system. This is also the case when the communicating processes are on separate machines. In UMIPS one can create individual sockets, give them names and send messages between them.

Sockets created by different programs use names to refer to one another; names generally must be translated into addresses for use. The space from which an address is drawn is referred to as a *domain*. There are several domains for sockets. Two that will be used in the examples here are the UNIX domain (or AF_UNIX, for Address Format UNIX) and the Internet domain (or AF_INET). UNIX domain IPC is an experimental facility in UMIPS. In the UNIX domain, a socket is given a path name within the file system name space. A file system node is created for the socket and other processes may then refer to the socket by giving the proper pathname. UNIX domain names, therefore, allow communication between any two processes that work in the same file system. The Internet domain is the UNIX implementation of the DARPA Internet standard protocols IP/TCP/UDP. Addresses in the Internet domain consist of a machine network address and an identifying number, called a port. Internet domain names allow communication between machines.

Communication follows some particular "style." Currently, communication is either through a *stream* or by *datagram*. Stream communication implies several things. Communication takes place across a connection between two sockets. The communication is reliable, error-free, and, as in pipes, no message boundaries are kept. Reading from a stream may result in reading the data sent from one or several calls to *write()* or only part of the data from a single call, if there is not enough room for the entire message, or if not all the data from a large message has been transferred. The protocol implementing such a style will retransmit messages received with errors. It will also return error messages if one tries to send a message after the connection has been broken. Datagram communication does not use connections. Each message is addressed individually. If the address is correct, it will generally be received, although this is not guaranteed. Often datagrams are used for requests that require a response from the recipient. If no response arrives in a reasonable amount of time, the request is repeated. The individual datagrams will be kept separate when they are read, that is, message boundaries are preserved.

The difference in performance between the two styles of communication is generally less important than the difference in semantics. The performance gain that one might find in using datagrams must be weighed against the increased complexity of the program, which must now concern itself with lost or out of order messages. If lost messages may simply be ignored, the quantity of traffic may be a consideration. The expense of setting up a connection is best justified by frequent use of the connection. Since the performance of a protocol changes as it is tuned for different situations, it is best to seek the most up-to-date information when making choices for a program in which performance is crucial.

A protocol is a set of rules, data formats and conventions that regulate the transfer of data between participants in the communication. In general, there is one protocol for each socket type (stream, datagram, etc.) within each domain. The code that implements a protocol keeps track of the names that are bound to sockets, sets up connections and transfers data between sockets, perhaps sending the data across a network. This code also keeps track of the names that are bound to sockets. It is possible for several protocols, differing only in low level details, to implement the same style of communication within a particular domain. Although it is possible to select which protocol should be used, for nearly all uses it is sufficient to request the default protocol. This has been done in all of the example programs.

One specifies the domain, style and protocol of a socket when it is created. For example, in Figure 16-5 the call to *socket()* causes the creation of a datagram socket with the default protocol in the UNIX domain.

Datagrams in the UNIX Domain

Let us now look at two programs that create sockets separately. The programs in Figures 16-5 and 16-6 use datagram communication rather than a stream. The structure used to name UNIX domain sockets is defined in the file *<sys/un.h>*. The definition has also been included in the example for clarity.

Each program creates a socket with a call to *socket()*. These sockets are in the UNIX domain. Once a name has been decided upon it is attached to a socket by the system call *bind()*. The program in Figure 16-5 uses the name "socket", which it binds to its socket. This name will appear in the working directory of the program. The routines in Figure 16-6 use its socket only for sending messages. It does not create a name for the socket because no other process has to refer to it.

Names in the UNIX domain are path names. Like file path names they may be either absolute (e.g. "/dev/imaginary") or relative (e.g. "socket"). Because these names are used to allow processes to rendezvous, relative path names can pose difficulties and should be used with care. When a name is bound into the name space, a file (inode) is allocated in the file system. If the inode is not deallocated, the name will continue to exist even after the bound socket is closed. This can cause subsequent runs of a program to find that a name is unavailable, and can cause directories to fill up with these objects. The names are removed by calling *unlink()* or using the **rm**(1) command. Names in the UNIX domain are only used for rendezvous. They are not used for message delivery once a connection is established. Therefore, in contrast with the Internet domain, unbound sockets need not be (and are not) automatically given addresses when they are connected.

There is no established means of communicating names to interested parties. In the example, the program in Figure 16-6 gets the name of the socket to which it will send its message through its command line arguments. Once a line of communication has been created, one can send the names of additional, perhaps new, sockets over the link. Facilities will have to be built that will make the distribution of names less of a problem than it now is.

UMIPS 3.0 does not support the AF_UNIX domain. Use AF-INET instead.

```
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>
/*
 * In the included file <sys/un.h>
 * a sockaddr_un is defined as follows
 * struct sockaddr_un {
 *     short sun_family;
 *     char  sun_path[108];
 * };
 */
#include <stdio.h>

#define NAME "socket"

/*
 * This program creates a UNIX domain datagram socket,
 * binds a name to it, then reads from the socket.
```

```
 */
main()
{
      int sock, length;
      struct sockaddr_un name;
      char buf[1024];

      /* Create socket from which to read. */
      sock = socket(AF_UNIX, SOCK_DGRAM, 0);
      if (sock < 0) {
            perror("opening datagram socket");
            exit(1);
      }
      /* Create name. */
      name.sun_family = AF_UNIX;
      strcpy(name.sun_path, NAME);
      if (bind(sock, &name, sizeof(struct sockaddr_un))) {
            perror("binding name to datagram socket");
            exit(1);
      }
      printf("socket -->%s\n", NAME);
      /* Read from the socket */
      if (read(sock, buf, 1024) < 0)
            perror("receiving datagram packet");
      printf("-->%s\n", buf);
      close(sock);
      unlink(NAME);
}
```

Figure 16-5: Reading UNIX domain datagrams

```
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <stdio.h>

#define DATA "The sea is calm tonight, the tide is full..."

/*
 * Here I send a datagram to a receiver whose name I
 * get from the command line arguments.  The form of the
 * command line is udgramsend pathname
 */

main(argc, argv)
      int argc;
      char *argv[];
{
      int sock;
      struct sockaddr_un name;

      /* Create socket on which to send. */
      sock = socket(AF_UNIX, SOCK_DGRAM, 0);
      if (sock < 0) {
            perror("opening datagram socket");
            exit(1);
      }
      /* Construct name of socket to send to. */
      name.sun_family = AF_UNIX;
      strcpy(name.sun_path, argv[1]);
      /* Send message. */
      if (sendto(sock, DATA, sizeof(DATA), 0,
          &name, sizeof(struct sockaddr_un)) < 0) {
            perror("sending datagram message");
      }
      close(sock);
}
```

Figure 16-6: Sending UNIX domain datagrams

Datagrams in the Internet Domain

The examples in Figures 16-7 and 16-8 are very close to the previous example except that the socket is in the Internet domain. The structure of Internet domain addresses is defined in the file *<netinet/in.h>*. Internet addresses specify a host address (a 32-bit number) and a delivery slot, or port, on that machine. These ports are managed by the system routines that implement a particular protocol. Unlike UNIX domain names, Internet socket names are not entered into the file system and, therefore, they do not have to be unlinked after the socket has been closed. When a message must be sent between machines it is sent to the protocol routine on the destination machine, which interprets the address to determine to which socket the message should be delivered. Several different protocols may be active on the same machine, but, in general, they will not communicate with one another. As a result, different protocols are allowed to use the same port numbers. Thus, implicitly, an Internet address is a triple including a protocol as well as the port and machine address. An *association* is a temporary or permanent specification of a pair of communicating sockets. An association is thus identified by the set of definitions *<protocol, local machine address, local port, remote machine address, remote port>*. An association may be transient when using datagram sockets; the association actually exists during a *send* operation. The protocol for a socket is chosen when the socket is created. The local machine address for a socket can be any valid network address of the machine, if it has more than one, or it can be the wildcard value INADDR_ANY. The wildcard value is used in the program in Figure 16-7. If a machine has several network addresses, it is likely that messages sent to any of the addresses should be deliverable to a socket. This will be the case if the wildcard value has been chosen. Note that even if the wildcard value is chosen, a program sending messages to the named socket must specify a valid network address. One can be willing to receive from "anywhere," but one cannot send a message "anywhere." The program in Figure 16-8 is given the destination host name as a command line argument. To determine a network address to which it can send the message, it looks up the host address by the call to *gethostbyname()*. The returned structure includes the host's network address, which is copied into the structure specifying the destination of the message.

The port number can be thought of as the number of a mailbox, into which the protocol places one's messages. Certain daemons, offering certain advertised services, have reserved or "well-known" port numbers. These fall in the range from 1 to 1023. Higher numbers are available to general users. Only servers need to ask for a particular number. The system will assign an unused port number when an address is bound to a socket. This may happen when an explicit *bind* call is made with a port number of 0, or when a *connect* or *send* is performed on an unbound socket. Note that port numbers are not automatically reported back to the user. After calling *bind()*, asking for port 0, one may call *getsockname()* to discover what port was actually assigned. The routine *getsockname()* will not work for names in the UNIX domain.

The format of the socket address is specified in part by standards within the Internet domain. The specification includes the order of the bytes in the address. Because machines differ in the internal representation they ordinarily use to represent integers, printing out the port number as returned by *getsockname()* may result in a misinterpretation. To print out the number, it is necessary to use the routine *ntohs()* (for *network to host: short*) to convert the number from the network representation to the host's representation. On some machines, such as 68000-based machines, this is a null operation. On others, such as VAXes, this results in a swapping of bytes. Another routine exists to convert a short integer from the host format to the network

format, called *htons()*; similar routines exist for long integers. For further information, refer to the entry for *byteorder* in section 3 of the *Programmer's Reference Manual*.

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <stdio.h>

/*
 * In the included file <netinet/in.h> a sockaddr_in
 * is defined as follows:
 * struct sockaddr_in {
 *    short sin_family;
 *    u_short     sin_port;
 *    struct in_addr sin_addr;
 *    char  sin_zero[8];
 * };
 *
 * This program creates a datagram socket, binds a name to
 * it, then reads from the socket.
 */
main()
{
      int sock, length;
      struct sockaddr_in name;
      char buf[1024];

      /* Create socket from which to read. */
      sock = socket(AF_INET, SOCK_DGRAM, 0);
      if (sock < 0) {
            perror("opening datagram socket");
            exit(1);
      }
      /* Create name with wildcards. */
      name.sin_family = AF_INET;
      name.sin_addr.s_addr = INADDR_ANY;
      name.sin_port = 0;
      if (bind(sock, &name, sizeof(name))) {
            perror("binding datagram socket");
            exit(1);
      }
      /* Find assigned port value and print it out. */
      length = sizeof(name);
      if (getsockname(sock, &name, &length)) {
            perror("getting socket name");
            exit(1);
```

```
        }
        printf("Socket has port #%d\n", ntohs(name.sin_port));
        /* Read from the socket */
        if (read(sock, buf, 1024) < 0)
                perror("receiving datagram packet");
        printf("-->%s\n", buf);
        close(sock);
}
```

Figure 16-7: Reading Internet domain datagrams

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>

#define DATA "The sea is calm tonight, the tide is full..."

/*
 * Here I send a datagram to a receiver whose name I get
 * from the command line arguments.  The form of the
 * command line is dgramsend hostname portnumber
 */

main(argc, argv)
        int argc;
        char *argv[];
{
        int sock;
        struct sockaddr_in name;
        struct hostent *hp, *gethostbyname();

        /* Create socket on which to send. */
        sock = socket(AF_INET, SOCK_DGRAM, 0);
        if (sock < 0) {
                perror("opening datagram socket");
                exit(1);
        }
        /*
         * Construct name, with no wildcards, of the
         * socket to send to.  Getnostbyname() returns a
         * structure including the network address of
         * the specified host.  The port number is
         * taken from the command line.
         */
        hp = gethostbyname(argv[1]);
        if (hp == 0) {
                fprintf(stderr, "%s: unknown host0, argv[1]);
                exit(2);
        }
        bcopy(hp->h_addr, &name.sin_addr, hp->h_length);
```

```
        name.sin_family = AF_INET;
        name.sin_port = htons(atoi(argv[2]));
        /* Send message. */
        if (sendto(sock, DATA, sizeof(DATA), 0, &name, \
                           sizeof(name)) < 0)
              perror("sending datagram message");
        close(sock);
}
```

Figure 16-8: Sending an Internet domain datagram

Connections

To send data between stream sockets (having communication style SOCK_STREAM), the sockets must be connected. Figures 16-9 and 16-10 show two programs that create such a connection. The program in 16-9 is relatively simple. To initiate a connection, this program simply creates a stream socket, then calls *connect()*, specifying the address of the socket to which it wishes its socket connected. Provided that the target socket exists and is prepared to handle a connection, connection will be complete, and the program can begin to send messages. Messages will be delivered in order without message boundaries, as with pipes. The connection is destroyed when either socket is closed (or soon thereafter). If a process persists in sending messages after the connection is closed, a SIGPIPE signal is sent to the process by the operating system. Unless explicit action is taken to handle the signal (see the manual page for *signal* or *sigvec*), the process will terminate and the shell will print the message "broken pipe."

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>

#define DATA "Half a league, half a league . . ."

/*
* This program creates a socket and initiates a
* connection with the socket given in the command
* line.  One message is sent over the connection and
* then the socket is closed, ending the connection. The
* form of the command line is streamwrite hostname
* portnumber
*/

main(argc, argv)
      int argc;
      char *argv[];
{
      int sock;
      struct sockaddr_in server;
      struct hostent *hp, *gethostbyname();
      char buf[1024];

      /* Create socket */
      sock = socket(AF_INET, SOCK_STREAM, 0);
      if (sock < 0) {
            perror("opening stream socket");
            exit(1);
      }
      /* Connect socket using name specified by command line. */
      server.sin_family = AF_INET;
      hp = gethostbyname(argv[1]);
      if (hp == 0) {
            fprintf(stderr, "%s: unknown host0, argv[1]);
            exit(2);
```

```
        }
        bcopy(hp->h_addr, &server.sin_addr, hp->h_length);
        server.sin_port = htons(atoi(argv[2]));

        if (connect(sock, &server, sizeof(server)) < 0) {
                perror("connecting stream socket");
                exit(1);
        }
        if (write(sock, DATA, sizeof(DATA)) < 0)
                perror("writing on stream socket");
        close(sock);
}
```

Figure 16-9: Initiating an Internet domain stream connection

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>
#define TRUE 1

/*
 * This program creates a socket and then begins an
 * infinite loop. Each time through the loop it accepts
 * a connection and prints out messages from it.  When
 * the connection breaks, or a termination message comes
 * through, the program accepts a new connection.
*/

main()
{
      int sock, length;
      struct sockaddr_in server;
      int msgsock;
      char buf[1024];
      int rval;
      int i;

      /* Create socket */
      sock = socket(AF_INET, SOCK_STREAM, 0);
      if (sock < 0) {
            perror("opening stream socket");
            exit(1);
      }
      /* Name socket using wildcards */
      server.sin_family = AF_INET;
      server.sin_addr.s_addr = INADDR_ANY;
      server.sin_port = 0;
      if (bind(sock, &server, sizeof(server))) {
            perror("binding stream socket");
            exit(1);
      }
      /* Find out assigned port number and print it out */
      length = sizeof(server);
      if (getsockname(sock, &server, &length)) {
            perror("getting socket name");
            exit(1);
      }
      printf("Socket has port #%d\n", ntohs(server.sin_port));

      /* Start accepting connections */
      listen(sock, 5);
      do {
            msgsock = accept(sock, 0, 0);
            if (msgsock == -1)
                  perror("accept");
            else do {
                  bzero(buf, sizeof(buf));
```

```
                if ((rval = read(msgsock, buf, 1024)) < 0)
                        perror("reading stream message");
                i = 0;
                if (rval == 0)
                        printf("Ending connection\n");
                else
                        printf("-->%s\n", buf);
            } while (rval != 0);
            close(msgsock);
        } while (TRUE);
        /*
         * Since this program has an infinite loop, the socket
         * "sock" is never explicitly closed.  However, all
         * sockets will be closed automatically when a process
         * is killed or terminates normally.
         */
}
```

Figure 16-10: Accepting an Internet domain stream connection

```
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/time.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>
#define TRUE 1

/*
 * This program uses select() to check that someone
 * is trying to connect before calling accept().
 */

main()
{
      int sock, length;
      struct sockaddr_in server;
      int msgsock;
      char buf[1024];
      int rval;
      fd_set ready;
      struct timeval to;

      /* Create socket */
      sock = socket(AF_INET, SOCK_STREAM, 0);
      if (sock < 0) {
            perror("opening stream socket");
            exit(1);
      }
      /* Name socket using wildcards */
      server.sin_family = AF_INET;
      server.sin_addr.s_addr = INADDR_ANY;
      server.sin_port = 0;
      if (bind(sock, &server, sizeof(server))) {
            perror("binding stream socket");
            exit(1);
      }
      /* Find out assigned port number and print it out */
      length = sizeof(server);
      if (getsockname(sock, &server, &length)) {
            perror("getting socket name");
            exit(1);
      }
      printf("Socket has port #%d\n", ntohs(server.sin_port));

      /* Start accepting connections */
      listen(sock, 5);
      do {
            FD_ZERO(&ready);
            FD_SET(sock, &ready);
            to.tv_sec = 5;
            if (select(sock + 1, &ready, 0, 0, &to) < 0) {
                  perror("select");
                  continue;
```

```
            }
            if (FD_ISSET(sock, &ready)) {
                  msgsock = accept(sock, (struct sockaddr *)0, \
                              (int *)0);
                  if (msgsock == -1)
                        perror("accept");
                  else do {
                        bzero(buf, sizeof(buf));
                        if ((rval = read(msgsock, buf, 1024)) < 0)
                              perror("reading stream message");
                        else if (rval == 0)
                              printf("Ending connection\n");
                        else
                              printf("-->%s\n", buf);
                  } while (rval > 0);
                  close(msgsock);
            } else
                  printf("Do something else\n");
      } while (TRUE);
}
```

Figure 16-11: Using **select**() to check for pending connections

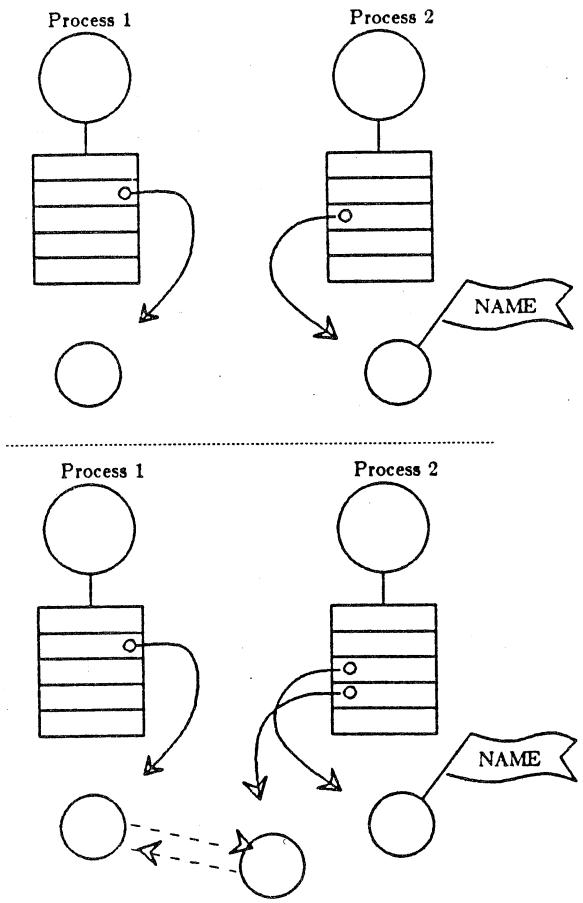


Figure 16-12: Establishing a stream connection

Forming a connection is asymmetrical; one process, such as the program in Figure 16-9, requests a connection with a particular socket, the other process accepts connection requests. Before a connection can be accepted a socket must be created and an address bound to it. This situation is illustrated in the top half of Figure 16-12. Process 2 has created a socket and bound a port number to it. Process 1 has created an unnamed socket. The address bound to process 2's socket is then made known to process 1 and, perhaps to several other potential communicants as well. If there are several possible communicants, this one socket might receive several requests for connections. As a result, a new socket is created for each connection. This new socket is the endpoint for communication within this process for this connection. A connection may be destroyed by closing the corresponding socket.

The program in Figure 16-10 is a rather trivial example of a server. It creates a socket to which it binds a name, which it then advertises. (In this case it prints out the socket number.) The program then calls *listen()* for this socket. Since several clients may attempt to connect more or less simultaneously, a queue of pending connections is maintained in the system address space. *Listen()* marks the socket as willing to accept connections and initializes the queue. When a connection is requested, it is listed in the queue. If the queue is full, an error status may be returned to the requester. The maximum length of this queue is specified by the second argument of

listen(); the maximum length is limited by the system. Once the listen call has been completed, the program enters an infinite loop. On each pass through the loop, a new connection is accepted and removed from the queue, and, hence, a new socket for the connection is created. The bottom half of Figure 16-12 shows the result of Process 1 connecting with the named socket of Process 2, and Process 2 accepting the connection. After the connection is created, the service, in this case printing out the messages, is performed and the connection socket closed. The *accept()* call will take a pending connection request from the queue if one is available, or block waiting for a request. Messages are read from the connection socket. Reads from an active connection will normally block until data is available. The number of bytes read is returned. When a connection is destroyed, the read call returns immediately. The number of bytes returned will be zero.

The program in Figure 16-11 is a slight variation on the server in Figure 16-10. It avoids blocking when there are no pending connection requests by calling *select()* to check for pending requests before calling *accept()*. This strategy is useful when connections may be received on more than one socket, or when data may arrive on other connected sockets before another connection request.

The programs in Figures 16-13 and 16-14 show a program using stream communication in the UNIX domain. Streams in the UNIX domain can be used for this sort of program in exactly the same way as Internet domain streams, except for the form of the names and the restriction of the connections to a single file system. There are some differences, however, in the functionality of streams in the two domains, notably in the handling of *out-of-band* data (discussed briefly below). These differences are beyond the scope of this chapter.

```
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <stdio.h>

#define DATA "Half a league, half a league . . ."

/*
 * This program connects to the socket named in the
 * command line and sends a one line message to that
 * socket. The form of the command line is
 * ustreamwrite pathname
 */
main(argc, argv)
      int argc;
      char *argv[];
{
      int sock;
      struct sockaddr_un server;
      char buf[1024];

      /* Create socket */
      sock = socket(AF_UNIX, SOCK_STREAM, 0);
      if (sock < 0) {
            perror("opening stream socket");
            exit(1);
      }
      /* Connect socket using name specified by command line. */
      server.sun_family = AF_UNIX;
```

```
        strcpy(server.sun_path, argv[1]);
        if (connect(sock, &server, sizeof(struct sockaddr_un)) < 0) {
                close(sock);
                perror("connecting stream socket");
                exit(1);
        }
        if (write(sock, DATA, sizeof(DATA)) < 0)
                perror("writing on stream socket");
}
```

Figure 16-13: Initiating a UNIX domain stream connection

```
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <stdio.h>

#define NAME "socket"

/*
 * This program creates a socket in the UNIX domain and
 * binds a name to it.  After printing the socket's name
 * it begins a loop. Each time through the loop it
 * accepts a connection and prints out messages from
 * it.  When the connection breaks, or a termination
 * message comes through, the program accepts a new
 * connection.
 */
main()
{
      int sock, msgsock, rval;
      struct sockaddr_un server;
      char buf[1024];

      /* Create socket */
      sock = socket(AF_UNIX, SOCK_STREAM, 0);
      if (sock < 0) {
            perror("opening stream socket");
            exit(1);
      }
      /* Name socket using file system name */
      server.sun_family = AF_UNIX;
      strcpy(server.sun_path, NAME);
      if (bind(sock, &server, sizeof(struct sockaddr_un))) {
            perror("binding stream socket");
            exit(1);
      }
      printf("Socket has name %s\n", server.sun_path);
      /* Start accepting connections */
      listen(sock, 5);
      for (;;) {
            msgsock = accept(sock, 0, 0);
            if (msgsock == -1)
                  perror("accept");
            else do {
                  bzero(buf, sizeof(buf));
                  if ((rval = read(msgsock, buf, 1024)) < 0)
                        perror("reading stream message");
                  else if (rval == 0)
                        printf("Ending connection\n");
                  else
                        printf("-->%s\n", buf);
            } while (rval > 0);
            close(msgsock);
      }
      /*
```

```
         *  The following statements are not executed,
         *  because they follow an infinite loop.
         * However, most ordinary programs will not run
         * forever.  In the UNIX domain it is necessary
         * to tell the file system that one is through
         * using NAME.  In most programs one uses the
         * call unlink() as below. Since the user will
         * have to kill this program, it will be
         * necessary to remove the name by a command
         * from the shell.
         */
        close(sock);
        unlink(NAME);
}
```

Figure 16-14: Accepting a UNIX domain stream connection

Reads, Writes, Recvs, etc.

UMIPS has several system calls for reading and writing information. The simplest calls are *read()* and *write()*. *Write()* takes as arguments the index of a descriptor, a pointer to a buffer containing the data and the size of the data. The descriptor may indicate either a file or a connected socket. "Connected" can mean either a connected stream socket (as described in the previous section) or a datagram socket for which a *connect()* call has provided a default destination (see the *connect()* manual page). *Read()* also takes a descriptor that indicates either a file or a socket. *Write()* requires a connected socket since no destination is specified in the parameters of the system call. *Read()* can be used for either a connected or an unconnected socket. These calls are, therefore, quite flexible and may be used to write applications that require no assumptions about the source of their input or the destination of their output. There are variations on *read()* and *write()* that allow the source and destination of the input and output to use several separate buffers, while retaining the flexibility to handle both files and sockets. These are *readv()* and *writev()*, for read and write *vector*.

It is sometimes necessary to send high priority data over a connection that may have unread low priority data at the other end. For example, a user interface process may be interpreting commands and sending them on to another process through a stream connection. The user interface may have filled the stream with as yet unprocessed requests when the user types a command to cancel all outstanding requests. Rather than have the high priority data wait to be processed after the low priority data, it is possible to send it as *out-of-band* (OOB) data. The notification of pending OOB data results in the generation of a SIGURG signal, if this signal has been enabled (see the manual page for *signal* or *sigvec*). See [Leffler 1986] for a more complete description of the OOB mechanism. There are a pair of calls similar to *read* and *write* that allow options, including sending and receiving OOB information; these are *send()* and *recv()*. These calls are used only with sockets; specifying a descriptor for a file will result in the return of an error status. These calls also allow *peeking* at data in a stream. That is, they allow a process to read data without removing the data from the stream. One use of this facility is to read ahead in a stream to determine the size of the next item to be read. When not using these options, these calls have the same functions as *read()* and *write()*.

To send datagrams, one must be allowed to specify the destination. The call *sendto()* takes a destination address as an argument and is therefore used for sending datagrams. The call *recvfrom()* is often used to read datagrams, since this call returns the address of the sender, if it is available, along with the data. If the identity of the sender does not matter, one may use *read()* or *recv()*.

Finally, there are a pair of calls that allow the sending and receiving of messages from multiple buffers, when the address of the recipient must be specified. These are *sendmsg()* and *recvmsg()*. These calls are actually quite general and have other uses, including, in the UNIX domain, the transmission of a file descriptor from one process to another.

The various options for reading and writing are shown in Figure 16-15, together with their parameters. The parameters for each system call reflect the differences in function of the different calls. In the examples given in this chapter, the calls *read()* and *write()* have been used whenever possible.

```
/*
 * The variable descriptor may be the descriptor
 * of either a file
 * or of a socket.
 */
cc = read(descriptor, buf, nbytes)
int cc, descriptor; char *buf; int nbytes;

/*
 * An iovec can include several source buffers.
 */
cc = readv(descriptor, iov, iovcnt)
int cc, descriptor; struct iovec *iov; int iovcnt;

cc = write(descriptor, buf, nbytes)
int cc, descriptor; char *buf; int nbytes;

cc = writev(descriptor, iovec, ioveclen)
int cc, descriptor; struct iovec *iovec; int ioveclen;

/*
 * The variable ''sock'' must be the descriptor of a
 * socket.  Flags may include MSG_OOB and MSG_PEEK.
 */
cc = send(sock, msg, len, flags)
int cc, sock; char *msg; int len, flags;

cc = sendto(sock, msg, len, flags, to, tolen)
int cc, sock; char *msg; int len, flags;
struct sockaddr *to; int tolen;

cc = sendmsg(sock, msg, flags)
int cc, sock; struct msghdr msg[]; int flags;

cc = recv(sock, buf, len, flags)
int cc, sock; char *buf; int len, flags;

cc = recvfrom(sock, buf, len, flags, from, fromlen)
int cc, sock; char *buf; int len, flags;
struct sockaddr *from; int *fromlen;

cc = recvmsg(sock, msg, flags)
int cc, socket; struct msghdr msg[]; int flags;
```

Figure 16-15: Varieties of read and write commands

Choices

This chapter has presented examples of some of the forms of communication supported by UMIPS. These have been presented in an order chosen for ease of presentation. It is useful to review these options emphasizing the factors that make each attractive.

Pipes have the advantage of portability, in that they are supported in all UNIX systems. They also are relatively simple to use. Socketpairs share this simplicity and have the additional advantage of allowing bidirectional communication. The major shortcoming of these mechanisms is that they require communicating processes to be descendants of a common process. They do not allow intermachine communication.

The two communication domains, UNIX and Internet, allow processes with no common ancestor to communicate. Of the two, only the Internet domain allows communication between machines. This makes the Internet domain a necessary choice for processes running on separate machines.

The choice between datagrams and stream communication is best made by carefully considering the semantic and performance requirements of the application. Streams can be both advantageous and disadvantageous. One disadvantage is that a process is only allowed a limited number of open streams, as there are usually only 64 entries available in the open descriptor table. This can cause problems if a single server must talk with a large number of clients. Another is that for delivering a short message the stream setup and teardown time can be unnecessarily long. Weighed against this are the reliability built into the streams. This will often be the deciding factor in favor of streams.

What to do Next

Many of the examples presented here can serve as models for multiprocess programs and for programs distributed across several machines. In developing a new multiprocess program, it is often easiest to first write the code to create the processes and communication paths. After this code is debugged, the code specific to the application can be added.

An introduction to the UNIX system and programming using UNIX system calls can be found in [Kernighan and Pike 1984]. Further documentation of the Berkeley UNIX 4.3BSD IPC mechanisms can be found in [Leffler et al. 1986]. More detailed information about particular calls and protocols is provided in sections 2, 3 and 7 of the *UNIX Programmer's Manual.* In particular the following manual pages are relevant:

creating and naming sockets	socket(2), bind(2)
establishing connections	listen(2), accept(2), connect(2)
transferring data	read(2), write(2), send(2), recv(2)
addresses	inet(7)
protocols	tcp(7), udp(7).

Acknowledgements

I would like to thank Sam Leffler and Mike Karels for their help in understanding the IPC mechanisms and all the people whose comments have helped in writing and improving this report.

This work was sponsored by the Defense Advanced Research Projects Agency (DoD), ARPA Order No. 4031, monitored by the Naval Electronics Systems Command under contract No. N00039-C-0235. The views and conclusions contained in this document are those of the author and should not be interpreted as representing official policies, either expressed or implied, of the Defense Research Projects Agency or of the US Government.

References

B.W. Kernighan & R. Pike, 1984,
The UNIX Programming Environment.
Englewood Cliffs, N.J.: Prentice-Hall.

B.W. Kernighan & D.M. Ritchie, 1978,
The C Programming Language,
Englewood Cliffs, N.J.: Prentice-Hall.

S.J. Leffler, R.S. Fabry, W.N. Joy, P. Lapsley, S. Miller & C. Torek, 1986,
An Advanced 4.3BSD Interprocess Communication Tutorial.
Computer Systems Research Group,
Department of Electrical Engineering and Computer Science,
University of California, Berkeley.

Computer Systems Research Group, 1986,
UNIX Programmer's Manual, 4.3 Berkeley Software Distribution.
Computer Systems Research Group,
Department of Electrical Engineering and Computer Science,
University of California, Berkeley.

Chapter 17: Advanced IPC Tutorial

Introduction

An Advanced 4.3BSD Interprocess Communication Tutorial

Samuel J. Leffler
Robert S. Fabry
William N. Joy
Phil Lapsley
Computer Systems Research Group
Department of Electrical Engineering and Computer Science
University of California, Berkeley
Berkeley, California 94720
Steve Miller
Chris Torek
Heterogeneous Systems Laboratory
Department of Computer Science
University of Maryland, College Park
College Park, Maryland 20742

This document provides an introduction to the interprocess communication facilities included in the 4.3BSD release of the UNIX system.

It discusses the overall model for interprocess communication and introduces the interprocess communication primitives which have been added to the system. The majority of the document considers the use of these primitives in developing applications. The reader is expected to be familiar with the C programming language as all examples are written in C. One of the most important additions to UNIX in 4.2BSD was interprocess communication. These facilities were the result of more than two years of discussion and research. The facilities provided in 4.2BSD incorporated many of the ideas from current research, while trying to maintain the UNIX philosophy of simplicity and conciseness. The current release of Berkeley UNIX, 4.3BSD, completes some of the IPC facilities and provides an upward-compatible interface. It is hoped that the interprocess communication facilities included in 4.3BSD will establish a standard for UNIX. From the response to the design, it appears many organizations carrying out work with UNIX are adopting it.

UNIX has previously been very weak in the area of interprocess communication. Prior to the 4BSD facilities, the only standard mechanism which allowed two processes to communicate were pipes (the mpx files which were part of Version 7 were experimental). Unfortunately, pipes are very restrictive in that the two communicating processes must be related through a common ancestor. Further, the semantics of pipes makes them almost impossible to maintain in a distributed environment.

Earlier attempts at extending the IPC facilities of UNIX have met with mixed reaction. The majority of the problems have been related to the fact that these facilities have been tied to the UNIX file system, either through naming or implementation. Consequently, the IPC facilities provided in 4.3BSD have been designed as a totally independent subsystem. The 4.3BSD IPC allows processes to rendezvous in many ways. Processes may rendezvous through a UNIX file system-like name space (a space where all names are path names) as well as through a network name space. In fact, new name spaces may be added at a future time with only minor changes visible to users. Further, the communication facilities have been extended to include more than the simple byte stream provided by a pipe. These extensions have resulted in a completely new part of the system which users will need time to familiarize themselves with. It is likely that as more use is made of these facilities they will be refined; only time will tell.

This document provides a high-level description of the IPC facilities in 4.3BSD and their use. It is designed to complement the manual pages for the IPC primitives by examples of their use. The remainder of this document is organized in four sections. "Basics" introduces the IPC-related system calls and the basic model of communication. "Network Library Routines" describes some of the supporting library routines users may find useful in constructing distributed applications. "Client/Server Model" is concerned with the client/server model used in developing applications and includes examples of the two major types of servers. "Advanced Topics" delves into advanced topics which sophisticated users are likely to encounter when using the IPC facilities.

Basics

The basic building block for communication is the *socket*. A socket is an endpoint of communication to which a name may be *bound*. Each socket in use has a *type* and one or more associated processes. Sockets exist within *communication domains*. A communication domain is an abstraction introduced to bundle common properties of processes communicating through sockets. One such property is the scheme used to name sockets. For example, in the UNIX communication domain sockets are named with UNIX path names; e.g. a socket may be named "/dev/foo". Sockets normally exchange data only with sockets in the same domain (it may be possible to cross domain boundaries, but only if some translation process is performed). The 4.3BSD IPC facilities support three separate communication domains: the UNIX domain, for on-system communication; the Internet domain, which is used by processes which communicate using the the DARPA standard communication protocols; and the NS domain, which is used by processes which communicate using the Xerox standard communication protocols. (See *Internet Transport Protocols*, Xerox System Integration Standard (XSIS)028112 for more information.) The underlying communication facilities provided by these domains have a significant influence on the internal system implementation as well as the interface to socket facilities available to a user. An example of the latter is that a socket "operating" in the UNIX domain sees a subset of the error conditions which are possible when operating in the Internet (or NS) domain.

Socket types

Sockets are typed according to the communication properties visible to a user. Processes are presumed to communicate only between sockets of the same type, although there is nothing that prevents communication between sockets of different types should the underlying communication protocols support this.

Four types of sockets currently are available to a user. A *stream* socket provides for the bidirectional, reliable, sequenced, and unduplicated flow of data without record boundaries. Aside from the bidirectionality of data flow, a pair of connected stream sockets provides an interface nearly identical to that of pipes. In the UNIX domain, in fact, the semantics are identical and, as one might expect, pipes have been implemented internally as simply a pair of connected stream sockets.

A *datagram* socket supports bidirectional flow of data which is not promised to be sequenced, reliable, or unduplicated. That is, a process receiving messages on a datagram socket may find messages duplicated, and, possibly, in an order different from the order in which it was sent. An important characteristic of a datagram socket is that record boundaries in data are preserved. Datagram sockets closely model the facilities found in many contemporary packet switched networks such as the Ethernet.

A *raw* socket provides users access to the underlying communication protocols which support socket abstractions. These sockets are normally datagram oriented, though their exact characteristics are dependent on the interface provided by the protocol. Raw sockets are not intended for the general user; they have been provided mainly for those interested in developing new communication protocols, or for gaining access to some of the more esoteric facilities of an existing protocol. The use of raw sockets is considered under "Advanced Topics".

A *sequenced packet* socket is similar to a stream socket, with the exception that record boundaries are preserved. This interface is provided only as part of the NS socket abstraction, and is very important in most serious NS applications. Sequenced-packet sockets allow the user to manipulate the SPP or IDP headers on a packet or a group of packets either by writing a prototype header along with whatever data is to be sent, or by specifying a default header to be used with all outgoing data, and allows the user to receive the headers on incoming packets. The use of these options is considered in the section "Advanced Topics".

Another potential socket type which has interesting properties is the *reliably delivered message* socket. The reliably delivered message socket has similar properties to a datagram socket, but with reliable delivery. There is currently no support for this type of socket, but a reliably delivered message protocol similar to Xerox's Packet Exchange Protocol (PEX) may be simulated at the user level. More information on this topic can be found under "Advanced Topics".

Socket creation

To create a socket the *socket* system call is used:

```
s = socket(domain, type, protocol);
```

This call requests that the system create a socket in the specified *domain* and of the specified *type*. A particular protocol may also be requested. If the protocol is left unspecified (a value of 0), the system will select an appropriate protocol from those protocols which comprise the communication domain and which may be used to support the requested socket type. The user is returned a descriptor (a small integer number) which may be used in later system calls which operate on sockets. The domain is specified as one of the manifest constants defined in the file *<sys/socket.h>*. For the UNIX domain the constant is AF_UNIX; for the Internet domain AF_INET; and for the NS domain, AF_NS. (The manifest constants are named AF_whatever as they indicate the "address format" to use in interpreting names.) The socket types are also defined in this file and one of SOCK_STREAM, SOCK_DGRAM, SOCK_RAW, or SOCK_SEQPACKET must be specified. To create a stream socket in the Internet domain the following call might be used:

```
s = socket(AF_INET, SOCK_STREAM, 0);
```

This call would result in a stream socket being created with the TCP protocol providing the underlying communication support. To create a datagram socket for on-machine use the call might be:

```
s = socket(AF_UNIX, SOCK_DGRAM, 0);
```

The default protocol (used when the *protocol* argument to the *socket* call is 0) should be correct for most every situation. However, it is possible to specify a protocol other than the default; this will be covered in "Advanced Topics".

There are several reasons a socket call may fail. Aside from the rare occurrence of lack of memory (ENOBUFS), a socket request may fail due to a request for an unknown protocol (EPROTONOSUPPORT), or a request for a type of socket for which there is no supporting protocol (EPROTOTYPE).

Binding local names

A socket is created without a name. Until a name is bound to a socket, processes have no way to reference it and, consequently, no messages may be received on it. Communicating processes are bound by an *association*. In the Internet and NS domains, an association is composed of local and foreign addresses, and local and foreign ports, while in the UNIX domain, an association is composed of local and foreign path names (the phrase "foreign pathname" means a pathname created by a foreign process, not a pathname on a foreign system). In most domains, associations must be unique. In the Internet domain there may never be duplicate <protocol, local address, local port, foreign address, foreign port> definitions. UNIX domain sockets need not always be bound to a name, but when bound there may never be duplicate <protocol, local pathname, foreign pathname> definitions. The pathnames may not refer to files already existing on the system in 4.3; the situation may change in future releases.

The *bind* system call allows a process to specify half of an association, <local address, local port> (or <local pathname>), while the *connect* and *accept* primitives are used to complete a socket's association.

In the Internet domain, binding names to sockets can be fairly complex. Fortunately, it is usually not necessary to specifically bind an address and port number to a socket, because the *connect* and *send* calls will automatically bind an appropriate address if they are used with an unbound socket. The process of binding names to NS sockets is similar in most ways to that of binding names to Internet sockets.

The *bind* system call is used as follows:

```
bind(s, name, namelen);
```

The bound name is a variable length byte string which is interpreted by the supporting protocol(s). Its interpretation may vary from communication domain to communication domain (this is one of the properties which comprise the "domain"). As mentioned, in the Internet domain names contain an Internet address and port number. NS domain names contain an NS address and port number. In the UNIX domain, names contain a path name and a family, which is always AF_UNIX. If one wanted to bind the name "/tmp/foo" to a UNIX domain socket, the following code would be used:

Note that, although the tendency here is to call the "addr" structure "sun", doing so would cause problems if the code were ever ported to a Sun workstation.

```
#include <sys/un.h>
 ...
struct sockaddr_un addr;
 ...
strcpy(addr.sun_path, "/tmp/foo");
addr.sun_family = AF_UNIX;
bind(s, (struct sockaddr *) &addr, strlen(addr.sun_path) +
    sizeof (addr.sun_family));
```

Note that in determining the size of a UNIX domain address null bytes are not counted, which is why *strlen* is used. In the current implementation of UNIX domain IPC under 4.3BSD, the file name referred to in *addr.sun_path* is created as a socket in the system file space. The caller must, therefore, have write permission in the directory where *addr.sun_path* is to reside, and this file should be deleted by the caller when it is no longer needed. Future versions of 4BSD may not create this file.

In binding an Internet address things become more complicated. The actual call is similar,

```
#include <sys/types.h>
#include <netinet/in.h>
 ...
struct sockaddr_in sin;
 ...
bind(s, (struct sockaddr *) &sin, sizeof (sin));
```

but the selection of what to place in the address *sin* requires some discussion. We will come back to the problem of formulating Internet addresses later when the library routines used in name resolution are discussed.

Binding an NS address to a socket is even more difficult, especially since the Internet library routines do not work with NS hostnames. The actual call is again similar:

```
#include <sys/types.h>
#include <netns/ns.h>
 ...
struct sockaddr_ns sns;
 ...
bind(s, (struct sockaddr *) &sns, sizeof (sns));
```

Again, discussion of what to place in a "struct sockaddr_ns" will be deferred to the section "Network Library Routines".

Connection establishment

Connection establishment is usually asymmetric, with one process a "client" and the other a "server". The server, when willing to offer its advertised services, binds a socket to a well-known address associated with the service and then passively "listens" on its socket. It is then possible for an unrelated process to rendezvous with the server. The client requests services from the server by initiating a "connection" to the server's socket. On the client side the *connect* call is used to initiate a connection. Using the UNIX domain, this might appear as:

```
struct sockaddr_un server;
 ...
connect(s, (struct sockaddr *)&server, strlen(server.sun_path) +
    sizeof (server.sun_family));
```

while in the Internet domain:

```
struct sockaddr_in server;
 ...
connect(s, (struct sockaddr *)&server, sizeof (server));
```

and in the NS domain:

```
struct sockaddr_ns server;
 ...
connect(s, (struct sockaddr *)&server, sizeof (server));
```

where *server* in the example above would contain either the UNIX pathname, Internet address and port number, or NS address and port number of the server to which the client process wishes to speak. If the client process's socket is unbound at the time of the connect call, the system will automatically select and bind a name to the socket if necessary. This is the usual way that local addresses are bound to a socket.

An error is returned if the connection was unsuccessful (any name automatically bound by the system, however, remains). Otherwise, the socket is associated with the server and data transfer may begin. Some of the more common errors returned when a connection attempt fails are:

ETIMEDOUT
: After failing to establish a connection for a period of time, the system decided there was no point in retrying the connection attempt any more. This usually occurs because the destination host is down, or because problems in the network resulted in transmissions being lost.

ECONNREFUSED
: The host refused service for some reason. This is usually due to a server process not being present at the requested name.
These operational errors are returned based on status information delivered to the client host by the underlying communication services.
These operational errors can occur either because the network or host is unknown (no route to the network or host is present), or because of status information returned by intermediate gateways or switching nodes. Many times the status returned is not sufficient to distinguish a network being down from a host being down, in which case the system indicates the entire network is unreachable.

For the server to receive a client's connection it must perform two steps after binding its socket. The first is to indicate a willingness to listen for incoming connection requests:

```
listen(s, 5);
```

The second parameter to the *listen* call specifies the maximum number of outstanding connections which may be queued awaiting acceptance by the server process; this number may be limited by the system. Should a connection be requested while the queue is full, the connection will not be refused, but rather the individual messages which comprise the request will be ignored. This gives a harried server time to make room in its pending connection queue while the client retries the connection request. Had the connection been returned with the ECONNREFUSED error, the client would be unable to tell if the server was up or not. As it is now it is still possible to get the ETIMEDOUT error back, though this is unlikely. The backlog figure supplied with the listen call is currently limited by the system to a maximum of 5 pending connections on any one queue. This avoids the problem of processes hogging system resources by setting an infinite backlog, then ignoring all connection requests.

With a socket marked as listening, a server may *accept* a connection:

```
struct sockaddr_in from;
 ...
fromlen = sizeof (from);
newsock = accept(s, (struct sockaddr *)&from, &fromlen);
```

(For the UNIX domain, *from* would be declared as a *struct sockaddr_un*, and for the NS domain, *from* would be declared as a *struct sockaddr_ns*, but nothing different would need to be done as far as *fromlen* is concerned. In the examples which follow, only Internet routines will be discussed.) A new descriptor is returned on receipt of a connection (along with a new socket). If the server wishes to find out who its client is, it may supply a buffer for the client socket's name. The value-result parameter *fromlen* is initialized by the server to indicate how much space is associated with *from*, then modified on return to reflect the true size of the name. If the client's name is not of interest, the second parameter may be a null pointer.

Accept normally blocks. That is, *accept* will not return until a connection is available or the system call is interrupted by a signal to the process. Further, there is no way for a process to indicate it will accept connections from only a specific individual, or individuals. It is up to the user process to consider who the connection is from and close down the connection if it does not wish to speak to the process. If the server process wants to accept connections on more than one socket, or wants to avoid blocking on the accept call, there are alternatives; they will be considered under "Advanced Topics".

Data transfer

With a connection established, data may begin to flow. To send and receive data there are a number of possible calls. With the peer entity at each end of a connection anchored, a user can send or receive a message without specifying the peer. As one might expect, in this case, then the normal *read* and *write* system calls are usable,

```
write(s, buf, sizeof (buf));
read(s, buf, sizeof (buf));
```

In addition to *read* and *write*, the new calls *send* and *recv* may be used:

```
send(s, buf, sizeof (buf), flags);
recv(s, buf, sizeof (buf), flags);
```

While *send* and *recv* are virtually identical to *read* and *write*, the extra *flags* argument is important. The flags, defined in *<sys/socket.h>*, may be specified as a non-zero value if one or more of the following is required:

```
MSG_OOB          send/receive out of band data
MSG_PEEK         look at data without reading
MSG_DONTROUTE    send data without routing packets
```

Out of band data is a notion specific to stream sockets, and one which we will not immediately consider. The option to have data sent without routing applied to the outgoing packets is currently used only by the routing table management process, and is unlikely to be of interest to the casual user. The ability to preview data is, however, of interest. When MSG_PEEK is specified with a *recv* call, any data present is

returned to the user, but treated as still "unread". That is, the next *read* or *recv* call applied to the socket will return the data previously previewed.

Discarding sockets

Once a socket is no longer of interest, it may be discarded by applying a *close* to the descriptor,

```
close(s);
```

If data is associated with a socket which promises reliable delivery (e.g. a stream socket) when a close takes place, the system will continue to attempt to transfer the data. However, after a fairly long period of time, if the data is still undelivered, it will be discarded. Should a user have no use for any pending data, it may perform a *shutdown* on the socket prior to closing it. This call is of the form:

```
shutdown(s, how);
```

where *how* is 0 if the user is no longer interested in reading data, 1 if no more data will be sent, or 2 if no data is to be sent or received.

Connectionless sockets

To this point we have been concerned mostly with sockets which follow a connection oriented model. However, there is also support for connectionless interactions typical of the datagram facilities found in contemporary packet switched networks. A datagram socket provides a symmetric interface to data exchange. While processes are still likely to be client and server, there is no requirement for connection establishment. Instead, each message includes the destination address.

Datagram sockets are created as before. If a particular local address is needed, the *bind* operation must precede the first data transmission. Otherwise, the system will set the local address and/or port when data is first sent. To send data, the *sendto* primitive is used,

```
sendto(s, buf, buflen, flags, (struct sockaddr *)&to, tolen);
```

The *s*, *buf*, *buflen*, and *flags* parameters are used as before. The *to* and *tolen* values are used to indicate the address of the intended recipient of the message. When using an unreliable datagram interface, it is unlikely that any errors will be reported to the sender. When information is present locally to recognize a message that can not be delivered (for instance when a network is unreachable), the call will return −1 and the global value *errno* will contain an error number.

To receive messages on an unconnected datagram socket, the *recvfrom* primitive is provided:

```
recvfrom(s, buf, buflen, flags, (struct sockaddr *)&from, &fromlen);
```

Once again, the *fromlen* parameter is handled in a value-result fashion, initially containing the size of the *from* buffer, and modified on return to indicate the actual size of the address from which the datagram was received.

In addition to the two calls mentioned above, datagram sockets may also use the *connect* call to associate a socket with a specific destination address. In this case, any data sent on the socket will automatically be addressed to the connected peer, and only data received from that peer will be delivered to the user. Only one connected address is permitted for each socket at one time; a second connect will change the destination address, and a connect to a null address (family AF_UNSPEC) will disconnect. Connect requests on datagram sockets return immediately, as this simply results in the system recording the peer's address (as compared to a stream socket, where a connect request initiates establishment of an end to end connection). *Accept* and *listen* are not used with datagram sockets.

While a datagram socket socket is connected, errors from recent *send* calls may be returned asynchronously. These errors may be reported on subsequent operations on the socket, or a special socket option used with *getsockopt*, SO_ERROR, may be used to interrogate the error status. A *select* for reading or writing will return true when an error indication has been received. The next operation will return the error, and the error status is cleared. Other of the less important details of datagram sockets are described in "Advanced Topics".

Input/Output multiplexing

One last facility often used in developing applications is the ability to multiplex i/o requests among multiple sockets and/or files. This is done using the *select* call:

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

fd_set readmask, writemask, exceptmask;
struct timeval timeout;
 ...
select(nfds, &readmask, &writemask, &exceptmask, &timeout);
```

Select takes as arguments pointers to three sets, one for the set of file descriptors for which the caller wishes to be able to read data on, one for those descriptors to which data is to be written, and one for which exceptional conditions are pending; out-of-band data is the only exceptional condition currently implemented by the socket If the user is not interested in certain conditions (i.e., read, write, or exceptions), the corresponding argument to the *select* should be a null pointer.

Each set is actually a structure containing an array of long integer bit masks; the size of the array is set by the definition FD_SETSIZE. The array is be long enough to hold one bit for each of FD_SETSIZE file descriptors.

The macros FD_SET(*fd, &mask*) and FD_CLR(*fd, &mask*) have been provided for adding and removing file descriptor *fd* in the set *mask*. The set should be zeroed before use, and the macro FD_ZERO(*&mask*) has been provided to clear the set *mask*. The parameter *nfds* in the *select* call specifies the range of file descriptors (i.e. one plus the value of the largest descriptor) to be examined in a set.

A timeout value may be specified if the selection is not to last more than a predetermined period of time. If the fields in *timeout* are set to 0, the selection takes the form of a *poll*, returning immediately. If the last parameter is a null pointer, the selection will block indefinitely. (To be more specific, a return takes place only when a descriptor is selectable, or when a signal is received by the caller, interrupting the

system call.) *Select* normally returns the number of file descriptors selected; if the *select* call returns due to the timeout expiring, then the value 0 is returned. If the *select* terminates because of an error or interruption, a −1 is returned with the error number in *errno*, and with the file descriptor masks unchanged.

Assuming a successful return, the three sets will indicate which file descriptors are ready to be read from, written to, or have exceptional conditions pending. The status of a file descriptor in a select mask may be tested with the *FD_ISSET(fd, &mask)* macro, which returns a non-zero value if *fd* is a member of the set *mask*, and 0 if it is not.

To determine if there are connections waiting on a socket to be used with an *accept* call, *select* can be used, followed by a *FD_ISSET(fd, &mask)* macro to check for read readiness on the appropriate socket. If *FD_ISSET* returns a non-zero value, indicating permission to read, then a connection is pending on the socket.

As an example, to read data from two sockets, *s1* and *s2* as it is available from each and with a one-second timeout, the following code might be used:

```
#include <sys/time.h>
#include <sys/types.h>
 ...
fd_set read_template;
struct timeval wait;
 ...
for (;;) {
   wait.tv_sec = 1;      /* one second */
   wait.tv_usec = 0;

   FD_ZERO(&read_template);

   FD_SET(s1, &read_template);
   FD_SET(s2, &read_template);

   nb = select(FD_SETSIZE, &read_template, (fd_set *) 0, \
         (fd_set *) 0, &wait);
   if (nb <= 0) {
      An error occurred during the select, or
      the select timed out.
   }

   if (FD_ISSET(s1, &read_template)) {
      Socket #1 is ready to be read from.
   }

   if (FD_ISSET(s2, &read_template)) {
      Socket #2 is ready to be read from.
   }
}
```

In 4.2, the arguments to *select* were pointers to integers instead of pointers to *fd_sets*. This type of call will still work as long as the number of file descriptors being examined is less than the number of bits in an integer; however, the methods illustrated above should be used in all current programs.

Select provides a synchronous multiplexing scheme. Asynchronous notification of output completion, input availability, and exceptional conditions is possible through use of the SIGIO and SIGURG signals described in "Advanced Topics".

Network Library Routines

The discussion in "Basics" indicated the possible need to locate and construct network addresses when using the interprocess communication facilities in a distributed environment. To aid in this task a number of routines have been added to the standard C run-time library. In this section we will consider the new routines provided to manipulate network addresses. While the 4.3BSD networking facilities support both the DARPA standard Internet protocols and the Xerox NS protocols, most of the routines presented in this section do not apply to the NS domain. Unless otherwise stated, it should be assumed that the routines presented in this section do not apply to the NS domain.

Locating a service on a remote host requires many levels of mapping before client and server may communicate. A service is assigned a name which is intended for human consumption; e.g. "the *login server* on host monet". This name, and the name of the peer host, must then be translated into network *addresses* which are not necessarily suitable for human consumption. Finally, the address must then used in locating a physical *location* and *route* to the service. The specifics of these three mappings are likely to vary between network architectures. For instance, it is desirable for a network to not require hosts to be named in such a way that their physical location is known by the client host. Instead, underlying services in the network may discover the actual location of the host at the time a client host wishes to communicate. This ability to have hosts named in a location independent manner may induce overhead in connection establishment, as a discovery process must take place, but allows a host to be physically mobile without requiring it to notify its clientele of its current location.

Standard routines are provided for: mapping host names to network addresses, network names to network numbers, protocol names to protocol numbers, and service names to port numbers and the appropriate protocol to use in communicating with the server process. The file *<netdb.h>* must be included when using any of these routines.

Host names

An Internet host name to address mapping is represented by the *hostent* structure:

```
struct   hostent {
   char  *h_name;       /* official name of host */
   char  **h_aliases;   /* alias list */
   int   h_addrtype;    /* host address type (e.g., AF_INET) */
   int   h_length;      /* length of address */
   char  **h_addr_list; /* list of addresses, null terminated */
};

#define h_addr h_addr_list[0] /* first address, network byte order */
```

The routine *gethostbyname*(3N) takes an Internet host name and returns a *hostent* structure, while the routine *gethostbyaddr*(3N) maps Internet host addresses into a *hostent* structure.

The official name of the host and its public aliases are returned by these routines, along with the address type (family) and a null terminated list of variable length address. This list of addresses is required because it is possible for a host to have many addresses, all having the same name. The *h_addr* definition is provided for backward compatibility, and is defined to be the first address in the list of addresses

in the *hostent* structure.

The database for these calls is provided by the file */etc/hosts* (*hosts*(5)). Whan using the host table version of *gethostbyname*, only one adress will be returned, but all listed aliases will be included.

Unlike Internet names, NS names are always mapped into host addresses by the use of a standard NS *Clearinghouse service*, a distributed name and authentication server. The algorithms for mapping NS names to addresses via a Clearinghouse are rather complicated, and the routines are not part of the standard libraries. The user-contributed Courier (Xerox remote procedure call protocol) compiler contains routines to accomplish this mapping; see the documentation and examples provided therein for more information. It is expected that almost all software that has to communicate using NS will need to use the facilities of the Courier compiler.

An NS host address is represented by the following:

```
union ns_host {
      u_char      c_host[6];
      u_short     s_host[3];
};

union ns_net {
      u_char      c_net[4];
      u_short     s_net[2];
};

struct ns_addr {
      union ns_net      x_net;
      union ns_host     x_host;
      u_short     x_port;
};
```

The following code fragment inserts a known NS address into a *ns_addr*:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netns/ns.h>
 ...
u_long netnum;
struct sockaddr_ns dst;
 ...
bzero((char *)&dst, sizeof(dst));

/*
 * There is no convenient way to assign a long
 * integer to a ''union ns_net'' at present; in
 * the future, something will hopefully be provided,
 * but this is the portable way to go for now.
 * The network number below is the one for the NS net
 * that the desired host (gyre) is on.
 */
netnum = htonl(2266);
dst.sns_addr.x_net = *(union ns_net *) &netnum;
dst.sns_family = AF_NS;

/*
 * host 2.7.1.0.2a.18 == "gyre:Computer Science:UofMaryland"
 */
dst.sns_addr.x_host.c_host[0] = 0x02;
dst.sns_addr.x_host.c_host[1] = 0x07;
dst.sns_addr.x_host.c_host[2] = 0x01;
dst.sns_addr.x_host.c_host[3] = 0x00;
dst.sns_addr.x_host.c_host[4] = 0x2a;
dst.sns_addr.x_host.c_host[5] = 0x18;
dst.sns_addr.x_port = htons(75);
```

Network names

As for host names, routines for mapping network names to numbers, and back, are provided. These routines return a *netent* structure:

```
/*
 * Assumption here is that a network number
 * fits in 32 bits -- probably a poor one.
 */
struct   netent {
   char  *n_name;       /* official name of net */
   char  **n_aliases;   /* alias list */
   int   n_addrtype;    /* net address type */
   int   n_net;         /* network number, host byte order */
};
```

The routines *getnetbyname*(3N), *getnetbynumber*(3N), and *getnetent*(3N) are the network counterparts to the host routines described above. The routines extract their information from */etc/networks*.

NS network numbers are determined either by asking your local Xerox Network Administrator (and hardcoding the information into your code), or by querying the Clearinghouse for addresses. The internetwork router is the only process that needs to manipulate network numbers on a regular basis; if a process wishes to communicate with a machine, it should ask the Clearinghouse for that machine's address (which will include the net number).

Protocol names

For protocols, which are defined in */etc/protocols*, the *protoent* structure defines the protocol-name mapping used with the routines *getprotobyname*(3N), *getprotobynumber*(3N), and *getprotoent*(3N):

```
struct   protoent {
   char  *p_name;        /* official protocol name */
   char  **p_aliases;    /* alias list */
   int   p_proto;        /* protocol number */
};
```

In the NS domain, protocols are indicated by the "client type" field of a IDP header. No protocol database exists; see "Advanced Topics" for more information.

Service names

Information regarding services is a bit more complicated. A service is expected to reside at a specific "port" and employ a particular communication protocol. This view is consistent with the Internet domain, but inconsistent with other network architectures. Further, a service may reside on multiple ports. If this occurs, the higher level library routines will have to be bypassed or extended. Services available are contained in the file */etc/services*. A service mapping is described by the *servent* structure,

```
struct   servent {
   char  *s_name;        /* official service name */
   char  **s_aliases;    /* alias list */
   int   s_port;         /* port number, network byte order */
   char  *s_proto;       /* protocol to use */
};
```

The routine *getservbyname*(3N) maps service names to a servent structure by specifying a service name and, optionally, a qualifying protocol. Thus the call

```
sp = getservbyname("telnet", (char *) 0);
```

returns the service specification for a telnet server using any protocol, while the call

```
sp = getservbyname("telnet", "tcp");
```

returns only that telnet server which uses the TCP protocol. The routines *getservbyport*(3N) and *getservent*(3N) are also provided. The *getservbyport* routine has an interface similar to that provided by *getservbyname*; an optional protocol name may

be specified to qualify lookups.

In the NS domain, services are handled by a central dispatcher provided as part of the Courier remote procedure call facilities. Again, the reader is referred to the Courier compiler documentation and to the Xerox standard (*Courier: The Remote Procedure Call Protocol*, XSIS 038112) for further details.

Miscellaneous

With the support routines described above, an Internet application program should rarely have to deal directly with addresses. This allows services to be developed as much as possible in a network independent fashion. It is clear, however, that purging all network dependencies is very difficult. So long as the user is required to supply network addresses when naming services and sockets there will always some network dependency in a program. For example, the normal code included in client programs, such as the remote login program, is of the form shown in Figure 17-2. (This example will be considered in more detail in "Client/Server Model".)

If we wanted to make the remote login program independent of the Internet protocols and addressing scheme we would be forced to add a layer of routines which masked the network dependent aspects from the mainstream login code. For the current facilities available in the system this does not appear to be worthwhile.

Aside from the address-related data base routines, there are several other routines available in the run-time library which are of interest to users. These are intended mostly to simplify manipulation of names and addresses. Figure 17-1 summarizes the routines for manipulating variable length byte strings and handling byte swapping of network addresses and values.

Call	Synopsis
bcmp(s1, s2, n)	compare byte-strings; 0 if same, not 0 otherwise
bcopy(s1, s2, n)	copy n bytes from s1 to s2
bzero(base, n)	zero-fill n bytes starting at base
htonl(val)	convert 32-bit quantity from host to network byte order
htons(val)	convert 16-bit quantity from host to network byte order
ntohl(val)	convert 32-bit quantity from network to host byte order
ntohs(val)	convert 16-bit quantity from network to host byte order

Figure 17-1: C run-time routines

The byte swapping routines are provided because the operating system expects addresses to be supplied in network order. On some architectures, such as the VAX, host byte ordering is different than network byte ordering. Consequently, programs are sometimes required to byte swap quantities. The library routines which return network addresses provide them in network order so that they may simply be copied into the structures provided to the system. This implies users should encounter the

byte swapping problem only when *interpreting* network addresses. For example, if an Internet port is to be printed out the following code would be required:

```
printf("port number %d\n", ntohs(sp->s_port));
```

On machines where unneeded these routines are defined as null macros.

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <stdio.h>
#include <netdb.h>
 ...
main(argc, argv)
   int argc;
   char *argv[];
{
   struct sockaddr_in server;
   struct servent *sp;
   struct hostent *hp;
   int s;
   ...
   sp = getservbyname("login", "tcp");
   if (sp == NULL) {
      fprintf(stderr, "rlogin: tcp/login: unknown service\n");
      exit(1);
   }
   hp = gethostbyname(argv[1]);
   if (hp == NULL) {
      fprintf(stderr, "rlogin: %s: unknown host\n", argv[1]);
      exit(2);
   }
   bzero((char *)&server, sizeof (server));
   bcopy(hp->h_addr, (char *)&server.sin_addr, hp->h_length);
   server.sin_family = hp->h_addrtype;
   server.sin_port = sp->s_port;
   s = socket(AF_INET, SOCK_STREAM, 0);
   if (s < 0) {
      perror("rlogin: socket");
      exit(3);
   }
   ...
   /* Connect does the bind() for us */

   if (connect(s, (char *)&server, sizeof (server)) < 0) {
      perror("rlogin: connect");
      exit(5);
   }
   ...
}
```

Figure 17-2: Remote login client code

Client/Server Model

The most commonly used paradigm in constructing distributed applications is the client/server model. In this scheme client applications request services from a server process. This implies an asymmetry in establishing communication between the client and server which has been examined in "Basics". In this section we will look more closely at the interactions between client and server, and consider some of the problems in developing client and server applications.

The client and server require a well known set of conventions before service may be rendered (and accepted). This set of conventions comprises a protocol which must be implemented at both ends of a connection. Depending on the situation, the protocol may be symmetric or asymmetric. In a symmetric protocol, either side may play the master or slave roles. In an asymmetric protocol, one side is immutably recognized as the master, with the other as the slave. An example of a symmetric protocol is the TELNET protocol used in the Internet for remote terminal emulation. An example of an asymmetric protocol is the Internet file transfer protocol, FTP. No matter whether the specific protocol used in obtaining a service is symmetric or asymmetric, when accessing a service there is a "client process" and a "server process". We will first consider the properties of server processes, then client processes.

A server process normally listens at a well known address for service requests. That is, the server process remains dormant until a connection is requested by a client's connection to the server's address. At such a time the server process "wakes up" and services the client, performing whatever appropriate actions the client requests of it.

Alternative schemes which use a service server may be used to eliminate a flock of server processes clogging the system while remaining dormant most of the time. For Internet servers in 4.3BSD, this scheme has been implemented via *inetd*, the so called "internet super-server." *Inetd* listens at a variety of ports, determined at start-up by reading a configuration file. When a connection is requested to a port on which *inetd* is listening, *inetd* executes the appropriate server program to handle the client. With this method, clients are unaware that an intermediary such as *inetd* has played any part in the connection. *Inetd* will be described in more detail in "Advanced Topics".

A similar alternative scheme is used by most Xerox services. In general, the Courier dispatch process (if used) accepts connections from processes requesting services of some sort or another. The client processes request a particular <program number, version number, procedure number> triple. If the dispatcher knows of such a program, it is started to handle the request; if not, an error is reported to the client. In this way, only one port is required to service a large variety of different requests. Again, the Courier facilities are not available without the use and installation of the Courier compiler. The information presented in this section applies only to NS clients and services that do not use Courier.

Servers

In 4.3BSD most servers are accessed at well known Internet addresses or UNIX domain names. For example, the remote login server's main loop is of the form shown in Figure 17-3.

```
main(argc, argv)
   int argc;
   char *argv[];
{
   int f;
   struct sockaddr_in from;
   struct servent *sp;

   sp = getservbyname("login", "tcp");
   if (sp == NULL) {
      fprintf(stderr, "rlogind: tcp/login: unknown service\n");
      exit(1);
   }
   ...
#ifndef DEBUG
   /* Disassociate server from controlling terminal */
   ...
#endif

   sin.sin_port = sp->s_port;
         /* Restricted port -- see "Advanced Topics" */
   ...
   f = socket(AF_INET, SOCK_STREAM, 0);
   ...
   if (bind(f, (struct sockaddr *) &sin, sizeof (sin)) < 0) {
      ...
   }
   ...
   listen(f, 5);
   for (;;) {
      int g, len = sizeof (from);

      g = accept(f, (struct sockaddr *) &from, &len);
      if (g < 0) {
         if (errno != EINTR)
            syslog(LOG_ERR, "rlogind: accept: %m");
         continue;
      }
      if (fork() == 0) {
         close(f);
         doit(g, &from);
      }
      close(g);
   }
}
```

Figure 17-3: Remote login server

The first step taken by the server is look up its service definition:

```
sp = getservbyname("login", "tcp");
if (sp == NULL) {
    fprintf(stderr, "rlogind: tcp/login: unknown service\n");
    exit(1);
}
```

The result of the *getservbyname* call is used in later portions of the code to define the Internet port at which it listens for service requests (indicated by a connection).

Step two is to disassociate the server from the controlling terminal of its invoker:

```
for (i = 0; i < 3; ++i)
    close(i);

open("/", O_RDONLY);
dup2(0, 1);
dup2(0, 2);

i = open("/dev/tty", O_RDWR);
if (i >= 0) {
    ioctl(i, TIOCNOTTY, 0);
    close(i);
}
```

This step is important as the server will likely not want to receive signals delivered to the process group of the controlling terminal. Note, however, that once a server has disassociated itself it can no longer send reports of errors to a terminal, and must log errors via *syslog*.

Once a server has established a pristine environment, it creates a socket and begins accepting service requests. The *bind* call is required to insure the server listens at its expected location. It should be noted that the remote login server listens at a restricted port number, and must therefore be run with a user-id of root. This concept of a "restricted port number" is 4BSD specific, and is covered under "Advanced Topics".

The main body of the loop is fairly simple:

```
for (;;) {
    int g, len = sizeof (from);

    g = accept(f, (struct sockaddr *)&from, &len);
    if (g < 0) {
        if (errno != EINTR)
        syslog(LOG_ERR, "rlogind: accept: %m");
        continue;
    }
    if (fork() == 0) {/* Child */
        close(f);
        doit(g, &from);
    }
    close(g);/* Parent */
}
```

An *accept* call blocks the server until a client requests service. This call could return a failure status if the call is interrupted by a signal such as SIGCHLD (to be discussed in "Advanced Topics"). Therefore, the return value from *accept* is checked to insure a connection has actually been established, and an error report is logged via *syslog* if an error has occurred.

With a connection in hand, the server then forks a child process and invokes the main body of the remote login protocol processing. Note how the socket used by the parent for queuing connection requests is closed in the child, while the socket created as a result of the *accept* is closed in the parent. The address of the client is also handed the *doit* routine because it requires it in authenticating clients.

Clients

The client side of the remote login service was shown earlier in Figure 17-2. One can see the separate, asymmetric roles of the client and server clearly in the code. The server is a passive entity, listening for client connections, while the client process is an active entity, initiating a connection when invoked.

Let us consider more closely the steps taken by the client remote login process. As in the server process, the first step is to locate the service definition for a remote login:

```
sp = getservbyname("login", "tcp");
if (sp == NULL) {
   fprintf(stderr, "rlogin: tcp/login: unknown service\n");
   exit(1);
}
```

Next the destination host is looked up with a *gethostbyname* call:

```
hp = gethostbyname(argv[1]);
if (hp == NULL) {
   fprintf(stderr, "rlogin: %s: unknown host\n", argv[1]);
   exit(2);
}
```

With this accomplished, all that is required is to establish a connection to the server at the requested host and start up the remote login protocol. The address buffer is cleared, then filled in with the Internet address of the foreign host and the port number at which the login process resides on the foreign host:

```
bzero((char *)&server, sizeof (server));
bcopy(hp->h_addr, (char *) &server.sin_addr, hp->h_length);
server.sin_family = hp->h_addrtype;
server.sin_port = sp->s_port;
```

A socket is created, and a connection initiated. Note that *connect* implicitly performs a *bind* call, since *s* is unbound.

```
s = socket(hp->h_addrtype, SOCK_STREAM, 0);
if (s < 0) {
   perror("rlogin: socket");
   exit(3);
}
 ...
if (connect(s, (struct sockaddr *) &server, sizeof (server)) < 0) {
   perror("rlogin: connect");
   exit(4);
}
```

The details of the remote login protocol will not be considered here.

Connectionless servers

While connection-based services are the norm, some services are based on the use of datagram sockets. One, in particular, is the "rwho" service which provides users with status information for hosts connected to a local area network. This service, while predicated on the ability to *broadcast* information to all hosts connected to a particular network, is of interest as an example usage of datagram sockets.

A user on any machine running the rwho server may find out the current status of a machine with the *ruptime*(1) program. The output generated is illustrated in Figure 17-4.

```
arpa        up   9:45,      5 users, load   1.15,   1.39,   1.31
cad         up   2+12:04,   8 users, load   4.67,   5.13,   4.59
calder      up   10:10,      0 users, load   0.27,   0.15,   0.14
dali        up   2+06:28,   9 users, load   1.04,   1.20,   1.65
degas       up   25+09:48,  0 users, load   1.49,   1.43,   1.41
ear         up   5+00:05,   0 users, load   1.51,   1.54,   1.56
ernie     down   0:24
esvax     down   17:04
ingres    down   0:26
kim         up   3+09:16,   8 users, load   2.03,   2.46,   3.11
matisse     up   3+06:18,   0 users, load   0.03,   0.03,   0.05
medea       up   3+09:39,   2 users, load   0.35,   0.37,   0.50
merlin    down   19+15:37
miro        up   1+07:20,   7 users, load   4.59,   3.28,   2.12
monet       up   1+00:43,   2 users, load   0.22,   0.09,   0.07
oz        down   16:09
statvax     up   2+15:57,   3 users, load   1.52,   1.81,   1.86
ucbvax      up   9:34,      2 users, load   6.08,   5.16,   3.28
```

Figure 17-4: **ruptime** output

Status information for each host is periodically broadcast by rwho server processes on each machine. The same server process also receives the status information and uses it to update a database. This database is then interpreted to generate the status information for each host. Servers operate autonomously, coupled only by the local network and its broadcast capabilities.

Note that the use of broadcast for such a task is fairly inefficient, as all hosts must process each message, whether or not using an rwho server. Unless such a service is sufficiently universal and is frequently used, the expense of periodic broadcasts outweighs the simplicity.

The rwho server, in a simplified form, is pictured in Figure 17-5. There are two separate tasks performed by the server. The first task is to act as a receiver of status information broadcast by other hosts on the network. This job is carried out in the main loop of the program. Packets received at the rwho port are interrogated to insure they've been sent by another rwho server process, then are time stamped with their arrival time and used to update a file indicating the status of the host. When a host has not been heard from for an extended period of time, the database interpretation routines assume the host is down and indicate such on the status reports. This algorithm is prone to error as a server may be down while a host is actually up, but serves our current needs.

```
main()
{
    ...
    sp = getservbyname("who", "udp");
    net = getnetbyname("localnet");
    sin.sin_addr = inet_makeaddr(INADDR_ANY, net);
    sin.sin_port = sp->s_port;
    ...
    s = socket(AF_INET, SOCK_DGRAM, 0);
    ...
    on = 1;
    if (setsockopt(s, SOL_SOCKET, SO_BROADCAST, &on, \
            sizeof(on)) < 0) {
        syslog(LOG_ERR, "setsockopt SO_BROADCAST: %m");
        exit(1);
    }
    bind(s, (struct sockaddr *) &sin, sizeof (sin));
    ...
    signal(SIGALRM, onalrm);
    onalrm();
    for (;;) {
        struct whod wd;
        int cc, whod, len = sizeof (from);

        cc = recvfrom(s, (char *)&wd, sizeof (struct whod), 0,
            (struct sockaddr *)&from, &len);
        if (cc <= 0) {
            if (cc < 0 && errno != EINTR)
                syslog(LOG_ERR, "rwhod: recv: %m");
            continue;
        }
        if (from.sin_port != sp->s_port) {
            syslog(LOG_ERR, "rwhod: %d: bad from port",
                ntohs(from.sin_port));
            continue;
        }
        ...
        if (!verify(wd.wd_hostname)) {
            syslog(LOG_ERR, "rwhod: malformed host name from %x",
                ntohl(from.sin_addr.s_addr));
            continue;
        }
        (void) sprintf(path, "%s/whod.%s", RWHODIR, wd.wd_hostname);
        whod = open(path, O_WRONLY | O_CREAT | O_TRUNC, 0666);
        ...
        (void) time(&wd.wd_recvtime);
        (void) write(whod, (char *)&wd, cc);
        (void) close(whod);
    }
}
```

Figure 17-5: **rwho** server

The second task performed by the server is to supply information regarding the status of its host. This involves periodically acquiring system status information, packaging it up in a message and broadcasting it on the local network for other rwho servers to hear. The supply function is triggered by a timer and runs off a signal. Locating the system status information is somewhat involved, but uninteresting. Deciding where to transmit the resultant packet is somewhat problematical, however.

Status information must be broadcast on the local network. For networks which do not support the notion of broadcast another scheme must be used to simulate or replace broadcasting. One possibility is to enumerate the known neighbors (based on the status messages received from other rwho servers). This, unfortunately, requires some bootstrapping information, for a server will have no idea what machines are its neighbors until it receives status messages from them. Therefore, if all machines on a net are freshly booted, no machine will have any known neighbors and thus never receive, or send, any status information. This is the identical problem faced by the routing table management process in propagating routing status information. The standard solution, unsatisfactory as it may be, is to inform one or more servers of known neighbors and request that they always communicate with these neighbors. If each server has at least one neighbor supplied to it, status information may then propagate through a neighbor to hosts which are not (possibly) directly neighbors. If the server is able to support networks which provide a broadcast capability, as well as those which do not, then networks with an arbitrary topology may share status information.

One must, however, be concerned about "loops". That is, if a host is connected to multiple networks, it will receive status information from itself. This can lead to an endless, wasteful, exchange of information.

It is important that software operating in a distributed environment not have any site-dependent information compiled into it. This would require a separate copy of the server at each host and make maintenance a severe headache. 4.3BSD attempts to isolate host-specific information from applications by providing system calls which return the necessary information. An example of such a system call is the *gethostname*(2) call which returns the host's "official" name.

A mechanism exists, in the form of an *ioctl* call, for finding the collection of networks to which a host is directly connected. Further, a local network broadcasting mechanism has been implemented at the socket level. Combining these two features allows a process to broadcast on any directly connected local network which supports the notion of broadcasting in a site independent manner. This allows 4.3BSD to solve the problem of deciding how to propagate status information in the case of *rwho*, or more generally in broadcasting: Such status information is broadcast to connected networks at the socket level, where the connected networks have been obtained via the appropriate *ioctl* calls. The specifics of such broadcastings are complex, however, and will be covered in "Advanced Topics".

Advanced Topics

A number of facilities have yet to be discussed. For most users of the IPC the mechanisms already described will suffice in constructing distributed applications. However, others will find the need to utilize some of the features which we consider in this section.

Out of band data

The stream socket abstraction includes the notion of "out of band" data. Out of band data is a logically independent transmission channel associated with each pair of connected stream sockets. Out of band data is delivered to the user independently of normal data. The abstraction defines that the out of band data facilities must support the reliable delivery of at least one out of band message at a time. This message may contain at least one byte of data, and at least one message may be pending delivery to the user at any one time. For communications protocols which support only in-band signaling (i.e. the urgent data is delivered in sequence with the normal data), the system normally extracts the data from the normal data stream and stores it separately. This allows users to choose between receiving the urgent data in order and receiving it out of sequence without having to buffer all the intervening data. It is possible to "peek" (via MSG_PEEK) at out of band data. If the socket has a process group, a SIGURG signal is generated when the protocol is notified of its existence. A process can set the process group or process id to be informed by the SIGURG signal via the appropriate *fcntl* call, as described below for SIGIO. If multiple sockets may have out of band data awaiting delivery, a *select* call for exceptional conditions may be used to determine those sockets with such data pending. Neither the signal nor the select indicate the actual arrival of the out-of-band data, but only notification that it is pending.

In addition to the information passed, a logical mark is placed in the data stream to indicate the point at which the out of band data was sent. The remote login and remote shell applications use this facility to propagate signals between client and server processes. When a signal flushs any pending output from the remote process(es), all data up to the mark in the data stream is discarded.

To send an out of band message the MSG_OOB flag is supplied to a *send* or *sendto* calls, while to receive out of band data MSG_OOB should be indicated when performing a *recvfrom* or *recv* call. To find out if the read pointer is currently pointing at the mark in the data stream, the SIOCATMARK ioctl is provided:

```
ioctl(s, SIOCATMARK, &yes);
```

If *yes* is a 1 on return, the next read will return data after the mark. Otherwise (assuming out of band data has arrived), the next read will provide data sent by the client prior to transmission of the out of band signal. The routine used in the remote login process to flush output on receipt of an interrupt or quit signal is shown in Figure 17-6. It reads the normal data up to the mark (to discard it), then reads the out-of-band byte.

```
#include <sys/ioctl.h>
#include <sys/file.h>
 ...
oob()
{
   int out = FWRITE;
   char waste[BUFSIZ], mark;

   /* flush local terminal output */
   ioctl(1, TIOCFLUSH, (char *)&out);
   for (;;) {
      if (ioctl(rem, SIOCATMARK, &mark) < 0) {
         perror("ioctl");
         break;
      }
      if (mark)
         break;
      (void) read(rem, waste, sizeof (waste));
   }
   if (recv(rem, &mark, 1, MSG_OOB) < 0) {
      perror("recv");
      ...
   }
   ...
}
```

Figure 17-6: Flushing terminal I/O on receipt of out of band data

A process may also read or peek at the out-of-band data without first reading up to the mark. This is more difficult when the underlying protocol delivers the urgent data in-band with the normal data, and only sends notification of its presence ahead of time (e.g., the TCP protocol used to implement streams in the Internet domain). With such protocols, the out-of-band byte may not yet have arrived when a *recv* is done with the MSG_OOB flag. In that case, the call will return an error of EWOULDBLOCK. Worse, there may be enough in-band data in the input buffer that normal flow control prevents the peer from sending the urgent data until the buffer is cleared. The process must then read enough of the queued data that the urgent data may be delivered.

Certain programs that use multiple bytes of urgent data and must handle multiple urgent signals (e.g., *telnet*(1C)) need to retain the position of urgent data within the stream. This treatment is available as a socket-level option, SO_OOBINLINE; see *setsockopt*(2) for usage. With this option, the position of urgent data (the "mark") is retained, but the urgent data immediately follows the mark within the normal data stream returned without the MSG_OOB flag. Reception of multiple urgent indications causes the mark to move, but no out-of-band data are lost.

Non-Blocking Sockets

It is occasionally convenient to make use of sockets which do not block; that is, I/O requests which cannot complete immediately and would therefore cause the process to be suspended awaiting completion are not executed, and an error code is returned. Once a socket has been created via the *socket* call, it may be marked as non-blocking by *fcntl* as follows:

```
#include <fcntl.h>
 ...
int   s;
 ...
s = socket(AF_INET, SOCK_STREAM, 0);
 ...
if (fcntl(s, F_SETFL, FNDELAY) < 0)
   perror("fcntl F_SETFL, FNDELAY");
   exit(1);
}
 ...
```

When performing non-blocking I/O on sockets, one must be careful to check for the error EWOULDBLOCK (stored in the global variable *errno*), which occurs when an operation would normally block, but the socket it was performed on is marked as non-blocking. In particular, *accept*, *connect*, *send*, *recv*, *read*, and *write* can all return EWOULDBLOCK, and processes should be prepared to deal with such return codes. If an operation such as a *send* cannot be done in its entirety, but partial writes are sensible (for example, when using a stream socket), the data that can be sent immediately will be processed, and the return value will indicate the amount actually sent.

Interrupt driven socket I/O

The SIGIO signal allows a process to be notified via a signal when a socket (or more generally, a file descriptor) has data waiting to be read. Use of the SIGIO facility requires three steps: First, the process must set up a SIGIO signal handler by use of the *signal* or *sigvec* calls. Second, it must set the process id or process group id which is to receive notification of pending input to its own process id, or the process group id of its process group (note that the default process group of a socket is group zero). This is accomplished by use of an *fcntl* call. Third, it must enable asynchronous notification of pending I/O requests with another *fcntl* call. Sample code to allow a given process to receive information on pending I/O requests as they occur for a socket *s* is given in Figure 17-7. With the addition of a handler for SIGURG, this code can also be used to prepare for receipt of SIGURG signals.

```
#include <fcntl.h>
 ...
int   io_handler();
 ...
signal(SIGIO, io_handler);

/* Set the process receiving SIGIO/SIGURG signals to us */

if (fcntl(s, F_SETOWN, getpid()) < 0) {
   perror("fcntl F_SETOWN");
   exit(1);
}

/* Allow receipt of asynchronous I/O signals */

if (fcntl(s, F_SETFL, FASYNC) < 0) {
   perror("fcntl F_SETFL, FASYNC");
   exit(1);
}
```

Figure 17-7: Use of asynchronous notification of I/O requests

Signals and process groups

Due to the existence of the SIGURG and SIGIO signals each socket has an associated process number, just as is done for terminals. This value is initialized to zero, but may be redefined at a later time with the F_SETOWN *fcntl*, such as was done in the code above for SIGIO. To set the socket's process id for signals, positive arguments should be given to the *fcntl* call. To set the socket's process group for signals, negative arguments should be passed to *fcntl*. Note that the process number indicates either the associated process id or the associated process group; it is impossible to specify both at the same time. A similar *fcntl*, F_GETOWN, is available for determining the current process number of a socket.

Another signal which is useful when constructing server processes is SIGCHLD. This signal is delivered to a process when any child processes have changed state. Normally servers use the signal to "reap" child processes that have exited without explicitly awaiting their termination or periodic polling for exit status. For example, the remote login server loop shown in Figure 17-2 may be augmented as shown in Figure 17-8.

```
int reaper();
 ...
signal(SIGCHLD, reaper);
listen(f, 5);
for (;;) {
   int g, len = sizeof (from);

   g = accept(f, (struct sockaddr *)&from, &len,);
   if (g < 0) {
      if (errno != EINTR)
         syslog(LOG_ERR, "rlogind: accept: %m");
      continue;
   }
   ...
}
 ...
#include <wait.h>
reaper()
{
   union wait status;

   while (wait3(&status, WNOHANG, 0) > 0)

}
```

Figure 17-8: Use of the SIGCHLD signal

If the parent server process fails to reap its children, a large number of "zombie" processes may be created.

Pseudo terminals

Many programs will not function properly without a terminal for standard input and output. Since sockets do not provide the semantics of terminals, it is often necessary to have a process communicating over the network do so through a *pseudo-terminal*. A pseudo- terminal is actually a pair of devices, master and slave, which allow a process to serve as an active agent in communication between processes and users. Data written on the slave side of a pseudo-terminal is supplied as input to a process reading from the master side, while data written on the master side are processed as terminal input for the slave. In this way, the process manipulating the master side of the pseudo-terminal has control over the information read and written on the slave side as if it were manipulating the keyboard and reading the screen on a real terminal. The purpose of this abstraction is to preserve terminal semantics over a network connection— that is, the slave side appears as a normal terminal to any process reading from or writing to it.

For example, the remote login server uses pseudo-terminals for remote login sessions. A user logging in to a machine across the network is provided a shell with a slave pseudo-terminal as standard input, output, and error. The server process then handles the communication between the programs invoked by the remote shell and the user's local client process. When a user sends a character that generates an

interrupt on the remote machine that flushes terminal output, the pseudo-terminal generates a control message for the server process. The server then sends an out of band message to the client process to signal a flush of data at the real terminal and on the intervening data buffered in the network.

Under 4.3BSD, the name of the slave side of a pseudo-terminal is of the form */dev/ttyxy*, where *x* is a single letter starting at 'p' and continuing to 't'. *y* is a hexadecimal digit (i.e., a single character in the range 0 through 9 or 'a' through 'f'). The master side of a pseudo-terminal is */dev/ptyxy*, where *x* and *y* correspond to the slave side of the pseudo-terminal.

In general, the method of obtaining a pair of master and slave pseudo-terminals is to find a pseudo-terminal which is not currently in use. The master half of a pseudo-terminal is a single-open device; thus, each master may be opened in turn until an open succeeds. The slave side of the pseudo-terminal is then opened, and is set to the proper terminal modes if necessary. The process then *fork*s; the child closes the master side of the pseudo-terminal, and *exec*s the appropriate program. Meanwhile, the parent closes the slave side of the pseudo-terminal and begins reading and writing from the master side. Sample code making use of pseudo-terminals is given in Figure 17-9; this code assumes that a connection on a socket *s* exists, connected to a peer who wants a service of some kind, and that the process has disassociated itself from any previous controlling terminal.

```
gotpty = 0;
for (c = 'p'; !gotpty && c <= 's'; c++) {
   line = "/dev/ptyXX";
   line[sizeof("/dev/pty")-1] = c;
   line[sizeof("/dev/ptyp")-1] = '0';
   if (stat(line, &statbuf) < 0)
      break;
   for (i = 0; i < 16; i++) {
      line[sizeof("/dev/ptyp")-1] = "0123456789abcdef"[i];
      master = open(line, O_RDWR);
      if (master > 0) {
         gotpty = 1;
         break;
      }
   }
}
if (!gotpty) {
   syslog(LOG_ERR, "All network ports in use");
   exit(1);
}

line[sizeof("/dev/")-1] = 't';
slave = open(line, O_RDWR);    /* slave is now slave side */
if (slave < 0) {
   syslog(LOG_ERR, "Cannot open slave pty %s", line);
   exit(1);
}

ioctl(slave, TIOCGETP, &b);    /* Set slave tty modes */
b.sg_flags = CRMOD|XTABS|ANYP;
ioctl(slave, TIOCSETP, &b);

i = fork();
if (i < 0) {
   syslog(LOG_ERR, "fork: %m");
   exit(1);
} else if (i) {       /* Parent */
   close(slave);
   ...
} else {      /* Child */
   (void) close(s);
   (void) close(master);
   dup2(slave, 0);
   dup2(slave, 1);
   dup2(slave, 2);
   if (slave > 2)
      (void) close(slave);
   ...
}
```

Figure 17-9: Creation and use of a pseudo terminal

Selecting specific protocols

If the third argument to the *socket* call is 0, *socket* will select a default protocol to use with the returned socket of the type requested. The default protocol is usually correct, and alternate choices are not usually available. However, when using "raw" sockets to communicate directly with lower-level protocols or hardware interfaces, the protocol argument may be important for setting up demultiplexing. For example, raw sockets in the Internet family may be used to implement a new protocol above IP, and the socket will receive packets only for the protocol specified. To obtain a particular protocol one determines the protocol number as defined within the communication domain. For the Internet domain one may use one of the library routines discussed in "Network Library Routines", such as *getprotobyname*:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
 ...
pp = getprotobyname("newtcp");
s = socket(AF_INET, SOCK_STREAM, pp->p_proto);
```

This would result in a socket *s* using a stream based connection, but with protocol type of "newtcp" instead of the default "tcp."

In the NS domain, the available socket protocols are defined in *<netns/ns.h>*. To create a raw socket for Xerox Error Protocol messages, one might use:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netns/ns.h>
 ...
s = socket(AF_NS, SOCK_RAW, NSPROTO_ERROR);
```

Address binding

As was mentioned in "Basics", binding addresses to sockets in the Internet and NS domains can be fairly complex. As a brief reminder, these associations are composed of local and foreign addresses, and local and foreign ports. Port numbers are allocated out of separate spaces, one for each system and one for each domain on that system. Through the *bind* system call, a process may specify half of an association, the <local address, local port> part, while the *connect* and *accept* primitives are used to complete a socket's association by specifying the <foreign address, foreign port> part. Since the association is created in two steps the association uniqueness requirement indicated previously could be violated unless care is taken. Further, it is unrealistic to expect user programs to always know proper values to use for the local address and local port since a host may reside on multiple networks and the set of allocated port numbers is not directly accessible to a user.

To simplify local address binding in the Internet domain the notion of a "wildcard" address has been provided. When an address is specified as INADDR_ANY (a manifest constant defined in <netinet/in.h>), the system interprets the address as "any valid address". For example, to bind a specific port number to a socket, but

leave the local address unspecified, the following code might be used:

```
#include <sys/types.h>
#include <netinet/in.h>
 ...
struct sockaddr_in sin;
 ...
s = socket(AF_INET, SOCK_STREAM, 0);
sin.sin_family = AF_INET;
sin.sin_addr.s_addr = htonl(INADDR_ANY);
sin.sin_port = htons(MYPORT);
bind(s, (struct sockaddr *) &sin, sizeof (sin));
```

Sockets with wildcarded local addresses may receive messages directed to the specified port number, and sent to any of the possible addresses assigned to a host. For example, if a host has addresses 128.32.0.4 and 10.0.0.78, and a socket is bound as above, the process will be able to accept connection requests which are addressed to 128.32.0.4 or 10.0.0.78. If a server process wished to only allow hosts on a given network connect to it, it would bind the address of the host on the appropriate network.

In a similar fashion, a local port may be left unspecified (specified as zero), in which case the system will select an appropriate port number for it. This shortcut will work both in the Internet and NS domains. For example, to bind a specific local address to a socket, but to leave the local port number unspecified:

```
hp = gethostbyname(hostname);
if (hp == NULL) {
   ...
}
bcopy(hp->h_addr, (char *) sin.sin_addr, hp->h_length);
sin.sin_port = htons(0);
bind(s, (struct sockaddr *) &sin, sizeof (sin));
```

The system selects the local port number based on two criteria. The first is that on 4BSD systems, Internet ports below IPPORT_RESERVED (1024) (for the Xerox domain, 0 through 3000) are reserved for privileged users (i.e., the super user); Internet ports above IPPORT_USERRESERVED (50000) are reserved for non-privileged servers. The second is that the port number is not currently bound to some other socket. In order to find a free Internet port number in the privileged range the *rresvport* library routine may be used as follows to return a stream socket in with a privileged port number:

```
int lport = IPPORT_RESERVED - 1;
int s;
 ...
s = rresvport(&lport);
if (s < 0) {
   if (errno == EAGAIN)
      fprintf(stderr, "socket: all ports in use\n");
   else
      perror("rresvport: socket");
   ...
}
```

The restriction on allocating ports was done to allow processes executing in a "secure" environment to perform authentication based on the originating address and port number. For example, the *rlogin*(1) command allows users to log in across a network without being asked for a password, if two conditions hold: First, the name of the system the user is logging in from is in the file */etc/hosts.equiv* on the system he is logging in to (or the system name and the user name are in the user's *.rhosts* file in the user's home directory), and second, that the user's rlogin process is coming from a privileged port on the machine from which he is logging. The port number and network address of the machine from which the user is logging in can be determined either by the *from* result of the *accept* call, or from the *getpeername* call.

In certain cases the algorithm used by the system in selecting port numbers is unsuitable for an application. This is because associations are created in a two step process. For example, the Internet file transfer protocol, FTP, specifies that data connections must always originate from the same local port. However, duplicate associations are avoided by connecting to different foreign ports. In this situation the system would disallow binding the same local address and port number to a socket if a previous data connection's socket still existed. To override the default port selection algorithm, an option call must be performed prior to address binding:

```
...
int   on = 1;
...
setsockopt(s, SOL_SOCKET, SO_REUSEADDR, &on, sizeof(on));
bind(s, (struct sockaddr *) &sin, sizeof (sin));
```

With the above call, local addresses may be bound which are already in use. This does not violate the uniqueness requirement as the system still checks at connect time to be sure any other sockets with the same local address and port do not have the same foreign address and port. If the association already exists, the error EADDRINUSE is returned.

Broadcasting and Determining Network Configuration

By using a datagram socket, it is possible to send broadcast packets on many networks supported by the system. The network itself must support broadcast; the system provides no simulation of broadcast in software. Broadcast messages can place a high load on a network since they force every host on the network to service them. Consequently, the ability to send broadcast packets has been limited to sockets which are explicitly marked as allowing broadcasting. Broadcast is typically used for one of two reasons: it is desired to find a resource on a local network without prior knowledge of its address, or important functions such as routing require that information be sent to all accessible neighbors.

To send a broadcast message, a datagram socket should be created:

```
s = socket(AF_INET, SOCK_DGRAM, 0);
```

or

```
s = socket(AF_NS, SOCK_DGRAM, 0);
```

The socket is marked as allowing broadcasting,

```
int   on = 1;

setsockopt(s, SOL_SOCKET, SO_BROADCAST, &on, sizeof (on));
```

and at least a port number should be bound to the socket:

```
sin.sin_family = AF_INET;
sin.sin_addr.s_addr = htonl(INADDR_ANY);
sin.sin_port = htons(MYPORT);
bind(s, (struct sockaddr *) &sin, sizeof (sin));
```

or, for the NS domain,

```
sns.sns_family = AF_NS;
netnum = htonl(net);
sns.sns_addr.x_net = *(union ns_net *) &netnum;
      /* insert net number */
sns.sns_addr.x_port = htons(MYPORT);
bind(s, (struct sockaddr *) &sns, sizeof (sns));
```

The destination address of the message to be broadcast depends on the network(s) on which the message is to be broadcast. The Internet domain supports a shorthand notation for broadcast on the local network, the address INADDR_BROADCAST (defined in *<netinet/in.h>*. To determine the list of addresses for all reachable neighbors requires knowledge of the networks to which the host is connected. Since this information should be obtained in a host-independent fashion and may be impossible to derive, 4.3BSD provides a method of retrieving this information from the system data structures. The SIOCGIFCONF *ioctl* call returns the interface configuration of a host in the form of a single *ifconf* structure; this structure contains a "data area" which is made up of an array of of *ifreq* structures, one for each network interface to which the host is connected. These structures are defined in *<net/if.h>* as follows:

```
struct ifconf {
   int   ifc_len;
         /* size of associated buffer */
   union {
      caddr_t  ifcu_buf;
      struct   ifreq *ifcu_req;
   } ifc_ifcu;
};

#define  ifc_buf  ifc_ifcu.ifcu_buf
         /* buffer address */
#define  ifc_req  ifc_ifcu.ifcu_req
         /* array of structures returned */

#define  IFNAMSIZ 16

struct ifreq {
   char  ifr_name[IFNAMSIZ];
         /* if name, e.g. "en0" */
   union {
      struct   sockaddr ifru_addr;
      struct   sockaddr ifru_dstaddr;
      struct   sockaddr ifru_broadaddr;
      short ifru_flags;
      caddr_t  ifru_data;
   } ifr_ifru;
};

      /* address */
#define  ifr_addr ifr_ifru.ifru_addr
      /* other end of p-to-p link */
#define  ifr_dstaddr ifr_ifru.ifru_dstaddr
      /* broadcast address */
#define  ifr_broadaddr  ifr_ifru.ifru_broadaddr
      /* flags */
#define  ifr_flags   ifr_ifru.ifru_flags
      /* for use by interface */
#define  ifr_data ifr_ifru.ifru_data
```

The actual call which obtains the interface configuration is

```
struct ifconf ifc;
char buf[BUFSIZ];

ifc.ifc_len = sizeof (buf);
ifc.ifc_buf = buf;
if (ioctl(s, SIOCGIFCONF, (char *) &ifc) < 0) {
   ...
}
```

After this call *buf* will contain one *ifreq* structure for each network to which the host is connected, and *ifc.ifc_len* will have been modified to reflect the number of bytes used by the *ifreq* structures.

For each structure there exists a set of "interface flags" which tell whether the network corresponding to that interface is up or down, point to point or broadcast, etc. The SIOCGIFFLAGS *ioctl* retrieves these flags for an interface specified by an *ifreq* structure as follows:

```
struct ifreq *ifr;

ifr = ifc.ifc_req;

for (n = ifc.ifc_len / sizeof (struct ifreq); --n >= 0; ifr++) {
   /*
    * We must be careful that we don't use an interface
    * devoted to an address family other than those intended;
    * if we were interested in NS interfaces, the
    * AF_INET would be AF_NS.
    */
   if (ifr->ifr_addr.sa_family != AF_INET)
      continue;
   if (ioctl(s, SIOCGIFFLAGS, (char *) ifr) < 0) {
      ...
   }
   /*
    * Skip boring cases.
    */
   if ((ifr->ifr_flags & IFF_UP) == 0 ||
       (ifr->ifr_flags & IFF_LOOPBACK) ||
       (ifr->ifr_flags & (IFF_BROADCAST | IFF_POINTTOPOINT)) == 0)
      continue;
```

Once the flags have been obtained, the broadcast address must be obtained. In the case of broadcast networks this is done via the SIOCGIFBRDADDR *ioctl*, while for point-to-point networks the address of the destination host is obtained with SIOCGIFDSTADDR.

```
      struct sockaddr dst;

      if (ifr->ifr_flags & IFF_POINTTOPOINT) {
         if (ioctl(s, SIOCGIFDSTADDR, (char *) ifr) < 0) {
            ...
         }
         bcopy((char *) ifr->ifr_dstaddr, (char *) &dst, \
               sizeof (ifr->ifr_dstaddr));
      } else if (ifr->ifr_flags & IFF_BROADCAST) {
         if (ioctl(s, SIOCGIFBRDADDR, (char *) ifr) < 0) {
            ...
         }
         bcopy((char *) ifr->ifr_broadaddr, (char *) &dst, \
               sizeof (ifr->ifr_broadaddr));
      }
```

After the appropriate *ioctl*'s have obtained the broadcast or destination address (now in *dst*), the *sendto* call may be used:

```
    sendto(s, buf, buflen, 0, (struct sockaddr *)&dst, \
          sizeof (dst));
}
```

In the above loop one *sendto* occurs for every interface to which the host is connected that supports the notion of broadcast or point-to-point addressing. If a process only wished to send broadcast messages on a given network, code similar to that outlined above would be used, but the loop would need to find the correct destination address.

Received broadcast messages contain the senders address and port, as datagram sockets are bound before a message is allowed to go out.

Socket Options

It is possible to set and get a number of options on sockets via the *setsockopt* and *getsockopt* system calls. These options include such things as marking a socket for broadcasting, not to route, to linger on close, etc. The general forms of the calls are:

```
setsockopt(s, level, optname, optval, optlen);
```

and

```
getsockopt(s, level, optname, optval, optlen);
```

The parameters to the calls are as follows: *s* is the socket on which the option is to be applied. *Level* specifies the protocol layer on which the option is to be applied; in most cases this is the "socket level", indicated by the symbolic constant SOL_SOCKET, defined in *<sys/socket.h>*. The actual option is specified in *optname*, and is a symbolic constant also defined in *<sys/socket.h>*. *Optval* and *Optlen* point to the value of the option (in most cases, whether the option is to be turned on or off), and the length of the value of the option, respectively. For *getsockopt*, *optlen* is a value-result parameter, initially set to the size of the storage area pointed to by *optval*, and modified upon return to indicate the actual amount of storage used.

An example should help clarify things. It is sometimes useful to determine the type (e.g., stream, datagram, etc.) of an existing socket; programs under *inetd* (described below) may need to perform this task. This can be accomplished as follows via the SO_TYPE socket option and the *getsockopt* call:

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

int type, size;

size = sizeof (int);

if (getsockopt(s, SOL_SOCKET, SO_TYPE, (char *) &type, &size) < 0) {
    ...
}
```

After the *getsockopt* call, *type* will be set to the value of the socket type, as defined in <*sys/socket.h*>. If, for example, the socket were a datagram socket, *type* would have the value corresponding to SOCK_DGRAM.

NS Packet Sequences

The semantics of NS connections demand that the user both be able to look inside the network header associated with any incoming packet and be able to specify what should go in certain fields of an outgoing packet. Using different calls to *setsockopt*, it is possible to indicate whether prototype headers will be associated by the user with each outgoing packet (SO_HEADERS_ON_OUTPUT), to indicate whether the headers received by the system should be delivered to the user (SO_HEADERS_ON_INPUT), or to indicate default information that should be associated with all outgoing packets on a given socket (SO_DEFAULT_HEADERS).

The contents of a SPP header (minus the IDP header) are:

```
struct sphdr {
        u_char  sp_cc;  /* connection control */
#define SP_SP   0x80    /* system packet */
#define SP_SA   0x40    /* send acknowledgement */
#define SP_OB   0x20    /* attention (out of band data) */
#define SP_EM   0x10    /* end of message */
        u_char  sp_dt;  /* datastream type */
        u_short sp_sid; /* source connection identifier */
        u_short sp_did; /* destination connection identifier */
        u_short sp_seq; /* sequence number */
v       u_short sp_ack; /* acknowledge number */
        u_short sp_alo; /* allocation number */
};
```

Here, the items of interest are the *datastream type* and the *connection control* fields. The semantics of the datastream type are defined by the application(s) in question; the value of this field is, by default, zero, but it can be used to indicate things such as Xerox's Bulk Data Transfer Protocol (in which case it is set to one). The connection control field is a mask of the flags defined just below it. The user may set or clear the end-of-message bit to indicate that a given message is the last of a given substream type, or may set/clear the attention bit as an alternate way to indicate that a packet should be sent out-of-band. As an example, to associate prototype headers with outgoing SPP packets, consider:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netns/ns.h>
#include <netns/sp.h>
 ...
struct sockaddr_ns sns, to;
int s, on = 1;
struct databuf {
   struct sphdr proto_spp; /* prototype header */
   char buf[534];     /* max. possible data by Xerox std. */
} buf;
 ...
s = socket(AF_NS, SOCK_SEQPACKET, 0);
 ...
bind(s, (struct sockaddr *) &sns, sizeof (sns));
setsockopt(s, NSPROTO_SPP, SO_HEADERS_ON_OUTPUT, &on, \
        sizeof(on));
 ...
buf.proto_spp.sp_dt = 1;    /* bulk data */
buf.proto_spp.sp_cc = SP_EM;  /* end-of-message */
strcpy(buf.buf, "hello world\n");
sendto(s, (char *) &buf, sizeof(struct sphdr) + \
        strlen("hello world\n"),
    (struct sockaddr *) &to, sizeof(to));
 ...
```

Note that one must be careful when writing headers; if the prototype header is not written with the data with which it is to be associated, the kernel will treat the first few bytes of the data as the header, with unpredictable results. To turn off the above association, and to indicate that packet headers received by the system should be passed up to the user, one might use:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netns/ns.h>
#include <netns/sp.h>
 ...
struct sockaddr sns;
int s, on = 1, off = 0;
 ...
s = socket(AF_NS, SOCK_SEQPACKET, 0);
 ...
bind(s, (struct sockaddr *) &sns, sizeof (sns));
setsockopt(s, NSPROTO_SPP, SO_HEADERS_ON_OUTPUT, &off, \
          sizeof(off));
setsockopt(s, NSPROTO_SPP, SO_HEADERS_ON_INPUT, &on, sizeof(on));
 ...
```

Output is handled somewhat differently in the IDP world. The header of an IDP-level packet looks like:

```
struct idp {
   u_short idp_sum;
      /* Checksum */
   u_short idp_len;
      /* Length, in bytes, including header */
   u_char  idp_tc;
      /* Transport Control (i.e., hop count) */
   u_char  idp_pt;
      /* Packet Type (i.e., level 2 protocol) */
   struct ns_addr idp_dna;
      /* Destination Network Address */
   struct ns_addr idp_sna;
      /* Source Network Address */
};
```

The primary field of interest in an IDP header is the *packet type* field. The standard values for this field are (as defined in *<netns/ns.h>*):

```
#define NSPROTO_RI       1      /* Routing Information */
#define NSPROTO_ECHO     2      /* Echo Protocol */
#define NSPROTO_ERROR    3      /* Error Protocol */
#define NSPROTO_PE       4      /* Packet Exchange */
#define NSPROTO_SPP      5      /* Sequenced Packet */
```

For SPP connections, the contents of this field are automatically set to NSPROTO_SPP; for IDP packets, this value defaults to zero, which means "unknown".

Setting the value of that field with SO_DEFAULT_HEADERS is easy:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netns/ns.h>
#include <netns/idp.h>
 ...
struct sockaddr sns;
struct idp proto_idp;       /* prototype header */
int s, on = 1;
 ...
s = socket(AF_NS, SOCK_DGRAM, 0);
 ...
bind(s, (struct sockaddr *) &sns, sizeof (sns));
proto_idp.idp_pt = NSPROTO_PE;   /* packet exchange */
setsockopt(s, NSPROTO_IDP, SO_DEFAULT_HEADERS, (char *) \
         &proto_idp,
    sizeof(proto_idp));
 ...
```

Using SO_HEADERS_ON_OUTPUT is somewhat more difficult. When SO_HEADERS_ON_OUTPUT is turned on for an IDP socket, the socket becomes (for all intents and purposes) a raw socket. In this case, all the fields of the prototype header (except the length and checksum fields, which are computed by the kernel) must be filled in correctly in order for the socket to send and receive data in a

sensible manner. To be more specific, the source address must be set to that of the host sending the data; the destination address must be set to that of the host for whom the data is intended; the packet type must be set to whatever value is desired; and the hopcount must be set to some reasonable value (almost always zero). It should also be noted that simply sending data using *write* will not work unless a *connect* or *sendto* call is used, in spite of the fact that it is the destination address in the prototype header that is used, not the one given in either of those calls. For almost all IDP applications , using SO_DEFAULT_HEADERS is easier and more desirable than writing headers.

Three-way Handshake

The semantics of SPP connections indicates that a three-way handshake, involving changes in the datastream type, should — but is not absolutely required to — take place before a SPP connection is closed. Almost all SPP connections are "well-behaved" in this manner; when communicating with any process, it is best to assume that the three-way handshake is required unless it is known for certain that it is not required. In a three-way close, the closing process indicates that it wishes to close the connection by sending a zero-length packet with end-of-message set and with datastream type 254. The other side of the connection indicates that it is OK to close by sending a zero-length packet with end-of-message set and datastream type 255. Finally, the closing process replies with a zero-length packet with substream type 255; at this point, the connection is considered closed. The following code fragments are simplified examples of how one might handle this three-way handshake at the user level; in the future, support for this type of close will probably be provided as part of the C library or as part of the kernel. The first code fragment below illustrates how a process might handle three-way handshake if it sees that the process it is communicating with wants to close the connection:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netns/ns.h>
#include <netns/sp.h>
 ...
#ifndef SPPSST_END
#define SPPSST_END 254
#define SPPSST_ENDREPLY 255
#endif
struct sphdr proto_sp;
int s;
 ...
read(s, buf, BUFSIZE);
if (((struct sphdr *)buf)->sp_dt == SPPSST_END) {
   /*
    * SPPSST_END indicates that the other side wants to
    * close.
    */
   proto_sp.sp_dt = SPPSST_ENDREPLY;
   proto_sp.sp_cc = SP_EM;
   setsockopt(s, NSPROTO_SPP, SO_DEFAULT_HEADERS, \
            (char *)&proto_sp,
       sizeof(proto_sp));
   write(s, buf, 0);
   /*
    * Write a zero-length packet with datastream
    * type = SPPSST_ENDREPLY to indicate that the
    * close is OK with us.  The packet that we don't
    * see (because we don't look for it) is another
    * packet from the other side of the connection,
    * with SPPSST_ENDREPLY on it, too.  Once that
    * packet is sent, the connection is considered
    * closed; note that we really ought to retransmit
    * the close for some time if we do not get a reply.
    */
   close(s);
}
 ...
```

To indicate to another process that we would like to close the connection, the following code would suffice:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netns/ns.h>
#include <netns/sp.h>
 ...
#ifndef SPPSST_END
#define SPPSST_END 254
#define SPPSST_ENDREPLY 255
#endif
struct sphdr proto_sp;
int s;
 ...
proto_sp.sp_dt = SPPSST_END;
proto_sp.sp_cc = SP_EM;
setsockopt(s, NSPROTO_SPP, SO_DEFAULT_HEADERS, (char *)&proto_sp,
    sizeof(proto_sp));
write(s, buf, 0); /* send the end request */
proto_sp.sp_dt = SPPSST_ENDREPLY;
setsockopt(s, NSPROTO_SPP, SO_DEFAULT_HEADERS, (char *)&proto_sp,
    sizeof(proto_sp));
/*
 * We assume (perhaps unwisely)
 * that the other side will send the
 * ENDREPLY, so we'll just send our final ENDREPLY
 * as if we'd seen theirs already.
 */
write(s, buf, 0);
close(s);
 ...
```

Packet Exchange

The Xerox standard protocols include a protocol that is both reliable and datagram-oriented. This protocol is known as Packet Exchange (PEX or PE) and, like SPP, is layered on top of IDP. PEX is important for a number of things: Courier remote procedure calls may be expedited through the use of PEX, and many Xerox servers are located by doing a PEX "BroadcastForServers" operation. Although there is no implementation of PEX in the kernel, it may be simulated at the user level with some clever coding and the use of one peculiar *getsockopt*. A PEX packet looks like:

```
/*
 * The packet-exchange header shown here is not defined
 * as part of any of the system include files.
 */
struct pex {
   struct idp  p_idp;    /* idp header */
   u_short  ph_id[2];    /* unique transaction ID for pex */
   u_short  ph_client;   /* client type field for pex */
};
```

The *ph_id* field is used to hold a "unique id" that is used in duplicate suppression; the *ph_client* field indicates the PEX client type (similar to the packet type field in the IDP header). PEX reliability stems from the fact that it is an idempotent ("I send a packet to you, you send a packet to me") protocol. Processes on each side of the connection may use the unique id to determine if they have seen a given packet before (the unique id field differs on each packet sent) so that duplicates may be detected, and to indicate which message a given packet is in response to. If a packet with a given unique id is sent and no response is received in a given amount of time, the packet is retransmitted until it is decided that no response will ever be received. To simulate PEX, one must be able to generate unique ids -- something that is hard to do at the user level with any real guarantee that the id is really unique. Therefore, a means (via *getsockopt*) has been provided for getting unique ids from the kernel. The following code fragment indicates how to get a unique id:

```
long uniqueid;
int s, idsize = sizeof(uniqueid);
 ...
s = socket(AF_NS, SOCK_DGRAM, 0);
 ...
/* get id from the kernel -- only on IDP sockets */
getsockopt(s, NSPROTO_PE, SO_SEQNO, (char *)&uniqueid, &idsize);
 ...
```

The retransmission and duplicate suppression code required to simulate PEX fully is left as an exercise for the reader.

Inetd

One of the daemons provided with 4.3BSD is *inetd*, the so called "internet super-server." *Inetd* is invoked at boot time, and determines from the file */etc/inetd.conf* the servers for which it is to listen. Once this information has been read and a pristine environment created, *inetd* proceeds to create one socket for each service it is to listen for, binding the appropriate port number to each socket.

Inetd then performs a *select* on all these sockets for read availability, waiting for somebody wishing a connection to the service corresponding to that socket. *Inetd* then performs an *accept* on the socket in question, *fork*s, *dup*s the new socket to file descriptors 0 and 1 (stdin and stdout), closes other open file descriptors, and *exec*s the appropriate server.

Servers making use of *inetd* are considerably simplified, as *inetd* takes care of the majority of the IPC work required in establishing a connection. The server invoked by *inetd* expects the socket connected to its client on file descriptors 0 and 1, and may immediately perform any operations such as *read*, *write*, *send*, or *recv*. Indeed, servers may use buffered I/O as provided by the "stdio" conventions, as long as as they remember to use *fflush* when appropriate.

One call which may be of interest to individuals writing servers under *inetd* is the *getpeername* call, which returns the address of the peer (process) connected on the other end of the socket. For example, to log the Internet address in "dot notation" (e.g., "128.32.0.4") of a client connected to a server under *inetd*, the following code might be used:

```
struct sockaddr_in name;
int namelen = sizeof (name);
 ...
if (getpeername(0, (struct sockaddr *)&name, &namelen) < 0) {
   syslog(LOG_ERR, "getpeername: %m");
   exit(1);
} else
   syslog(LOG_INFO, "Connection from %s", inet_ntoa(name.sin_addr));
 ...
```

While the *getpeername* call is especially useful when writing programs to run with *inetd*, it can be used under other circumstances. Be warned, however, that *getpeername* will fail on UNIX domain sockets.

Chapter 18: External Data Representation Protocol Specification

Introduction

This chapter describes library routines that allow a C programmer to describe arbitrary data structures in a machine-independent fashion. The eXternal Data Representation (XDR) standard is the backbone of the Remote Procedure Call (RPC) package, in the sense that data for remote procedure calls is transmitted using the standard. XDR library routines should be used to transmit data that is accessed (read or written) by more than one type of machine.

This manual contains a description of XDR library routines, a guide to accessing currently available XDR streams, information on defining new streams and data types, and a formal definition of the XDR standard. XDR was designed to work across different languages, operating systems, and machine architectures. Most users (particularly RPC the users) only need the information in the sections "XDR Library Primitives" and "XDR Stream Access" of this document. Programmers wishing to implement RPC and XDR on new machines will need the information in the sections "XDR Stream Implementation" and "XDR Standard". Advanced topics, not necessary for all implementations, are covered under "Advanced Topics" and "Synopsis of XDR Routines". C programs that want to use XDR routines must include the file `<rpc/rpc.h>`, which contains all the necessary interfaces to the XDR system. Since the C library `libsun.a` contains all the XDR routines, compile using this command:

```
cc  -I/usr/include/sun -I/usr/include/bsd prog.c -lsun -lbsd
```

Justification

Consider the following two programs, `writer` and `reader`:

```
#include <stdio.h>

main()              /* writer.c */
{
  long i;

  for (i = 0; i < 8; i++) {
    if (fwrite((char *)&i, sizeof(i), 1, stdout) != 1) {
      fprintf(stderr, "failed!\n");
      exit(1);
     }
  }
}
```

Figure 18-1: writer program

```
#include <stdio.h>

main()              /* reader.c */
{
  long i, j;

  for (j = 0; j < 8; j++) {
    if (fread((char *)&i, sizeof (i), 1, stdin) != 1) {
      fprintf(stderr, "failed!\n");
      exit(1);
     }
     printf("%ld ", i);
  }
  printf("\n");
 }
```

Figure 18-2: reader program

The two programs appear to be portable, because (a) they pass **lint** checking, and (b) they exhibit the same behavior when executed on two different hardware architectures, a Sun and a VAX.

Piping the output of the `writer` program to the `reader` program gives identical results on a Sun or a VAX. VAX is a trademark of Digital Equipment Corporation.

```
sun% writer | reader
0 1 2 3 4 5 6 7
sun%

vax% writer | reader
0 1 2 3 4 5 6 7
vax%
```

With the advent of local area networks and Berkeley's 4.2 BSD UNIX came the concept of "network pipes" — a process produces data on one machine, and a second process consumes data on another machine. A network pipe can be constructed with `writer` and `reader`. Here are the results if the first produces data on a Sun, and the second consumes data on a VAX.

```
sun% writer | rsh vax reader
0 16777216 33554432 50331648 67108864 83886080 \
           100663296 117440512
sun%
```

Identical results can be obtained by executing `writer` on the VAX and `reader` on the Sun. These results occur because the byte ordering of long integers differs between the VAX and the Sun, even though word size is the same. Note that 16777216 is 2^{24} — when four bytes are reversed, the 1 winds up in the 24th bit.

Whenever data is shared by two or more machine types, there is a need for portable data. Programs can be made data-portable by replacing the `read()` and `write()` calls with calls to an XDR library routine `xdr_long()` , a filter that knows the standard representation of a long integer in its external form. Here are the revised versions of `writer`:

```
#include <stdio.h>
#include <rpc/rpc.h>
/* xdr is a sub-library of the rpc library */

main()              /* writer.c */
{
    XDR xdrs;
    long i;

    xdrstdio_create(&xdrs, stdout, XDR_ENCODE);
    for (i = 0; i < 8; i++) {
      if (! xdr_long(&xdrs, &i)) {
          fprintf(stderr, "failed!\n");
          exit(1);
      }
    }
}
```

and reader:

```
#include <stdio.h>
#include <rpc/rpc.h>
/* xdr is a sub-library of the rpc library */

main()              /* reader.c */
{
    XDR xdrs;
    long i, j;

    xdrstdio_create(&xdrs, stdin, XDR_DECODE);
    for (j = 0; j < 8; j++) {
      if (! xdr_long(&xdrs, &i)) {
         fprintf(stderr, "failed!\n");
         exit(1);
      }
      printf("%ld ", i);
    }
    printf("\n");
}
```

The new programs were executed on a Sun, on a VAX, and from a Sun to a VAX; the results are shown below.

```
sun% writer | reader
0 1 2 3 4 5 6 7
sun%

vax% writer | reader
0 1 2 3 4 5 6 7
vax%

sun% writer | rsh vax reader
0 1 2 3 4 5 6 7
sun%
```

Dealing with integers is just the tip of the portable-data iceberg. Arbitrary data structures present portability problems, particularly with respect to alignment and pointers. Alignment on word boundaries may cause the size of a structure to vary from machine to machine. Pointers are convenient to use, but have no meaning outside the machine where they are defined.

The XDR Library

The XDR library package solves data portability problems. It allows you to write and read arbitrary C constructs in a consistent, specified, well-documented manner. Thus, it makes sense to use the library even when the data is not shared among machines on a network.

The XDR library has filter routines for strings (null-terminated arrays of bytes), structures, unions, and arrays, to name a few. Using more primitive routines, you can write your own specific XDR routines to describe arbitrary data structures, including elements of arrays, arms of unions, or objects pointed at from other structures. The structures themselves may contain arrays of arbitrary elements, or pointers to other structures.

Let's examine the two programs more closely. There is a family of XDR stream creation routines in which each member treats the stream of bits differently. In our example, data is manipulated using standard I/O routines, so we use `xdrstdio_create()`. The parameters to XDR stream creation routines vary according to their function. In our example, `xdrstdio_create()` takes a pointer to an XDR structure that it initializes, a pointer to a FILE that the input or output is performed on, and the operation. The operation may be XDR_ENCODE for serializing in the `writer` program, or XDR_DECODE for deserializing in the `reader` program.

Note: RPC clients never need to create XDR streams; the RPC system itself creates these streams, which are then passed to the clients.

The `xdr_long()` primitive is characteristic of most XDR library primitives and all client XDR routines. First, the routine returns FALSE (0) if it fails, and TRUE (1) if it succeeds. Second, for each data type, `xxx`, there is an associated XDR routine of the form:

```
xdr_xxx(xdrs, fp)
        XDR *xdrs;
        xxx *fp;
{
}
```

In our case, `xxx` is long, and the corresponding XDR routine is a primitive, `xdr_long`. The client could also define an arbitrary structure `xxx` in which case the client would also supply the routine `xdr_xxx`, describing each field by calling XDR routines of the appropriate type. In all cases the first parameter, `xdrs` can be treated as an opaque handle, and passed to the primitive routines.

XDR routines are direction independent; that is, the same routines are called to serialize or deserialize data. This feature is critical to software engineering of portable data. The idea is to call the same routine for either operation — this almost guarantees that serialized data can also be deserialized. One routine is used by both producer and consumer of networked data. This is implemented by always passing the address of an object rather than the object itself — only in the case of

deserialization is the object modified. This feature is not shown in our trivial example, but its value becomes obvious when nontrivial data structures are passed among machines. If needed, you can obtain the direction of the XDR operation. See "XDR Operation Directions".

Let's look at a slightly more complicated example. Assume that a person's gross assets and liabilities are to be exchanged among processes. Also assume that these values are important enough to warrant their own data type:

```
struct gnumbers {
    long g_assets;
    long g_liabilities;
};
```

The corresponding XDR routine describing this structure would be:

```
bool_t
   /* TRUE is success, FALSE is failure */
xdr_gnumbers(xdrs, gp)
     XDR *xdrs;
     struct gnumbers *gp;
{
     if (xdr_long(xdrs, &gp->g_assets) &&
        xdr_long(xdrs, &gp->g_liabilities))
           return(TRUE);
      return(FALSE);
}
```

Note that the parameter `xdrs` is never inspected or modified; it is only passed on to the subcomponent routines. It is imperative to inspect the return value of each XDR routine call, and to give up immediately and return FALSE if the subroutine fails.

This example also shows that the type `bool_t` is declared as an integer whose only values are TRUE (1) and FALSE (0). This document uses the following definitions:

```
#define bool_t    int
#define TRUE      1
#define FALSE     0

#define enum_t int
/* enum_t's are used for generic enum's */
```

Keeping these conventions in mind, `xdr_gnumbers()` can be rewritten as follows:

```
xdr_gnumbers(xdrs, gp)
   XDR *xdrs;
   struct gnumbers *gp;
{
   return (xdr_long(xdrs, &gp->g_assets) &&
      xdr_long(xdrs, &gp->g_liabilities));
}
```

This document uses both coding styles.

XDR Library Primitives

This section gives a synopsis of each XDR primitive. It starts with basic data types and moves on to constructed data types. Finally, XDR utilities are discussed. The interface to these primitives and utilities is defined in the include file `<rpc/xdr.h>`, automatically included by `<rpc/rpc.h>`.

Number Filters

The XDR library provides primitives that translate between C numbers and their corresponding external representations. The primitives cover the set of numbers in:

*[signed, unsigned] * [short, int, long]*

Specifically, the six primitives are:

```
bool_t xdr_int(xdrs, ip)
      XDR *xdrs;
      int *ip;

bool_t xdr_u_int(xdrs, up)
      XDR *xdrs;
      unsigned *up;

bool_t xdr_long(xdrs, lip)
      XDR *xdrs;
      long *lip;

bool_t xdr_u_long(xdrs, lup)
      XDR *xdrs;
      u_long *lup;

bool_t xdr_short(xdrs, sip)
      XDR *xdrs;
      short *sip;

bool_t xdr_u_short(xdrs, sup)
      XDR *xdrs;
      u_short *sup;
```

The first parameter, `xdrs`, is an XDR stream handle. The second parameter is the address of the number that provides data to the stream or receives data from it. All routines return TRUE if they complete successfully, and FALSE otherwise.

Floating Point Filters

The XDR library also provides primitive routines for C's floating point types:

```
bool_t xdr_float(xdrs, fp)
      XDR *xdrs;
      float *fp;
```

```
bool_t xdr_double(xdrs, dp)
      XDR *xdrs;
      double *dp;
```

The first parameter, `xdrs` is an XDR stream handle. The second parameter is the address of the floating point number that provides data to the stream or receives data from it. All routines return TRUE if they complete successfully, and FALSE otherwise.

Note: Since the numbers are represented in IEEE floating point, routines may fail when decoding a valid IEEE representation into a machine-specific representation, or vice-versa.

Enumeration Filters

The XDR library provides a primitive for generic enumerations. The primitive assumes that a C `enum` has the same representation inside the machine as a C integer. The boolean type is an important instance of the `enum`. The external representation of a boolean is always one (TRUE) or zero (FALSE).

```
#define bool_t    int
#define FALSE     0
#define TRUE      1

#define enum_t int

bool_t xdr_enum(xdrs, ep)
      XDR *xdrs;
      enum_t *ep;

bool_t xdr_bool(xdrs, bp)
      XDR *xdrs;
      bool_t *bp;
```

The second parameters `ep` and `bp` are addresses of the associated type that provides data to, or receives data from, the stream `xdrs`. The routines return TRUE if they complete successfully, and FALSE otherwise.

No Data

Occasionally, an XDR routine must be supplied to the RPC system, even when no data is passed or required. The library provides such a routine:

```
bool_t xdr_void();  /* always returns TRUE */
```

Constructed Data Type Filters

Constructed or compound data type primitives require more parameters and perform more complicated functions then the primitives discussed above. This section includes primitives for strings, arrays, unions, and pointers to structures.

Constructed data type primitives may use memory management. In many cases, memory is allocated when deserializing data with XDR_DECODE. Therefore, the XDR package must provide means to deallocate memory. This is done by an XDR operation, XDR_FREE. To review, the three XDR directional operations are XDR_ENCODE, XDR_DECODE, and XDR_FREE.

Strings

In C, a string is defined as a sequence of bytes terminated by a null byte, which is not considered when calculating string length. However, when a string is passed or manipulated, a pointer to it is employed. Therefore, the XDR library defines a string to be a `char *`, and not a sequence of characters. The external representation of a string is drastically different from its internal representation. Externally, strings are represented as sequences of ASCII characters, while internally, they are represented with character pointers. Conversion between the two representations is accomplished with the routine `xdr_string()`:

```
bool_t xdr_string(xdrs, sp, maxlength)
        XDR *xdrs;
        char **sp;
        u_int maxlength;
```

The first parameter `xdrs` is the XDR stream handle. The second parameter `sp` is a pointer to a string (type `char **`). The third parameter `maxlength` specifies the maximum number of bytes allowed during encoding or decoding; its value is usually specified by a protocol. For example, a protocol specification may say that a file name may be no longer than 255 characters. The routine returns FALSE if the number of characters exceeds `maxlength`, and TRUE if it doesn't.

The behavior of `xdr_string()` is similar to the behavior of other routines discussed in this section. The direction XDR_ENCODE is easiest to understand. The parameter `sp` points to a string of a certain length; if it does not exceed `maxlength`, the bytes are serialized.

The effect of deserializing a string is subtle. First the length of the incoming string is determined; it must not exceed `maxlength`. Next `sp` is dereferenced; if the the value is NULL, then a string of the appropriate length is allocated and `*sp` is set to this string. If the original value of `*sp` is non-NULL, then the XDR package assumes that a target area has been allocated, which can hold strings no longer than `maxlength`. In either case, the string is decoded into the target area. The routine then appends a null character to the string.

In the XDR_FREE operation, the string is obtained by dereferencing `sp`. If the string is not NULL, it is freed and `*sp` is set to NULL. In this operation, `xdr_string` ignores the `maxlength` parameter.

Byte Arrays

Often variable-length arrays of bytes are preferable to strings. Byte arrays differ from strings in the following three ways: 1) the length of the array (the byte count) is explicitly located in an unsigned integer, 2) the byte sequence is not terminated by a null character, and 3) the external representation of the bytes is the same as their internal representation. The primitive `xdr_bytes()` converts between the internal and external representations of byte arrays:

```
bool_t xdr_bytes(xdrs, bpp, lp, maxlength)
      XDR *xdrs;
      char **bpp;
      u_int *lp;
      u_int maxlength;
```

The usage of the first, second and fourth parameters are identical to the first, second and third parameters of `xdr_string()`, respectively. The length of the byte area is obtained by dereferencing `lp` when serializing; `*lp` is set to the byte length when deserializing.

Arrays

The XDR library package provides a primitive for handling arrays of arbitrary elements. The `xdr_bytes()` routine treats a subset of generic arrays, in which the size of array elements is known to be 1, and the external description of each element is built-in. The generic array primitive, `xdr_array()` requires parameters identical to those of `xdr_bytes()` plus two more: the size of array elements, and an XDR routine to handle each of the elements. This routine is called to encode or decode each element of the array.

```
bool_t xdr_array(xdrs, ap, lp, maxlength, elementsize,
   xdr_element)
      XDR *xdrs;
      char **ap;
      u_int *lp;
      u_int maxlength;
      u_int elementsize;
      bool_t (*xdr_element)();
```

The parameter `ap` is the address of the pointer to the array. If `*ap` is NULL when the array is being deserialized, XDR allocates an array of the appropriate size and sets `*ap` to that array. The element count of the array is obtained from `*lp` when the array is serialized; `*lp` is set to the array length when the array is deserialized. The parameter `maxlength` is the maximum number of elements that the array is allowed to have; `elementsize` is the byte size of each element of the array (the C function `sizeof()` can be used to obtain this value). The routine `xdr_element` is called to serialize, deserialize, or free each element of the array.

Examples

Before defining more constructed data types, it is appropriate to present three examples.

Example A

A user on a networked machine can be identified by (a) the machine name, such as `krypton`: see **gethostname**(2); (b) the user's UID: see **geteuid**(2); and (c) the group numbers to which the user belongs: see **getegid**(2). A structure with this information and its associated XDR routine could be coded like this:

```
struct netuser {
      char  *nu_machinename;
      int   nu_uid;
      u_int nu_glen;
      int   *nu_gids;
};
#define NLEN 255
/* machine names must be shorter than 256 chars */
#define NGRPS 20
/* user can't be a member of more than 20 groups */

bool_t
xdr_netuser(xdrs, nup)
    XDR *xdrs;
    struct netuser *nup;
{
    return (xdr_string(xdrs, &nup->nu_machinename, NLEN) &&
       xdr_int(xdrs, &nup->nu_uid) &&
       xdr_array(xdrs, &nup->nu_gids, &nup->nu_glen, NGRPS,
          sizeof (int), xdr_int));
}
```

Example B

A party of network users could be implemented as an array of `netuser` structure. The declaration and its associated XDR routines are as follows:

```
struct party {
   u_int p_len;
   struct netuser *p_nusers;
};
#define PLEN 500 /* max number of users in a party */
bool_t
xdr_party(xdrs, pp)
   XDR *xdrs;
   struct party *pp;
{
   return (xdr_array(xdrs, &pp->p_nusers, &pp->p_len, PLEN,
          sizeof (struct netuser), xdr_netuser));
}
```

Example C

The well-known parameters to `main()`, `argc` and `argv` can be combined into a structure. An array of these structures can make up a history of commands. The declarations and XDR routines might look like:

```
struct cmd {
    u_int c_argc;
    char **c_argv;
}; /* args can be no longer than 1000 chars */
#define ALEN 1000
#define NARGC 100
   /* commands may have no more than 100 args */
```

```
struct history {
    u_int h_len;
    struct cmd *h_cmds;
};
#define NCMDS 75
/* history is no more than 75 commands */

bool_t
xdr_wrap_string(xdrs, sp)
     XDR *xdrs;
     char **sp;
{
     return (xdr_string(xdrs, sp, ALEN));
}

bool_t
xdr_cmd(xdrs, cp)
     XDR *xdrs;
     struct cmd *cp;
{
     return (xdr_array(xdrs, &cp->c_argv, &cp->c_argc, NARGC,
        sizeof (char *), xdr_wrap_string));
}

bool_t
xdr_history(xdrs, hp)
    XDR *xdrs;
      struct history *hp;
{
    return (xdr_array(xdrs, &hp->h_cmds, &hp->h_len, NCMDS,
        sizeof (struct cmd), xdr_cmd));
}
```

The most confusing part of this example is that the routine `xdr_wrap_string()` is needed to package the `xdr_string()` routine, because the implementation of `xdr_array()` only passes two parameters to the array element description routine; `xdr_wrap_string()` supplies the third parameter to `xdr_string()`.

By now the recursive nature of the XDR library should be obvious. Let's continue with more constructed data types.

Opaque Data

In some protocols, handles are passed from a server to client. The client passes the handle back to the server at some later time. Handles are never inspected by clients; they are obtained and submitted. That is to say, handles are opaque. The primitive `xdr_opaque()` is used for describing fixed sized, opaque bytes.

```
bool_t xdr_opaque(xdrs, p, len)
      XDR *xdrs;
      char *p;
      u_int len;
```

The parameter `p` is the location of the bytes; `len` is the number of bytes in the opaque object. By definition, the actual data contained in the opaque object are not machine portable.

Fixed Sized Arrays

The XDR library does not provide a primitive for fixed-length arrays (the primitive `xdr_array()` is for varying-length arrays). Example A could be rewritten to use fixed-sized arrays in the following fashion:

```
#define NLEN 255
/* machine names must be shorter than 256 chars */
#define NGRPS 20
/* user cannot be a member of more than 20 groups */

struct netuser {
    char *nu_machinename;
    int nu_uid;
    int nu_gids[NGRPS];
};

bool_t
xdr_netuser(xdrs, nup)
XDR *xdrs;
struct netuser *nup;
{
    int i;

    if (! xdr_string(xdrs, &nup->nu_machinename, NLEN))
            return (FALSE);
    if (! xdr_int(xdrs, &nup->nu_uid))
            return (FALSE);
    for (i = 0; i < NGRPS; i++) {
            if (! xdr_int(xdrs, &nup->nu_gids[i]))
            return (FALSE);
    }
    return (TRUE);
}
```

Exercise: Rewrite Example A so that it uses varying-length arrays and so that the netuser structure contains the actual `nu_gids` array body as in the example above.

Discriminated Unions

The XDR library supports discriminated unions. A discriminated union is a C union and an `enum_t` value that selects an "arm" of the union.

```
struct xdr_discrim {
      enum_t value;
      bool_t (*proc)();
};

bool_t xdr_union(xdrs, dscmp, unp, arms, defaultarm)
      XDR *xdrs;
      enum_t *dscmp;
      char *unp;
      struct xdr_discrim *arms;
      bool_t (*defaultarm)();  /* may equal NULL */
```

First the routine translates the discriminant of the union located at `*dscmp`. The

discriminant is always an `enum_t`. Next the union located at `*unp` is translated. The parameter `arms` is a pointer to an array of `xdr_discrim` structures. Each structure contains an order pair of `[value,proc]`. If the union's discriminant is equal to the associated `value`, then the `proc` is called to translate the union. The end of the `xdr_discrim` structure array is denoted by a routine of value NULL (0). If the discriminant is not found in the `arms` array, then the `defaultarm` procedure is called if it is non-NULL; otherwise the routine returns FALSE.

Example D

Suppose the type of a union may be integer, character pointer (a string), or a `gnumbers` structure. Also, assume the union and its current type are declared in a structure. The declaration is:

```
enum utype { INTEGER=1, STRING=2, GNUMBERS=3 };

struct u_tag {
      enum utype utype;
         /* this is the union's discriminant */
      union {
            int ival;
            char *pval;
            struct gnumbers gn;
      } uval;
};
```

The following constructs and XDR procedure (de)serialize the discriminated union:

```
struct xdr_discrim u_tag_arms[4] = {
    { INTEGER, xdr_int },
    { GNUMBERS, xdr_gnumbers }
    { STRING, xdr_wrap_string },
    { __dontcare__, NULL }
       /* always terminate arms with a NULL xdr_proc */
}

bool_t
xdr_u_tag(xdrs, utp)
    XDR *xdrs;
    struct u_tag *utp;
{
    return (xdr_union(xdrs, &utp->utype, &utp->uval, u_tag_arms,
          NULL));
}
```

The routine `xdr_gnumbers()` was presented in the section "The XDR Library"; `xdr_wrap_string()` was presented in example C. The default arm parameter to `xdr_union()` (the last parameter) is NULL in this example. Therefore the value of the union's discriminant legally may take on only values listed in the `u_tag_arms` array. This example also demonstrates that the elements of the arm's array do not need to be sorted.

It is worth pointing out that the values of the discriminant may be sparse, though in this example they are not. It is always good practice to assign explicitly integer values to each element of the discriminant's type. This practice both documents the external representation of the discriminant and guarantees that different C compilers emit identical discriminant values.

Exercise: Implement `xdr_union()` using the other primitives in this section.

Pointers

In C it is often convenient to put pointers to another structure within a structure. The primitive `xdr_reference()` makes it easy to serialize, deserialize, and free these referenced structures.

```
bool_t xdr_reference(xdrs, pp, size, proc)
      XDR *xdrs;
      char **pp;
      u_int ssize;
      bool_t (*proc)();
```

Parameter `pp` is the address of the pointer to the structure; parameter `ssize` is the size in bytes of the structure (use the C function `sizeof()` to obtain this value); and `proc` is the XDR routine that describes the structure. When decoding data, storage is allocated if `*pp` is NULL.

There is no need for a primitive `xdr_struct()` to describe structures within structures, because pointers are always sufficient.

Exercise: Implement `xdr_reference()` using `xdr_array()`. Warning: `xdr_reference()` and `xdr_array()` are NOT interchangeable external representations of data.

Example E

Suppose there is a structure containing a person's name and a pointer to a `gnumbers` structure containing the person's gross assets and liabilities. The construct is:

```
struct pgn {
    char *name;
    struct gnumbers *gnp;
};
```

The corresponding XDR routine for this structure is:

```
bool_t
xdr_pgn(xdrs, pp)
        XDR *xdrs;
        struct pgn *pp;
{
    if (xdr_string(xdrs, &pp->name, NLEN) &&
        xdr_reference(xdrs, &pp->gnp, sizeof(struct gnumbers),
        xdr_gnumbers))
      return(TRUE);
    return(FALSE);
}
```

Pointer Semantics and XDR

In many applications, C programmers attach double meaning to the values of a pointer. Typically the value NULL (or zero) means data is not needed, yet some application-specific interpretation applies. In essence, the C programmer is encoding a discriminated union efficiently by overloading the interpretation of the value of a pointer. For instance, in example E a NULL pointer value for `gnp` could indicate

that the person's assets and liabilities are unknown. That is, the pointer value encodes two things: whether or not the data is known; and if it is known, where it is located in memory. Linked lists are an extreme example of the use of application-specific pointer interpretation.

The primitive `xdr_reference()` cannot and does not attach any special meaning to a NULL-value pointer during serialization. That is, passing an address of a pointer whose value is NULL to `xdr_reference()` when serialing data will most likely cause a memory fault and, on UNIX, a core dump for debugging.

It is the explicit responsibility of the programmer to expand non-dereferenceable pointers into their specific semantics. This usually involves describing data with a two-armed discriminated union. One arm is used when the pointer is valid; the other is used when the pointer is invalid (NULL). The section "Synopsis of XDR Routines" has an example (linked lists encoding) that deals with invalid pointer interpretation.

Exercise: After reading the section "Synopsis of XDR Routines", return here and extend example E so that it can correctly deal with null pointer values.

Exercise: Using the `xdr_union()`, `xdr_reference()` and `xdr_void()` primitives, implement a generic pointer handling primitive that implicitly deals with NULL pointers. The XDR library does not provide such a primitive because it does not want to give the illusion that pointers have meaning in the external world.

Non-filter Primitive

XDR streams can be manipulated with the primitives discussed in this section.

```
u_int xdr_getpos(xdrs)
      XDR *xdrs;

bool_t xdr_setpos(xdrs, pos)
      XDR *xdrs;
      u_int pos;

xdr_destroy(xdrs)
      XDR *xdrs;
```

The routine `xdr_getpos()` returns an unsigned integer that describes the current position in the data stream. Warning: In some XDR streams, the returned value of `xdr_getpos()` is meaningless; the routine returns a −1 in this case (though −1 should be a legitimate value).

The routine `xdr_setpos()` sets a stream position to `pos`. Warning: In some XDR streams, setting a position is impossible; in such cases, `xdr_setpos()` will return FALSE. This routine will also fail if the requested position is out-of-bounds. The definition of bounds varies from stream to stream.

The `xdr_destroy()` primitive destroys the XDR stream. Usage of the stream after calling this routine is undefined.

XDR Operation Directions

At times you may wish to optimize XDR routines by taking advantage of the direction of the operation (XDR_ENCODE, XDR_DECODE, or XDR_FREE). The value `xdrs->x_op` always contains the direction of the XDR operation. Programmers are not encouraged to take advantage of this information. Therefore, no example is presented here. However, an example in the section "Synopsis of XDR Routines" demonstrates the usefulness of the `xdrs->x_op` field.

XDR Stream Access

An XDR stream is obtained by calling the appropriate creation routine. These creation routines take arguments that are tailored to the specific properties of the stream.

Streams currently exist for (de)serialization of data to or from standard I/O FILE streams, TCP/IP connections and UNIX files, and memory. The section "XDR Stream Implementation" documents the XDR object and how to make new XDR streams when they are required.

Standard I/O Streams

XDR streams can be interfaced to standard I/O using the `xdrstdio_create()` routine as follows:

```
#include <stdio.h>
#include <rpc/rpc.h>
/* xdr streams are a part of the rpc library */

void
xdrstdio_create(xdrs, fp, x_op)
        XDR *xdrs;
        FILE *fp;
        enum xdr_op x_op;
```

The routine `xdrstdio_create()` initializes an XDR stream pointed to by `xdrs`. The XDR stream interfaces to the standard I/O library. Parameter `fp` is an open file, and `x_op` is an XDR direction.

Memory Streams

Memory streams allow the streaming of data into or out of a specified area of memory:

```
#include <rpc/rpc.h>

void
xdrmem_create(xdrs, addr, len, x_op)
        XDR *xdrs;
        char *addr;
        u_int len;
        enum xdr_op x_op;
```

The routine `xdrmem_create()` initializes an XDR stream in local memory. The memory is pointed to by parameter `addr`; parameter `len` is the length in bytes of the memory. The parameters `xdrs` and `x_op` are identical to the corresponding parameters of `xdrstdio_create()`. Currently, the UDP/IP implementation of RPC uses `xdrmem_create()`. Complete call or result messages are built in memory before calling the `sendto()` system routine.

Record (TCP/IP) Streams

A record stream is an XDR stream built on top of a record marking standard that is built on top of the UNIX file or 4.2 BSD connection interface.

```
#include <rpc/rpc.h>
    /* xdr streams are a part of the rpc library */

xdrrec_create(xdrs, sendsize, recvsize, iohandle, readproc,
   writeproc)
      XDR *xdrs;
      u_int sendsize, recvsize;
      char *iohandle;
      int (*readproc)(), (*writeproc)();
```

The routine `xdrrec_create()` provides an XDR stream interface that allows for a bidirectional, arbitrarily long sequence of records. The contents of the records are meant to be data in XDR form. The stream's primary use is for interfacing RPC to TCP connections. However, it can be used to stream data into or out of normal UNIX files.

The parameter `xdrs` is similar to the corresponding parameter described above. The stream does its own data buffering similar to that of standard I/O. The parameters `sendsize` and `recvsize` determine the size in bytes of the output and input buffers, respectively; if their values are zero (0), then predetermined defaults are used. When a buffer needs to be filled or flushed, the routine `readproc` or `writeproc` is called, respectively. The usage and behavior of these routines are similar to the UNIX system calls `read()` and `write()`. However, the first parameter to each of these routines is the opaque parameter `iohandle`. The other two parameters `buf` (and `nbytes`) and the results (byte count) are identical to the system routines. If `xxx` is `readproc` or `writeproc`, then it has the following form:

```
/*
/* returns the actual number of bytes transferred.
 * -1 is an error
 */
int
xxx(iohandle, buf, len)
      char *iohandle;
      char *buf;
      int nbytes;
```

The XDR stream provides means for delimiting records in the byte stream. The implementation details of delimiting records in a stream are discussed in "Synopsis of XDR Routines". The primitives that are specific to record streams are as follows:

```
bool_t
xdrrec_endofrecord(xdrs, flushnow)
      XDR *xdrs;
      bool_t flushnow;

bool_t
xdrrec_skiprecord(xdrs)
      XDR *xdrs;

bool_t
xdrrec_eof(xdrs)
      XDR *xdrs;
```

The routine `xdrrec_endofrecord()` causes the current outgoing data to be marked as a record. If the parameter `flushnow` is TRUE, then the stream's `writeproc()` will be called; otherwise, `writeproc()` will be called when the output buffer has been filled.

The routine `xdrrec_skiprecord()` causes an input stream's position to be moved past the current record boundary and onto the beginning of the next record in the stream.

If there is no more data in the stream's input buffer, then the routine `xdrrec_eof()` returns TRUE. That is not to say that there is no more data in the underlying file descriptor.

XDR Stream Implementation

This section provides the abstract data types needed to implement new instances of XDR streams.

The XDR Object

The following structure defines the interface to an XDR stream:

```
enum xdr_op { XDR_ENCODE = 0, XDR_DECODE = 1, XDR_FREE = 2 };

typedef struct {
      enum xdr_op x_op;
         /* operation; fast additional param */
      struct xdr_ops {
          bool_t  (*x_getlong)();
            /* get a long from underlying stream */
          bool_t  (*x_putlong)();
            /* put a long to " */
          bool_t  (*x_getbytes)();
            /* get some bytes from " */
          bool_t  (*x_putbytes)();
            /* put some bytes to " */
          u_int   (*x_getpostn)();
            /* returns byte offset from beginning */
          bool_t  (*x_setpostn)();
             /* repositions position in stream */
          caddr_t (*x_inline)();
             /* buf quick ptr to buffered data */
          VOID    (*x_destroy)();
             /* free privates of this xdr_stream */
      } *x_ops;
      caddr_t           x_public;
          /* users' data */
      caddr_t           x_private;
          /* pointer to private data */
      caddr_t           x_base;
          /* private used for position info */
      int         x_handy;
          /* extra private word */
} XDR;
```

The `x_op` field is the current operation being performed on the stream. This field is important to the XDR primitives, but should not affect a stream's implementation. That is, a stream's implementation should not depend on this value. The fields `x_private`, `x_base` , and `x_handy` are private to the particular stream's implementation. The field `x_public` is for the XDR client and should never be used by the XDR stream implementations or the XDR primitives.

Macros for accessing operations `x_getpostn()`, `x_setpostn()`, and `x_destroy()` were defined in "The XDR Object". The operation `x_inline()` takes two parameters: an XDR *, and an unsigned integer, which is a byte count. The routine returns a pointer to a piece of the stream's internal buffer. The caller can then use the buffer segment for any purpose. From the stream's point of view, the bytes in the buffer segment have been consumed or put. The routine may return NULL if it cannot return a buffer segment of the requested size. (The `x_inline`

routine is for cycle squeezers. Use of the resulting buffer is not data-portable. Users are encouraged not to use this feature.)

The operations `x_getbytes()` and `x_putbytes()` blindly get and put sequences of bytes from or to the underlying stream; they return TRUE if they are successful, and FALSE otherwise. The routines have identical parameters (replace xxx):

```
bool_t
xxxbytes(xdrs, buf, bytecount)
      XDR *xdrs;
      char *buf;
      u_int bytecount;
```

The operations `x_getlong()` and `x_putlong()` receive and put long numbers from and to the data stream. It is the responsibility of these routines to translate the numbers between the machine representation and the (standard) external representation. The UNIX primitives `htonl()` and `ntohl()` can be helpful in accomplishing this. "Advanced Topics" defines the standard representation of numbers. The higher-level XDR implementation assumes that signed and unsigned long integers contain the same number of bits, and that nonnegative integers have the same bit representations as unsigned integers. The routines return TRUE if they succeed, and FALSE otherwise. They have identical parameters:

```
bool_t
xxxlong(xdrs, lp)
      XDR *xdrs;
      long *lp;
```

Implementors of new XDR streams must make an XDR structure (with new operation routines) available to clients, using some kind of create routine.

XDR Standard

This section defines the external data representation standard. The standard is independent of languages, operating systems and hardware architectures. Once data is shared among machines, it should not matter that the data was produced on a Sun, but is consumed by a VAX (or vice versa). Similarly the choice of operating systems should have no influence on how the data is represented externally. For programming languages, data produced by a C program should be readable by a FORTRAN or Pascal program.

The external data representation standard depends on the assumption that bytes (or octets) are portable. A byte is defined to be eight bits of data. It is assumed that hardware that encodes bytes onto various media will preserve the bytes' meanings across hardware boundaries. For example, the Ethernet standard suggests that bytes be encoded "little endian" style. Both Sun and VAX hardware implementations adhere to the standard.

The XDR standard also suggests a language used to describe data. The language is a bastardized C; it is a data description language, not a programming language. (The Xerox Courier Standard uses bastardized Mesa as its data description language.)

Basic Block Size

The representation of all items requires a multiple of four bytes (or 32 bits) of data. The bytes are numbered `0` through `n-1` where `(n mod 4) = 0`. The bytes are read or written to some byte stream such that byte `m` always precedes byte `m+1`.

Integer

An XDR signed integer is a 32-bit datum that encodes an integer in the range `[-2147483648,2147483647]`. The integer is represented in two's complement notation. The most and least significant bytes are 0 and 3, respectively. The data description of integers is `integer`.

Unsigned Integer

An XDR unsigned integer is a 32-bit datum that encodes a nonnegative integer in the range `[0,4294967295]`. It is represented by an unsigned binary number whose most and least significant bytes are 0 and 3, respectively. The data description of unsigned integers is `unsigned`.

Enumerations

Enumerations have the same representation as integers. Enumerations are handy for describing subsets of the integers. The data description of enumerated data is as follows:

```
typedef enum { name = value, .... } type-name;
```

For example the three colors red, yellow and blue could be described by an enumerated type:

```
typedef enum { RED = 2, YELLOW = 3, BLUE = 5 } colors;
```

Booleans

Booleans are important enough and occur frequently enough to warrant their own explicit type in the standard. Boolean is an enumeration with the following form:

```
typedef enum { FALSE = 0, TRUE = 1 } boolean;
```

Hyper Integer and Hyper Unsigned

The standard also defines 64-bit (8-byte) numbers called `hyper integer` and `hyper unsigned`. Their representations are the obvious extensions of the integer and unsigned defined above. The most and least significant bytes are 0 and 7, respectively.

Floating Point and Double Precision

The standard defines the encoding for the floating point data types `float` (32 bits or 4 bytes) and `double` (64 bits or 8 bytes). The encoding used is the IEEE standard for normalized single- and double-precision floating point numbers. See the IEEE floating point standard for more information. The standard encodes the following three fields, which describe the floating point number:

S The sign of the number. Values 0 and 1 represent positive and negative, respectively.

E The exponent of the number, base 2. Floats devote 8 bits to this field, while doubles devote 11 bits. The exponents for float and double are biased by 127 and 1023, respectively.

F The fractional part of the number's mantissa, base 2. Floats devote 23 bits to this field, while doubles devote 52 bits.

Therefore, the floating point number is described by:

$$(-1)^S * 2^{(E - Bias)} * 1.F$$

Just as the most and least significant bytes of a number are 0 and 3, the most and least significant bits of a single-precision floating point number are 0 and 31. The beginning bit (and most significant bit) offsets of *S*, *E*, and *F* are 0, 1, and 9, respectively.

Doubles have the analogous extensions. The beginning bit (and most significant bit) offsets of *S*, *E*, and *F* are 0, 1, and 12, respectively.

The IEEE specification should be consulted concerning the encoding for signed zero, signed infinity (overflow), and denormalized numbers (underflow). Under IEEE specifications, the "NaN" (not a number) is system dependent and should not be used.

Opaque Data

At times fixed-sized uninterpreted data needs to be passed among machines. This data is called `opaque` and is described as:

```
typedef opaque type-name[n];
opaque name[n];
```

where `n` is the (static) number of bytes necessary to contain the opaque data. If `n` is not a multiple of four, then the `n` bytes are followed by enough (up to 3) zero-valued bytes to make the total byte count of the opaque object a multiple of four.

Counted Byte Strings

The standard defines a string of n (numbered 0 through n-1 bytes to be the number n encoded as unsigned, and followed by the n bytes of the string. If n is not a multiple of four, then the n bytes are followed by enough (up to 3) zero-valued bytes to make the total byte count a multiple of four. The data description of strings is as follows:

```
typedef string type-name<N>;
typedef string type-name<>;
string name<N>;
string name<>;
```

Note that the data description language uses angle brackets (< and >) to denote anything that is varying-length (as opposed to square brackets to denote fixed-length sequences of data).

The constant N denotes an upper bound of the number of bytes that a string may contain. If N is not specified, it is assumed to be $2^{32} - 1$ the maximum length. The constant N would normally be found in a protocol specification. For example, a filing protocol may state that a file name can be no longer than 255 bytes, such as:

```
string filename<255>;
```

The XDR specification does not say what the individual bytes of a string represent; this important information is left to higher-level specifications. A reasonable default is to assume that the bytes encode ASCII characters.

Fixed Arrays

The data description for fixed-size arrays of homogeneous elements is as follows:

```
typedef elementtype type-name[n];
elementtype name[n]
```

Fixed-size arrays of elements numbered 0 through n-1 are encoded by individually encoding the elements of the array in their natural order, 0 through n-1.

Counted Arrays

Counted arrays provide the ability to encode varyiable-length arrays of homogeneous elements. The array is encoded as: the element count n (an unsigned integer), followed by the encoding of each of the array's elements, starting with element 0 and progressing through element n-1. The data description for counted arrays is similar to that of counted strings:

```
typedef elementtype type-name<N>;
typedef elementtype type-name<>;
elementtype name<N>;
elementtype name<>;
```

Again, the constant N specifies the maximum acceptable element count of an array; if N is not specified, it is assumed to be $2^{32} - 1$.

Structures

The data description for structures is very similar to that of standard C:

```
typedef struct {
        component-type component-name;
        ...
} type-name;
```

The components of the structure are encoded in the order of their declaration in the structure.

Discriminated Unions

A discriminated union is a type composed of a discriminant followed by a type selected from a set of prearranged types according to the value of the discriminant. The type of the discriminant is always an enumeration. The component types are called "arms" of the union. The discriminated union is encoded as its discriminant followed by the encoding of the implied arm. The data description for discriminated unions is as follows:

```
typedef union switch (discriminant-type) {
        discriminant-value: arm-type;
        ...
        default: default-arm-type;
} type-name;
```

The default arm is optional. If it is not specified, then a valid encoding of the union cannot take on unspecified discriminant values. Most specifications neither need nor use default arms.

Missing Specifications

The standard lacks representations for bit fields and bitmaps, since the standard is based on bytes. This is not to say that no specification should be attempted.

Library Primitive/XDR Standard Cross Reference

The following table describes the association between the C library primitives discussed in "XDR Stream Access", and the standard data types defined in this section.

C Primitive	XDR Type	Sections
xdr_int xdr_long xdr_short	integer	3.1, 6.2
xdr_u_int xdr_u_long xdr_u_short	unsigned	3.1, 6.3
-	hyper integer hyper unsigned	6.6
xdr_float	float	3.2, 6.7
xdr_double	double	3.2, 6.7
xdr_enum	enum_t	3.3, 6.4
xdr_bool	bool_t	3.3, 6.5
xdr_string xdr_bytes	string	3.5.1, 6.9 3.5.2
xdr_array	(varying arrays)	3.5.3, 6.11
-	(fixed arrays)	3.5.5, 6.10
xdr_opaque	opaque	3.5.4, 6.8
xdr_union	union	3.5.6, 6.13
xdr_reference	-	3.5.7
-	struct	6.6

Advanced Topics

This section describes techniques for passing data structures that are not covered in the preceding sections. Such structures include linked lists (of arbitrary lengths). Unlike the simpler examples covered in the earlier sections, the following examples are written using both the XDR C library routines and the XDR data description language. The section "Synopsis of XDR Library Routines" describes the XDR data definition language used below.

Linked Lists

The last example in Section "XDR Library Primitives" presented a C data structure and its associated XDR routines for a person's gross assets and liabilities. The example is duplicated below:

```
struct gnumbers {
   long g_assets;
   long g_liabilities;
};

bool_t
xdr_gnumbers(xdrs, gp)
   XDR *xdrs;
   struct gnumbers *gp;
{
   if (xdr_long(xdrs, &(gp->g_assets)))
      return (xdr_long(xdrs, &(gp->g_liabilities)));
   return (FALSE);
}
```

Now assume that we wish to implement a linked list of such information. A data structure could be constructed as follows:

```
typedef struct gnnode {
   struct gnumbers gn_numbers;
   struct gnnode *nxt;
};

typedef struct gnnode *gnumbers_list;
```

The head of the linked list can be thought of as the data object; that is, the head is not merely a convenient shorthand for a structure. Similarly the `nxt` field is used to indicate whether or not the object has terminated. Unfortunately, if the object continues, the `nxt` field is also the address of where it continues. The link addresses carry no useful information when the object is serialized.

The XDR data description of this linked list is described by the recursive type declaration of gnumbers_list:

```
struct gnumbers {
   unsigned g_assets;
   unsigned g_liabilities;
};

typedef union switch (boolean) {
   case TRUE: struct {
      struct gnumbers current_element;
      gnumbers_list rest_of_list;
   };
   case FALSE: struct {};
} gnumbers_list;
```

In this description, the boolean indicates whether there is more data following it. If the boolean is FALSE, then it is the last data field of the structure. If it is TRUE, then it is followed by a `gnumbers` structure and (recursively) by a `gnumbers_list` (the rest of the object). Note that the C declaration has no boolean explicitly declared in it (though the `nxt` field implicitly carries the information), while the XDR data description has no pointer explicitly declared in it.

Hints for writing a set of XDR routines to successfully (de)serialize a linked list of entries can be taken from the XDR description of the pointer-less data. The set consists of the mutually recursive routines `xdr_gnumbers_list`, `xdr_wrap_list`, and `xdr_gnnode`.

```
bool_t
xdr_gnnode(xdrs, gp)
   XDR *xdrs;
   struct gnnode *gp;
{
   return (xdr_gnumbers(xdrs, &(gp->gn_numbers)) &&
      xdr_gnumbers_list(xdrs, &(gp->nxt)) );
}

bool_t
xdr_wrap_list(xdrs, glp)
   XDR *xdrs;
   gnumbers_list *glp;
{
   return (xdr_reference(xdrs, glp, sizeof(struct gnnode),
      xdr_gnnode));
}

struct xdr_discrim choices[2] = {
   /* called if another node needs (de)serializing */
   { TRUE, xdr_wrap_list },
   /* called when there are no more nodes to  */
   /* be (de)serialized                       */
   { FALSE, xdr_void }
}

bool_t
xdr_gnumbers_list(xdrs, glp)
XDR *xdrs;
   gnumbers_list *glp;
{
   bool_t more_data;

   more_data = (*glp != (gnumbers_list)NULL);
   return (xdr_union(xdrs, &more_data, glp, choices, NULL);
}
```

The entry routine is `xdr_gnumbers_list()`; its job is to translate between the boolean value `more_data` and the list pointer values. If there is no more data, the `xdr_union()` primitive calls `xdr_void()` and the recursion is terminated. Otherwise, `xdr_union()` calls `xdr_wrap_list()`, whose job is to dereference the list pointers. The `xdr_gnnode()` routine actually (de)serializes data of the current node of the linked list, and recursively calls `xdr_gnumbers_list()` to handle the remainder of the list.

You should convince yourself that these routines function correctly in all three directions (XDR_ENCODE, XDR_DECODE and XDR_FREE) for linked lists of any length (including zero). Note that the boolean `more_data` is always initialized, but in the XDR_DECODE case it is overwritten by an externally generated value. Also note that the value of the `bool_t` is lost in the stack. The essence of the value is reflected in the list's pointers.

The unfortunate side effect of (de)serializing a list with these routines is that the C stack grows linearly with respect to the number of nodes in the list. This is due to the recursion. The routines are also hard to code (and understand) due to the number and nature of primitives involved (such as `xdr_reference`, `xdr_union`, and `xdr_void`).

The following routine collapses the recursive routines. It also has other optimizations that are discussed below.

```
bool_t
xdr_gnumbers_list(xdrs, glp)
    XDR *xdrs;
    gnumbers_list *glp;
{
    bool_t more_data;

    while (TRUE) {
       more_data = (*glp != (gnumbers_list)NULL);
       if (! xdr_bool(xdrs, &more_data))
         return (FALSE);
       if (! more_data)
         return (TRUE);  /* we are done */
       if (! xdr_reference(xdrs, glp, sizeof(struct gnnode),
          xdr_gnumbers))
          return (FALSE);
       glp = &((*glp)->nxt);
    }
}
```

The claim is that this one routine is easier to code and understand than the three recursive routines above. (It is also buggy, as discussed below.) The parameter `glp` is treated as the address of the pointer to the head of the remainder of the list to be (de)serialized. Thus, `glp` is set to the address of the current node's `nxt` field at the end of the while loop. The discriminated union is implemented in-line; the variable `more_data` has the same use in this routine as in the routines above. Its value is recomputed and re-(de)serialized each iteration of the loop. Since `*glp` is a pointer to a node, the pointer is dereferenced using `xdr_reference()`. Note that the third parameter is truly the size of a node (data values plus `nxt` pointer), while `xdr_gnumbers()` only (de)serializes the data values. We can get away with this tricky optimization only because the `nxt` data comes after all legitimate external data.

The routine is buggy in the XDR_FREE case. The bug is that `xdr_reference()` will free the node `*glp`. Upon return the assignment `glp = &((*glp)->nxt)` cannot be guaranteed to work since `*glp` is no longer a legitimate node. The following is a rewrite that works in all cases. The hard part is to avoid dereferencing a pointer which has not been initialized or which has been freed.

```
bool_t
xdr_gnumbers_list(xdrs, glp)
    XDR *xdrs;
    gnumbers_list *glp;
{
    bool_t more_data;
    bool_t freeing;
    gnumbers_list *next;  /* the next value of glp */

    freeing = (xdrs->x_op == XDR_FREE);
    while (TRUE) {
   more_data = (*glp != (gnumbers_list)NULL);
   if (! xdr_bool(xdrs, &more_data))
      return (FALSE);
   if (! more_data)
      return (TRUE);  /* we are done */
   if (freeing)
      next = &((*glp)->nxt);
   if (! xdr_reference(xdrs, glp, sizeof(struct gnnode),
      xdr_gnumbers))
      return (FALSE);
   glp = (freeing) ? next : &((*glp)->nxt);
    }
}
```

Note that this is the first example in this document that actually inspects the direction of the operation `xdrs->x_op`). (The claim is that the correct iterative implementation is still easier to understand or code than the recursive implementation. It is certainly more efficient with respect to C stack requirements.

The Record Marking Standard

A record is composed of one or more record fragments. A record fragment is a four-byte header followed by `0` to $2^{31} - 1$ bytes of fragment data. The bytes encode an unsigned binary number; as with XDR integers, the byte order is from highest to lowest. The number encodes two values — a boolean that indicates whether the fragment is the last fragment of the record (bit value 1 implies the fragment is the last fragment), and a 31-bit unsigned binary value which is the length in bytes of the fragment's data. The boolean value is the high-order bit of the header; the length is the 31 low-order bits.

(Note that this record specification is **not** in XDR standard form and cannot be implemented using XDR primitives!)

Synopsis of XDR Routines

xdr_array()

```
xdr_array(xdrs, arrp, sizep, maxsize, elsize, elproc)
      XDR *xdrs;
      char **arrp;
      u_int *sizep, maxsize, elsize;
      xdrproc_t elproc;
```

A filter primitive that translates between arrays and their corresponding external representations. The parameter `arrp` is the address of the pointer to the array, while `sizep` is the address of the element count of the array; this element count cannot exceed `maxsize` . The parameter `elsize` is the `sizeof()` each of the array's elements, and `elproc` is an XDR filter that translates between the array elements' C form, and their external representation. This routine returns one if it succeeds, zero otherwise.

xdr_bool()

```
xdr_bool(xdrs, bp)
      XDR *xdrs;
      bool_t *bp;
```

A filter primitive that translates between booleans (C integers) and their external representations. When encoding data, this filter produces values of either one or zero. This routine returns one if it succeeds, zero otherwise.

xdr_bytes()

```
xdr_bytes(xdrs, sp, sizep, maxsize)
      XDR *xdrs;
      char **sp;
      u_int *sizep, maxsize;
```

A filter primitive that translates between counted byte strings and their external representations. The parameter `sp` is the address of the string pointer. The length of the string is located at address `sizep`; strings cannot be longer than `maxsize`. This routine returns one if it succeeds, zero otherwise.

xdr_destroy()

```
void
xdr_destroy(xdrs)
      XDR *xdrs;
```

A macro that invokes the destroy routine associated with the XDR stream, `xdrs`. Destruction usually involves freeing private data structures associated with the stream. Using `xdrs` after invoking `xdr_destroy()` is undefined.

xdr_double()

```
xdr_double(xdrs, dp)
      XDR *xdrs;
      double *dp;
```

A filter primitive that translates between C `double` precision numbers and their external representations. This routine returns one if it succeeds, zero otherwise.

xdr_enum()

```
xdr_enum(xdrs, ep)
        XDR *xdrs;
        enum_t *ep;
```

A filter primitive that translates between C `enums` (actually integers) and their external representations. This routine returns one if it succeeds, zero otherwise.

xdr_float()

```
xdr_float(xdrs, fp)
        XDR *xdrs;
        float *fp;
```

A filter primitive that translates between C `floats` and their external representations. This routine returns one if it succeeds, zero otherwise.

xdr_getpos()

```
u_int
xdr_getpos(xdrs)
        XDR *xdrs;
```

A macro that invokes the get-position routine associated with the XDR stream, `xdrs`. The routine returns an unsigned integer, which indicates the position of the XDR byte stream. A desirable feature of XDR streams is that simple arithmetic works with this number, although the XDR stream instances need not guarantee this.

xdr_inline()

```
long *
xdr_inline(xdrs, len)
        XDR *xdrs;
        int len;
```

A macro that invokes the in-line routine associated with the XDR stream, `xdrs`. The routine returns a pointer to a contiguous piece of the stream's buffer; `len` is the byte length of the desired buffer. Note that the pointer is cast to `long *`. Warning: `xdr_inline()` may return 0 (NULL) if it cannot allocate a contiguous piece of a buffer. Therefore the behavior may vary among stream instances; it exists for the sake of efficiency.

xdr_int()

```
xdr_int(xdrs, ip)
        XDR *xdrs;
        int *ip;
```

A filter primitive that translates between C integers and their external representations. This routine returns one if it succeeds, zero otherwise.

xdr_long()

```
xdr_long(xdrs, lp)
        XDR *xdrs;
        long *lp;
```

A filter primitive that translates between C `long` integers and their external representations. This routine returns one if it succeeds, zero otherwise.

xdr_opaque()

```
xdr_opaque(xdrs, cp, cnt)
        XDR *xdrs;
        char *cp;
        u_int cnt;
```

A filter primitive that translates between fixed size opaque data and its external representation. The parameter `cp` is the address of the opaque object, and `cnt` is its size in bytes. This routine returns one if it succeeds, zero otherwise.

xdr_reference()

```
xdr_reference(xdrs, pp, size, proc)
        XDR *xdrs;
        char **pp;
        u_int size;
        xdrproc_t proc;
```

A primitive that provides pointer chasing within structures. The parameter `pp` is the address of the pointer; `size` is the `sizeof()` the structure that `*pp` points to; and `proc` is an XDR procedure that filters the structure between its C form and its external representation. This routine returns one if it succeeds, zero otherwise.

xdr_setpos()

```
xdr_setpos(xdrs, pos)
        XDR *xdrs;
        u_int pos;
```

A macro that invokes the set position routine associated with the XDR stream `xdrs`. The parameter `pos` is a position value obtained from `xdr_getpos()`. This routine returns one if the XDR stream could be repositioned, and zero otherwise. Warning: it is difficult to reposition some types of XDR streams, so this routine may fail with one type of stream and succeed with another.

xdr_short()

```
xdr_short(xdrs, sp)
        XDR *xdrs;
        short *sp;
```

A filter primitive that translates between C `short` integers and their external representations. This routine returns one if it succeeds, zero otherwise.

xdr_string()

```
xdr_string(xdrs, sp, maxsize)
        XDR *xdrs;
        char **sp;
        u_int maxsize;
```

A filter primitive that translates between C strings and their corresponding external representations. Strings cannot cannot be longer than `maxsize`. Note that `sp` is the address of the string's pointer. This routine returns one if it succeeds, zero otherwise.

xdr_u_int()

```
xdr_u_int(xdrs, up)
        XDR *xdrs;
        unsigned *up;
```

A filter primitive that translates between C `unsigned` integers and their external representations. This routine returns one if it succeeds, zero otherwise.

xdr_u_long()

```
xdr_u_long(xdrs, ulp)
        XDR *xdrs;
        unsigned long *ulp;
```

A filter primitive that translates between C `unsigned long` integers and their external representations. This routine returns one if it succeeds, zero otherwise.

xdr_u_short()

```
xdr_u_short(xdrs, usp)
        XDR *xdrs;
        unsigned short *usp;
```

A filter primitive that translates between C `unsigned short` integers and their external representations. This routine returns one if it succeeds, zero otherwise.

xdr_union()

```
xdr_union(xdrs, dscmp, unp, choices, dfault)
        XDR *xdrs;
        int *dscmp;
        char *unp;
        struct xdr_discrim *choices;
        xdrproc_t dfault;
```

A filter primitive that translates between a discriminated C `union` and its corresponding external representation. The parameter `dscmp` is the address of the union's discriminant, while `unp` in the address of the union. This routine returns one if it succeeds, zero otherwise.

xdr_void()

This routine always returns one. It may be passed to RPC routines that require a function parameter, where nothing is to be done.

xdr_wrapstring()

```
xdr_wrapstring(xdrs, sp)
        XDR *xdrs;
        char **sp;
```

A primitive that calls `xdr_string(xdrs,sp,MAXUNSIGNED)`; where MAXUNSIGNED is the maximum value of an unsigned integer. This is handy because the RPC package passes only two parameters XDR routines, whereas `xdr_string()`, one of the most frequently used primitives, requires three parameters. This routine returns one if it succeeds, zero otherwise.

xdrmem_create()

```
void
xdrmem_create(xdrs, addr, size, op)
        XDR *xdrs;
        char *addr;
        u_int size;
        enum xdr_op op;
```

This routine initializes the XDR stream object pointed to by `xdrs`. The stream's data is written to, or read from, a chunk of memory at location `addr` whose length is no more than `size` bytes long. The `op` determines the direction of the XDR stream (either XDR_ENCODE, XDR_DECODE, or XDR_FREE).

xdrrec_create()

```
void
xdrrec_create(xdrs, sendsize, recvsize, handle, \
                    readit, writeit)
        XDR *xdrs;
        u_int sendsize, recvsize;
        char *handle;
        int (*readit)(), (*writeit)();
```

This routine initializes the XDR stream object pointed to by `xdrs`. The stream's data is written to a buffer of size `sendsize`; a value of zero indicates the system should use a suitable default. The stream's data is read from a buffer of size `recvsize`; it too can be set to a suitable default by passing a zero value. When a stream's output buffer is full, `writeit()` is called. Similarly, when a stream's input buffer is empty, `readit()` is called. The behavior of these two routines is similar to the UNIX system calls `read` and `write`, except that `handle` is passed to the former routines as the first parameter. Note that the XDR stream's `op` field must be set by the caller. Warning: this XDR stream implements an intermediate record stream. Therefore there are additional bytes in the stream to provide record boundary information.

xdrrec_endofrecord()

```
xdrrec_endofrecord(xdrs, sendnow)
        XDR *xdrs;
        int sendnow;
```

This routine can be invoked only on streams created by `xdrrec_create()`. The data in the output buffer is marked as a completed record, and the output buffer is optionally written out if `sendnow` is non-zero. This routine returns one if it succeeds, zero otherwise.

xdrrec_eof()

```
xdrrec_eof(xdrs)
        XDR *xdrs;
        int empty;
```

This routine can be invoked only on streams created by `xdrrec_create()`. After consuming the rest of the current record in the stream, this routine returns one if the stream has no more input, zero otherwise.

xdrrec_skiprecord()

```
xdrrec_skiprecord(xdrs)
        XDR *xdrs;
```

This routine can be invoked only on streams created by `xdrrec_create()`. It tells the XDR implementation that the rest of the current record in the stream's input buffer should be discarded. This routine returns one if it succeeds, zero otherwise.

xdrstdio_create()

```
void
xdrstdio_create(xdrs, file, op)
        XDR *xdrs;
        FILE *file;
        enum xdr_op op;
```

This routine initializes the XDR stream object pointed to by `xdrs`. The XDR stream data is written to, or read from, the Standard I/O stream `file`. The parameter `op` determines the direction of the XDR stream (either XDR_ENCODE, XDR_DECODE, or XDR_FREE). Warning: the destroy routine associated with such XDR streams calls `fflush()` on the `file` stream, but never `close()`.

Chapter 19: Remote Procedure Call Programming Guide

Introduction

This chapter is intended for programmers who wish to write network applications using remote procedure calls (explained below), thus avoiding low-level system primitives based on sockets. The reader must be familiar with the C programming language, and should have a working knowledge of network theory.

Programs that communicate over a network need a paradigm for communication. A low-level mechanism might send a signal on the arrival of incoming packets, causing a network signal handler to execute. A high-level mechanism would be the Ada `rendezvous`. The method used at Sun is the Remote Procedure Call (RPC) paradigm, in which a client communicates with a server. In this process, the client first calls a procedure to send a data packet to the server. When the packet arrives, the server calls a dispatch routine, performs whatever service is requested, sends back the reply, and the procedure call returns to the client.

Layers of RPC

The RPC interface is divided into three layers. The highest layer is totally transparent to the programmer. To illustrate, at this level a program can contain a call to `rnusers()`, which returns the number of users on a remote machine. You don't have to be aware that RPC is being used, since you simply make the call in a program, just as you would call `malloc()`.

At the middle layer, the routines `registerrpc()` and `callrpc()` are used to make RPC calls: `registerrpc()` obtains a unique system-wide number, while `callrpc()` executes a remote procedure call. The `rnusers()` call is implemented using these two routines The middle-layer routines are designed for most common applications, and shield the user from knowing about sockets.

The lowest layer is used for more sophisticated applications, which may want to alter the defaults of the routines. At this layer, you can explicitly manipulate sockets used for transmitting RPC messages. This level should be avoided if possible.

The section "Higher Layers of RPC" illustrates use of the highest two layers while the section "Lower Layers of RPC" presents the low-level interface. The section "Other RPC Features" discusses miscellaneous topics. The final section, "Synopsis of RPC Routines", summarizes all the entry points into the RPC system.

Although this chapter only discusses the interface to C, remote procedure calls can be made from any language. Even though this chapter discusses RPC when it is used to communicate between processes on different machines, it works just as well for communication between different processes on the same machine.

There is a diagram of the RPC paradigm on the next page.

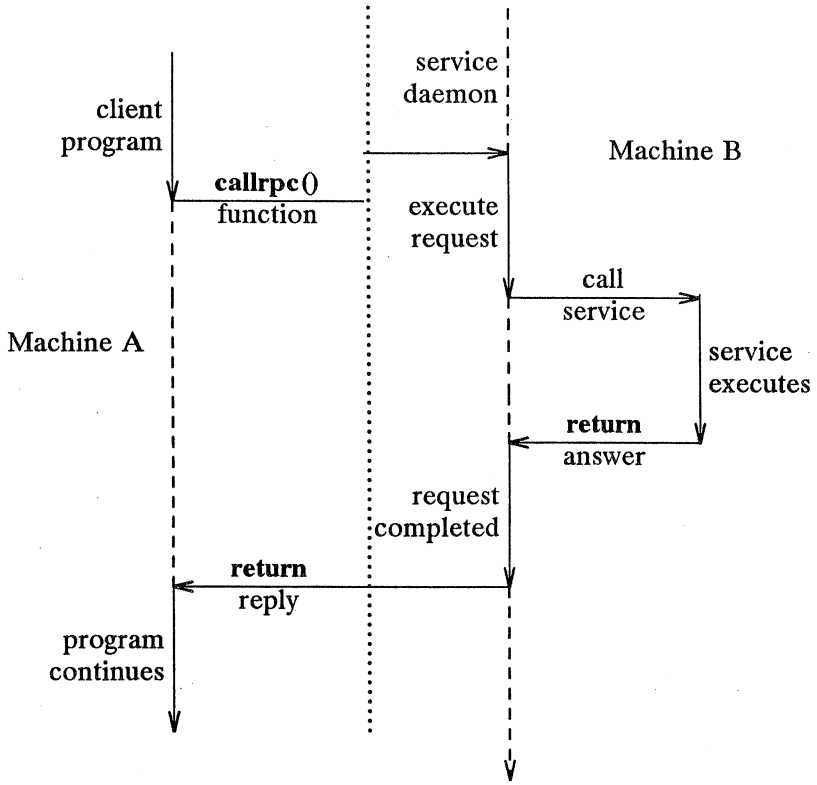


Figure 19-1: Network Communication with the Remote Procedure Call

Higher Layers of RPC

Highest Layer

Imagine you're writing a program that needs to know how many users are logged into a remote machine. You can do this by calling the library routine `rnusers()`, as illustrated below:

```
#include <stdio.h>

main(argc, argv)
    int argc;
    char **argv;
{
    unsigned num;

    if (argc < 2) {
            fprintf(stderr, "usage: rnusers hostname\n");
            exit(1);
    }
    if ((num = rnusers(argv[1])) < 0) {
      fprintf(stderr, "error: rnusers\n");
      exit(-1);
}
   printf("%d users on %s\n", num, argv[1]);
   exit(0);
}
```

RPC library routines such as `rnusers()` are included in the C library `libc.a`. Thus, the program above could be compiled with

```
cc -I/usr/include/sun -I/usr/include/bsd prog.c -lsun -lbsd
```

Another library routines is `rstat()` to gather remote performance statistics.

Intermediate Layer

The simplest interface, which explicitly makes RPC calls, uses the functions `callrpc()` and `registerrpc()`. Using this method, another way to get the number of remote users is:

```
#include <stdio.h>
#include <rpcsvc/rusers.h>

main(argc, argv)
    int argc;
    char **argv;
{
    unsigned long nusers;

    if (argc < 2) {
      fprintf(stderr, "usage: nusers hostname\n");
      exit(-1);
    }
      if (callrpc(argv[1], RUSERSPROG, RUSERSVERS,
           RUSERSPROC_NUM,
          xdr_void, 0, xdr_u_long, &nusers) != 0) {
             fprintf(stderr, "error: callrpc\n");
             exit(1);
    }
    printf("number of users on %s is %d\n", argv[1], nusers);
    exit(0);
}
```

A program number, version number, and procedure number defines each RPC procedure. The program number defines a group of related remote procedures, each of which has a different procedure number. Each program also has a version number, so when a minor change is made to a remote service (adding a new procedure, for example), a new program number doesn't have to be assigned. When you want to call a procedure to find the number of remote users, you look up the appropriate program, version and procedure numbers in a manual, similar to when you look up the name of memory allocator when you want to allocate memory.

The simplest routine in the RPC library used to make remote procedure calls is `callrpc()`. It has eight parameters. The first is the name of the remote machine. The next three parameters are the program, version, and procedure numbers. The following two parameters define the argument of the RPC call, and the final two parameters are for the return value of the call. If it completes successfully, `callrpc()` returns zero, but nonzero otherwise. The exact meaning of the return codes is found in `<rpc/clnt.h>`, and is in fact an `enum clnt_stat` cast into an integer.

Since data types may be represented differently on different machines, `callrpc()` needs both the type of the RPC argument, as well as a pointer to the argument itself (and similarly for the result). For RUSERSPROC_NUM, the return value is an `unsigned long`, so `callrpc()` has `xdr_u_long` as its first return parameter, which says that the result is of type `unsigned long`, and `&nusers` as its second return parameter, which is a pointer to where the long result will be placed. Since RUSERSPROC_NUM takes no argument, the argument parameter of `callrpc()` is `xdr_void` .

After trying several times to deliver a message, if `callrpc()` gets no answer, it returns with an error code. The delivery mechanism is UDP, which stands for User Datagram Protocol. Methods for adjusting the number of retries or for using a different protocol require you to use the lower layer of the RPC library, discussed later in this chapter. The remote server procedure corresponding to the above might

look like this:

```
char *
nuser(indata)
        char *indata;
{
        static int nusers;

        /*
         * code here to compute the number of users
         * and place result in variable nusers
         */
        return ((char *)&nusers);
}
```

It takes one argument, which is a pointer to the input of the remote procedure call (ignored in our example), and it returns a pointer to the result. In the current version of C, character pointers are the generic pointers, so both the input argument and the return value are cast to `char *`.

Normally, a server registers all of the RPC calls it plans to handle, and then goes into an infinite loop waiting to service requests. In this example, there is only a single procedure to register, so the main body of the server would look like this:

```
#include <stdio.h>
#include <rpcsvc/rusers.h>

char *nuser();

main()
{
    registerrpc(RUSERSPROG, RUSERSVERS, RUSERSPROC_NUM, nuser,
       xdr_void, xdr_u_long);
    svc_run();          /* never returns */
    fprintf(stderr, "Error: svc_run returned!\n");
    exit(1);
}
```

The `registerrpc()` routine establishes what C procedure corresponds to each RPC procedure number. The first three parameters, RUSERPROG, RUSERSVERS, and RUSERSPROC_NUM are the program, version, and procedure numbers of the remote procedure to be registered; `nuser` is the name of the C procedure implementing it; and `xdr_void` and `xdr_u_long` are the types of the input to and output from the procedure.

Only the UDP transport mechanism can use `registerrpc()` ; thus, it is always safe in conjunction with calls generated by `callrpc()`.

Warning: the UDP transport mechanism can only deal with arguments and results less than 8K bytes in length.

Assigning Program Numbers

Program numbers are assigned in groups of 0x20000000 (536870912) according to the following chart:

Number			Assignment
0	-	1fffffff	defined by sun
20000000	-	3fffffff	defined by user
40000000	-	5fffffff	transient
60000000	-	7fffffff	reserved
80000000	-	9fffffff	reserved
a0000000	-	bfffffff	reserved
c0000000	-	dfffffff	reserved
e0000000	-	ffffffff	reserved

Sun Microsystems administers the first group of numbers, which should be identical for all Sun customers. If a customer develops an application that might be of general interest, that application should be given an assigned number in the first range. The second group of numbers is reserved for specific customer applications. This range is intended primarily for debugging new programs. The third group is reserved for applications that generate program numbers dynamically. The final groups are reserved for future use, and should not be used.

The exact registration process for Sun defined numbers is yet to be established.

Passing Arbitrary Data Types

In the previous example, the RPC call passes a single `unsigned long`. RPC can handle arbitrary data structures, regardless of different machines' byte orders or structure layout conventions, by always converting them to a network standard called *eXternal Data Representation* (XDR) before sending them over the wire. The process of converting from a particular machine representation to XDR format is called *serializing*, and the reverse process is called *deserializing*. The type field parameters of `callrpc()` and `registerrpc()` can be a built-in procedure like `xdr_u_long()` in the previous example, or a user supplied one. XDR has these built-in type routines:

```
xdr_int()       xdr_u_int()     xdr_enum()
xdr_long()      xdr_u_long()    xdr_bool()
xdr_short()     xdr_u_short()   xdr_string()
```

As an example of a user-defined type routine, if you wanted to send the structure

```
struct simple {
        int a;
        short b;
} simple;
```

then you would call `callrpc` as

```
callrpc(hostname, PROGNUM, VERSNUM, PROCNUM, xdr_simple,
    &simple ...);
```

where `xdr_simple()` is written as:

```
#include <rpc/rpc.h>

xdr_simple(xdrsp, simplep)
        XDR *xdrsp;
        struct simple *simplep;
{
        if (!xdr_int(xdrsp, &simplep->a))
                return (0);
        if (!xdr_short(xdrsp, &simplep->b))
                return (0);
        return (1);
}
```

An XDR routine returns nonzero (true in the sense of C) if it completes successfully, and zero otherwise. A complete description of XDR is in the *XDR Protocol Specification*, so this section only gives a few examples of XDR implementation.

In addition to the built-in primitives, there are also the prefabricated building blocks:

```
xdr_array()         xdr_bytes()
xdr_reference()     xdr_union()
```

To send a variable array of integers, you might package them up as a structure like this

```
struct varintarr {
        int *data;
        int arrlnth;
} arr;
```

and make an RPC call such as

```
callrpc(hostname, PROGNUM, VERSNUM, PROCNUM, xdr_varintarr,
   &arr...);
```

with `xdr_varintarr()` defined as:

```
xdr_varintarr(xdrsp, varintarr)
        XDR *xdrsp;
        struct varintarr *arrp;
{
        xdr_array(xdrsp, &arrp->data, &arrp->arrlnth,
                MAXLEN,
                sizeof(int), xdr_int);
}
```

This routine takes as parameters the XDR handle, a pointer to the array, a pointer to the size of the array, the maximum allowable array size, the size of each array element, and an XDR routine for handling each array element.

If the size of the array is known in advance, then the following could also be used to send out an array of length SIZE:

```
int intarr[SIZE];

xdr_intarr(xdrsp, intarr)
      XDR *xdrsp;
      int intarr[];
{
      int i;

      for (i = 0; i < SIZE; i++) {
            if (!xdr_int(xdrsp, &intarr[i]))
                  return (0);
      }
      return (1);
}
```

XDR always converts quantities to 4-byte multiples when deserializing. Thus, if either of the examples above involved characters instead of integers, each character would occupy 32 bits. That is the reason for the XDR routine `xdr_bytes()`, which is like `xdr_array()` except that it packs characters. It has four parameters, the same as the first four parameters of `xdr_array()`. For null-terminated strings, there is also the `xdr_string()` routine, which is the same as `xdr_bytes()` without the length parameter. On serializing it gets the string length from `strlen()`, and on deserializing it creates a null-terminated string.

Here is a final example that calls the previously written `xdr_simple()` as well as the built-in functions `xdr_string()` and `xdr_reference()`, which chases pointers:

```
struct finalexample {
      char *string;
      struct simple *simplep;
} finalexample;

xdr_finalexample(xdrsp, finalp)
      XDR *xdrsp;
      struct finalexample *finalp;
{
      int i;

      if (!xdr_string(xdrsp, &finalp->string, MAXSTRLEN))
            return (0);
      if (!xdr_reference(xdrsp, &finalp->simplep,
          sizeof(struct simple), xdr_simple);
            return (0);
      return (1);
}
```

Lower Layers of RPC

In the examples given so far, RPC takes care of many details automatically for you. In this section, we'll show you how you can change the defaults by using lower layers of the RPC library. It is assumed that you are familiar with sockets and the system calls for dealing with them. If not, consult *The IPC Tutorial*.

More on the Server Side

There are a number of assumptions built into `registerrpc()`. One is that you are using the UDP datagram protocol. Another is that you don't want to do anything unusual while deserializing, since the deserialization process happens automatically before the user's server routine is called. The server for the `nusers` program shown below is written using a lower layer of the RPC package, which does not make these assumptions.

```
#include <stdio.h>
#include <rpc/rpc.h>
#include <rpcsvc/rusers.h>

int nuser();

main()
{
    SVCXPRT *transp;

    transp = svcudp_create(RPC_ANYSOCK);
    if (transp == NULL){
           fprintf(stderr, "couldn't create an RPC server\n");
    exit(1);
    }
    pmap_unset(RUSERSPROG, RUSERSVERS);
    if (!svc_register(transp, RUSERSPROG, RUSERSVERS,
         nuser,
         IPPROTO_UDP)) {
         fprintf(stderr, "couldn't register RUSER service\n");
         exit(1);
     }
     svc_run();  /* never returns */
     fprintf(stderr, "should never reach this point\n");
}
nuser(rqstp, tranp)
    struct svc_req *rqstp;
    SVCXPRT *transp;
{
    unsigned long nusers;

    switch (rqstp->rq_proc) {
    case NULLPROC:
   if (!svc_sendreply(transp, xdr_void, 0)) {
            fprintf(stderr, "couldn't reply to RPC call\n");
       exit(1);
   }
   return;
```

```
    case RUSERSPROC_NUM:
   /*
    * code here to compute the number of users
    * and put in variable nusers
    */
   if (!svc_sendreply(transp, xdr_u_long, &nusers) {
        fprintf(stderr, "couldn't reply to RPC call\n");
        exit(1);
   }
   return;
    default:
   svcerr_noproc(transp);
   return;
    }
}
```

First, the server gets a transport handle, which is used for sending out RPC messages. `registerrpc()` uses `svcudp_create()` to get a UDP handle. If you require a reliable protocol, call `svctcp_create()` instead. If the argument to `svcudp_create()` is RPC_ANYSOCK, the RPC library creates a socket on which to send out RPC calls. Otherwise, `svcudp_create()` expects its argument to be a valid socket number. If you specify your own socket, it can be bound or unbound. If it is bound to a port by the user, the port numbers of `svcudp_create()` and `clntudp_create()` (the low-level client routine) must match.

When the user specifies RPC_ANYSOCK for a socket or gives an unbound socket, the system determines port numbers in the following way: when a server starts up, it advertises to a port mapper demon on its local machine, which picks a port number for the RPC procedure if the socket specified to `svcudp_create()` isn't already bound. When the `clntudp_create()` call is made with an unbound socket, the system queries the port mapper on the machine to which the call is being made, and gets the appropriate port number. If the port mapper is not running or has no port corresponding to the RPC call, the RPC call fails. Users can make RPC calls to the port mapper themselves. The appropriate procedure numbers are in the include file `<rpc/pmap_prot.h>`.

After creating an SVCXPRT, the next step is to call `pmap_unset()` so that if the `nusers` server crashed earlier, any previous trace of it is erased before restarting. More precisely, `pmap_unset()` erases the entry for RUSERS from the port mapper's tables.

Finally, we associate the program number for `nusers` with the procedure `nuser()`. The final argument to `svc_register()` is normally the protocol being used, which, in this case, is IPPROTO_UDP. Notice that unlike `registerrpc()`, there are no XDR routines involved in the registration process. Also, registration is done on the program, rather than procedure, level.

The user routine `nuser()` must call and dispatch the appropriate XDR routines based on the procedure number. Note that two things are handled by `nuser()` that `registerrpc()` handles automatically. The first is that procedure NULLPROC (currently zero) returns with no arguments. This can be used as a simple test for detecting if a remote program is running. Second, there is a check for invalid procedure numbers. If one is detected, `svcerr_noproc()` is called to handle the error.

The user service routine serializes the results and returns them to the RPC caller via `svc_sendreply()`. Its first parameter is the SVCXPRT handle, the second is the XDR routine, and the third is a pointer to the data to be returned. Not illustrated above is how a server handles an RPC program that passes data. As an example, we can add a procedure RUSERSPROC_BOOL, which has an argument `nusers`, and returns TRUE or FALSE depending on whether there are nusers logged on. It would look like this:

```
case RUSERSPROC_BOOL: {
      int bool;
      unsigned nuserquery;

      if (!svc_getargs(transp, xdr_u_int, &nuserquery) {
            svcerr_decode(transp);
            return;
      }
      /*
       * code to set nusers = number of users
       */
      if (nuserquery == nusers)
            bool = TRUE;
      else
            bool = FALSE;
      if (!svc_sendreply(transp, xdr_bool, &bool){
             fprintf(stderr, "couldn't reply to RPC
                 call\n");
             exit(1);
      }
      return;
}
```

The relevant routine is `svc_getargs()`, which takes an SVCXPRT handle, the XDR routine, and a pointer to where the input is to be placed as arguments.

Memory Allocation with XDR

XDR routines not only do input and output, they also do memory allocation. This is why the second parameter of `xdr_array()` is a pointer to an array, rather than the array itself. If it is NULL, then `xdr_array()` allocates space for the array and returns a pointer to it, putting the size of the array in the third argument. As an example, consider the following XDR routine `xdr_chararr1()`, which deals with a fixed array of bytes with length SIZE:

```
xdr_chararr1(xdrsp, chararr)
      XDR *xdrsp;
      char chararr[];
{
      char *p;
      int len;

      p = chararr;
      len = SIZE;
      return (xdr_bytes(xdrsp, &p, &len, SIZE));
}
```

It might be called from a server like this,

```
        char chararr[SIZE];

        svc_getargs(transp, xdr_chararr1, chararr);
```

where `chararr` has already allocated space. If you want XDR to do the allocation, you would have to rewrite this routine in the following way:

```
xdr_chararr2(xdrsp, chararrp)
        XDR *xdrsp;
        char **chararrp;
{
        int len;

        len = SIZE;
        return (xdr_bytes(xdrsp, charrarrp, &len, SIZE));
}
```

Then the RPC call might look like this:

```
        char *arrptr;

        arrptr = NULL;
        svc_getargs(transp, xdr_chararr2, &arrptr);
        /*
         * use the result here
         */
        svc_freeargs(xdrsp, xdr_chararr2, &arrptr);
```

After using the character array, it can be freed with `svc_freeargs()`. In the routine `xdr_finalexample()` given earlier, if `finalp->string` was NULL in the call

```
        svc_getargs(transp, xdr_finalexample, &finalp);
```

then

```
        svc_freeargs(xdrsp, xdr_finalexample, &finalp);
```

frees the array allocated to hold `finalp->string`; otherwise, it frees nothing. The same is true for `finalp->simplep`.

To summarize, each XDR routine is responsible for serializing, deserializing, and allocating memory. When an XDR routine is called from `callrpc()`, the serializing part is used. When called from `svc_getargs()`, the deserializer is used. And when called from `svc_freeargs()`, the memory deallocator is used. When building simple examples like those in this section, a user doesn't have to worry about the three modes. The XDR reference manual has examples of more sophisticated XDR routines that determine which of the three modes they are in to function correctly.

The Calling Side

When you use `callrpc`, you have no control over the RPC delivery mechanism or the socket used to transport the data. To illustrate the layer of RPC that lets you adjust these parameters, consider the following code to call the `nusers` service:

```
#include <stdio.h>
#include <rpc/rpc.h>
#include <rpcsvc/rusers.h>
#include <sys/socket.h>
#include <sys/time.h>
#include <netdb.h>

main(argc, argv)
   int argc;
   char **argv;
{
   struct hostent *hp;
   struct timeval pertry_timeout, total_timeout;
   struct sockaddr_in server_addr;
   int addrlen, sock = RPC_ANYSOCK;
   register CLIENT *client;
   enum clnt_stat clnt_stat;
   unsigned long nusers;

   if (argc < 2) {
     fprintf(stderr, "usage: nusers hostname\n");
      exit(-1);
   }
   if ((hp = gethostbyname(argv[1])) == NULL) {
      fprintf(stderr, "cannot get addr for '%s'\n", argv[1]);
      exit(-1);
   }
   pertry_timeout.tv_sec = 3;
   pertry_timeout.tv_usec = 0;
   addrlen = sizeof(struct sockaddr_in);
   bcopy(hp->h_addr, (caddr_t)&server_addr.sin_addr, \
                        hp->h_length);
   server_addr.sin_family = AF_INET;
   server_addr.sin_port =  0;
   if ((client = clntudp_create(&server_addr, RUSERSPROG,
       RUSERSVERS, pertry_timeout, &sock)) == NULL) {
       perror("clntudp_create");
       exit(-1);
}

    total_timeout.tv_sec = 20;
    total_timeout.tv_usec = 0;
    clnt_stat = clnt_call(client, RUSERSPROC_NUM, \
                        xdr_void, 0,
          xdr_u_long, &nusers, total_timeout);
    if (clnt_stat != RPC_SUCCESS) {
            clnt_perror(client, "rpc");
            exit(-1);
    }
    clnt_destroy(client);
}
```

The low-level version of `callrpc()` is `clnt_call()`, which takes a CLIENT pointer rather than a host name. The parameters to `clnt_call()` are a CLIENT pointer, the procedure number, the XDR routine for serializing the argument, a pointer to the argument, the XDR routine for deserializing the return value, a pointer to where the return value will be placed, and the time in seconds to wait for a reply.

The CLIENT pointer is encoded with the transport mechanism. `callrpc()` uses UDP, thus it calls `clntudp_create()` to get a CLIENT pointer. To get TCP (Transport Control Protocol), you would use `clnttcp_create()`.

The parameters to `clntudp_create()` are the server address, the length of the server address, the program number, the version number, a timeout value (between tries), and a pointer to a socket. The final argument to `clnt_call()` is the total time to wait for a response. Thus, the number of tries is the `clnt_call()` timeout divided by the `clntudp_create()` timeout.

There is one thing to note when using the `clnt_destroy()` call. It deallocates any space associated with the CLIENT handle, but it does not close the socket associated with it, which was passed as an argument to `clntudp_create()`. The reason is that if there are multiple client handles using the same socket, then it is possible to close one handle without destroying the socket that other handles are using.

To make a stream connection, the call to `clntudp_create()` is replaced with a call to `clnttcp_create()`.

```
clnttcp_create(&server_addr, prognum, versnum, &socket,
    inputsize, outputsize);
```

There is no timeout argument; instead, the receive and send buffer sizes must be specified. When the `clnttcp_create()` call is made, a TCP connection is established. All RPC calls using that CLIENT handle would use this connection. The server side of an RPC call using TCP has `svcudp_create()` replaced by `svctcp_create()`.

Other RPC Features

This section discusses some other aspects of RPC that are occasionally useful.

Select on the Server Side

Suppose a process is processing RPC requests while performing some other activity. If the other activity involves periodically updating a data structure, the process can set an alarm signal before calling `svc_run()`. But if the other activity involves waiting on a a file descriptor, the `svc_run()` call won't work. The code for `svc_run()` is as follows:

```
void
svc_run()
{
    int readfds;

     for (;;) {
      readfds = svc_fds;
      switch (select(32, &readfds, NULL, NULL, NULL)) {

      case -1:
          if (errno == EINTR)
            continue;
          perror("rstat: select");
          return;
      case 0:
          break;
      default:
          svc_getreq(readfds);
      }
    }
}
```

You can bypass `svc_run()` and call `svc_getreq()` yourself. All you need to know are the file descriptors of the socket(s) associated with the programs you are waiting on. Thus you can have your own `select()` that waits on both the RPC socket, and your own descriptors.

Broadcast RPC

The `pmap` and RPC protocols implement broadcast RPC. Here are the main differences between broadcast RPC and normal RPC calls:

1. Normal RPC expects one answer, whereas broadcast RPC expects many answers (one or more answer from each responding machine).
2. Broadcast RPC can only be supported by packet-oriented (connectionless) transport protocols like UPD/IP.
3. The implementation of broadcast RPC treats all unsuccessful responses as garbage by filtering them out. Thus, if there is a version mismatch between the broadcaster and a remote service, the user of broadcast RPC never knows.

4. All broadcast messages are sent to the portmap port. Thus, only services that register themselves with their portmapper are accessible via the broadcast RPC mechanism.

Broadcast RPC Synopsis

```
#include <rpc/pmap_clnt.h>

enum clnt_stat clnt_stat;

clnt_stat = .
clnt_broadcast(prog, vers, proc, xargs, argsp,
    xresults, resultsp, eachresult)
u_long      prog;
    /* program number */
u_long      vers;
    /* version number */
u_long      proc;
    /* procedure number */
xdrproc_t   xargs;
    /* xdr routine for args */
caddr_t     argsp;
    /* pointer to args */
xdrproc_t   xresults;
    /* xdr routine for results */
caddr_t     resultsp;
    /* pointer to results */
bool_t (*eachresult)();
    /* call with each result obtained */
```

The procedure `eachresult()` is called each time a valid result is obtained. It returns a boolean that indicates whether or not the client wants more responses.

```
bool_t          done;

done =
eachresult(resultsp, raddr)
caddr_t         resultsp;
struct sockaddr_in   *raddr;
/* address of machine that sent response */
```

If `done` is TRUE, then broadcasting stops and `clnt_broadcast()` returns successfully. Otherwise, the routine waits for another response. The request is rebroadcast after a few seconds of waiting. If no responses come back, the routine returns with RPC_TIMEDOUT. To interpret `clnt_stat` errors, feed the error code to `clnt_perrno()`.

Batching

The RPC architecture is designed so that clients send a call message, and wait for servers to reply that the call succeeded. This implies that clients do not compute while servers are processing a call. This is inefficient if the client does not want or need an acknowledgement for every message sent. It is possible for clients to continue computing while waiting for a response, using RPC batch facilities.

RPC messages can be placed in a "pipeline" of calls to a desired server; this is called batching. Batching assumes that: 1) each RPC call in the pipeline requires no response from the server, and the server does not send a response message; and 2) the pipeline of calls is transported on a reliable byte stream transport such as TCP/IP. Since the server does not respond to every call, the client can generate new calls in parallel with the server executing previous calls. Furthermore, the TCP/IP implementation can buffer up many call messages, and send them to the server in one `write` system call.

This overlapped execution greatly decreases the interprocess communication overhead of the client and server processes, and the total elapsed time of a series of calls.

Since the batched calls are buffered, the client should eventually do a legitimate call in order to flush the pipeline.

A contrived example of batching follows. Assume a string rendering service (like a window system) has two similar calls: one renders a string and returns void results, while the other renders a string and remains silent. The service (using the TCP/IP transport) may look like:

```
#include <stdio.h>
#include <rpc/rpc.h>
#include <rpcsvc/windows.h>

void windowdispatch();

main()
{
    SVCXPRT *transp;

    transp = svctcp_create(RPC_ANYSOCK, 0, 0);
    if (transp == NULL){
   fprintf(stderr, "couldn't create an RPC server\n");
   exit(1);
}
    pmap_unset(WINDOWPROG, WINDOWVERS);
    if (!svc_register(transp, WINDOWPROG, WINDOWVERS,
   windowdispatch,
        IPPROTO_TCP)) {
           fprintf(stderr, "couldn't register WINDOW service\n");
       exit(1);
}
    svc_run();  /* never returns */
    fprintf(stderr, "should never reach this point\n");
}
void
windowdispatch(rqstp, transp)
    struct svc_req *rqstp;
    SVCXPRT *transp;
{
    char *s = NULL;

    switch (rqstp->rq_proc) {
    case NULLPROC:
   if (!svc_sendreply(transp, xdr_void, 0)) {
      fprintf(stderr, "couldn't reply to RPC
```

```
                call\n");
        exit(1);
    }
     return;

case RENDERSTRING:
     if (!svc_getargs(transp, xdr_wrapstring, &s)) {
            fprintf(stderr, "couldn't decode arguments\n");
       svcerr_decode(transp);
       /* tell caller he screwed up */
        break;
     }
     /*
      * call here to to render the string s
      */
    if (!svc_sendreply(transp, xdr_void, NULL)) {
       fprintf(stderr, "couldn't reply to RPC
          call\n");
       exit(1);
    }
    break;
     case RENDERSTRING_BATCHED:
    if (!svc_getargs(transp, xdr_wrapstring, &s)) {
       fprintf(stderr, "couldn't decode
          arguments\n");
       /*
        * we are silent in the face of protocol
           errors
        */
       break;
    }
    /*
     * call here to to render the string s,
     * but sends no reply!
     */
    break;
     default:
    svcerr_noproc(transp);
    return;
}
/*
 * now free string allocated while decoding arguments
 */
svc_freeargs(transp, xdr_wrapstring, &s);
}
```

Of course the service could have one procedure that takes the string and a boolean to indicate whether or not the procedure should respond.

In order for a client to take advantage of batching, the client must perform RPC calls on a TCP-based transport and the actual calls must have the following attributes: 1) the result's XDR routine must be zero (NULL), and 2) the RPC call's timeout must be zero.

Here is an example of a client that uses batching to render a bunch of strings; the batching is flushed when the client gets a null string:

```
#include <stdio.h>
#include <rpc/rpc.h>
#include <rpcsvc/windows.h>
#include <sys/socket.h>
#include <sys/time.h>
#include <netdb.h>

main(argc, argv)
   int argc;
   char **argv;
{
   struct hostent *hp;
   struct timeval pertry_timeout, total_timeout;
   struct sockaddr_in server_addr;
   int addrlen, sock = RPC_ANYSOCK;
   register CLIENT *client;
   enum clnt_stat clnt_stat;
   char buf[1000];
   char *s = buf;

   /*
    * initial as in example 3.3
    */
   if ((client = clnttcp_create(&server_addr,
            WINDOWPROG,
       WINDOWVERS, &sock, 0, 0)) == NULL) {
      perror("clnttcp_create");
      exit(-1);
   }
   total_timeout.tv_sec = 0;
   total_timeout.tv_usec = 0;
   while (scanf("%s", s) != EOF) {
      clnt_stat = clnt_call(client,
                  RENDERSTRING_BATCHED,
          xdr_wrapstring, &s, NULL, NULL,
                     total_timeout);
      if (clnt_stat != RPC_SUCCESS) {
         clnt_perror(client,
                              "batched rpc");
         exit(-1);
      }
   }
   /*
    * now flush the pipeline
    */
   total_timeout.tv_sec = 20;
   clnt_stat = clnt_call(client, NULLPROC,
       xdr_void, NULL, xdr_void, NULL, total_timeout);
   if (clnt_stat != RPC_SUCCESS) {
      clnt_perror(client, "rpc");
      exit(-1);
```

```
    }

    clnt_destroy(client);
}
```

Since the server sends no message, the clients cannot be notified of any of the failures that may occur. Therefore, clients are on their own when it comes to handling errors.

The above example was completed to render all of the (2000) lines in the file */etc/termcap*. The rendering service did nothing but to throw the lines away. The example was run in the following four configurations: 1) machine to itself, regular RPC; 2) machine to itself, batched RPC; 3) machine to another, regular RPC; and 4) machine to another, batched RPC. The results are as follows: 1) 50 seconds; 2) 16 seconds; 3) 52 seconds; 4) 10 seconds. Running `fscanf()` on */etc/termcap* only requires six seconds. These timings show the advantage of protocols that allow for overlapped execution, though these protocols are often hard to design.

Authentication

In the examples presented so far, the caller never identified itself to the server, and the server never required an ID from the caller. Clearly, some network services, such as a network filesystem, require stronger security than what has been presented so far.

In reality, every RPC call is authenticated by the RPC package on the server, and similarly, the RPC client package generates and sends authentication parameters. Just as different transports (TCP/IP or UDP/IP) can be used when creating RPC clients and servers, different forms of authentication can be associated with RPC clients; the default authentication type used as a default is type *none*.

The authentication subsystem of the RPC package is open ended. That is, numerous types of authentication are easy to support. However, this section deals only with *unix* type authentication, which besides *none* is the only supported type.

The Client Side

When a caller creates a new RPC client handle as in:

```
clnt = clntudp_create(address, prognum, versnum, wait, sockp)
```

the appropriate transport instance defaults the associate authentication handle to be

```
    clnt->cl_auth = authnone_create();
```

The RPC client can choose to use *unix* style authentication by setting `clnt->cl_auth` after creating the RPC client handle:

```
    clnt->cl_auth = authunix_create_default();
```

This causes each RPC call associated with `clnt` to carry with it the following authentication credentials structure:

```
/*
 * Unix style credentials.
 */
struct authunix_parms {
   u_long   aup_time;
        /* credentials creation time */
   char  *aup_machname;
        /* host name of where the client is calling */
   int   aup_uid;
        /* client's UNIX effective uid */
   int   aup_gid;
        /* client's current UNIX group id */
   u_int aup_len;
        /* the element length of aup_gids array */
   int   *aup_gids;
        /* array of 4.2 groups to which user belongs */
};
```

These fields are set by `authunix_create_default()` by invoking the appropriate system calls.

Since the RPC user created this new style of authentication, he is responsible for destroying it with:

```
auth_destroy(clnt->cl_auth);
```

The Server Side

Service implementors have a harder time dealing with authentication issues since the RPC package passes the service dispatch routine a request that has an arbitrary authentication style associated with it. Consider the fields of a request handle passed to a service dispatch routine:

```
/*
 * An RPC Service request
 */
struct svc_req {
   u_long  rq_prog;
                /* service program number */
   u_long  rq_vers;
            /* service protocol version number*/
   u_long  rq_proc;
            /* the desired procedure number*/
   struct opaque_auth rq_cred;
            /* raw credentials from the "wire" */
   caddr_t  rq_clntcred;
            /* read only, cooked credentials */
};
```

The `rq_cred` is mostly opaque, except for one field of interest: the style of authentication credentials:

```
/*
 * Authentication info.  Mostly opaque to the programmer.
 */
struct opaque_auth {
   enum_t   oa_flavor;
   /* style of credentials */
   caddr_t  oa_base;
   /* address of more auth stuff */
   u_int oa_length;
   /* not to exceed MAX_AUTH_BYTES */
};
```

The RPC package guarantees the following to the service dispatch routine:

1. That the request's `rq_cred` is well formed. Thus the service implementor may inspect the request's `rq_cred.oa_flavor` to determine which style of authentication the caller used. The service implementor may also wish to inspect the other fields of `rq_cred` if the style is not one of the styles supported by the RPC package.

2. That the request's `rq_clntcred` field is either NULL or points to a well formed structure that corresponds to a supported style of authentication credentials. Remember that only *unix* style is currently supported, so (currently) `rq_clntcred` could be cast to a pointer to an `authunix_parms` structure. If `rq_clntcred` is NULL, the service implementor may wish to inspect the other (opaque) fileds of `rq_cred` in case the service knows about a new type of authentication that the RPC package does not know about.

Our remote users service example can be extended so that it computes results for all users except UID 16:

```
nuser(rqstp, tranp)
   struct svc_req *rqstp;
   SVCXPRT *transp;
{
   struct authunix_parms *unix_cred;
   int uid;
   unsigned long nusers;

   /*
    * we don't care about authentication for
          /* the null procedure
    */
   if (rqstp->rq_proc == NULLPROC) {
         if (!svc_sendreply(transp, xdr_void, 0)) {
            fprintf(stderr, "couldn't reply to
         RPC call\n");
        exit(1);
      }
      return;
   }
   /*
    * now get the uid
    */
   switch (rqstp->rq_cred.oa_flavor) {
   case AUTH_UNIX:
      unix_cred = (struct authunix_parms *)
          rqstp->rq_clntcred;
      uid = unix_cred->aup_uid;
      break;

   case AUTH_NULL:
   default:
      svcerr_weakauth(transp);
      return;
   }
   switch (rqstp->rq_proc) {
   case RUSERSPROC_NUM:
   /*
    * make sure the caller is allow
        to call this procedure.
    */
   if (uid == 16) {
      svcerr_systemerr(transp);
      return;
   }
   /*
    * code here to compute the number of
    * users and put in variable nusers
    */
   if (!svc_sendreply(transp, xdr_u_long,
      &nusers) {
      fprintf(stderr, "couldn't reply to RPC
        call\n");
      exit(1);
```

```
    }
    return;
default:
    svcerr_noproc(transp);
    return;
    }
}
```

A few things should be noted here. First, it is customary not to check the authentication parameters associated with the NULLPROC (procedure number zero). Second, if the authentication parameter's type is not suitable for your service, you should call `svcerr_weakauth()`. And finally, the service protocol itself should return status for access denied; in the case of our example, the protocol does not have such a status, so we call the service primitive `svcerr_systemerr()` instead.

The last point underscores the relation between the RPC authentication package and the services; RPC deals only with authentication and not with individual services' access control. The services themselves must implement their own access control policies and reflect these policies as return statuses in their protocols.

Using inetd

An RPC server can be started from `inetd`. The only difference from the usual code is that `svcudp_create()` should be called as

```
transp = svcudp_create(0);
```

since `inet` passes a socket as file descriptor 0. Also, `svc_register()` should be called as

```
svc_register(PROGNUM, VERSNUM, service, transp, 0);
```

with the final flag as 0, since the program would already be registered by `inetd`. Remember that if you want to exit from the server process and return control to `inet`, you need to explicitly exit, since `svc_run()` never returns.

RPC services are described in entries in **/usr/etc/inetd.conf** which is read by **inetd**(1M) at boot time. The fields of the **inetd.conf** configuration file are as follows:

rpc specification
program number
version number
socket type
protocol
wait/nowait
user
server program
server program arguments

The first 3 fields tell **inetd** that the service is an RPC service, that the RPC program number is 100005, and that the program version number is 1. The socket type should be one of "stream", "dgram", "raw", "rdm", or "seqpacket", depending on whether the socket is a stream, datagram, raw, reliably delivered message, or sequenced packet socket. The protocol must be a valid protocol as given in /etc/protocols. Examples might be "tcp" or"udp". The wait/nowait entry is applicable to datagram sockets only (other sockets should have a "nowait" entry in this space). The user entry should contain the user name of the user as whom the server should run. This allows for servers to be given less permission than root. The server program entry

should contain the pathname of the program which is to be executed by inetd when a request is found on its socket. The arguments to the server program should bejust as they normally are,starting with argv[0], which is the name of the program.

For example:

```
rpc  100005  1
     dgram   udp  wait  root  /usr/etc/rpc.mountd  mountd
```

(Since all fields must be present, the fact that this entry takes two lines makes no difference, as the parser keeps reading until it has all fields.)

More Examples

Versions

By convention, the first version number of program FOO is FOOVERS_ORIG and the most recent version is FOOVERS. Suppose there is a new version of the `user` program that returns an `unsigned short` rather than a `long`. If we name this version RUSERSVERS_SHORT, then a server that wants to support both versions would do a double register.

```
if (!svc_register(transp, RUSERSPROG, RUSERSVERS_ORIG,
        nuser,
     IPPROTO_TCP)) {
    fprintf(stderr, "couldn't register RUSER
            service\n");
    exit(1);
}
if (!svc_register(transp, RUSERSPROG, RUSERSVERS_SHORT,
    nuser,
     IPPROTO_TCP)) {
    fprintf(stderr, "couldn't register RUSER
            service\n");
    exit(1);
}
```

Both versions can be handled by the same C procedure:

```
nuser(rqstp, tranp)
    struct svc_req *rqstp;
    SVCXPRT *transp;
{
    unsigned long nusers;
    unsigned short nusers2

    switch (rqstp->rq_proc) {
    case NULLPROC:
   if (!svc_sendreply(transp, xdr_void, 0)) {
      fprintf(stderr, "couldn't reply to
         RPC call\n");
      exit(1);
   }
   return;

    case RUSERSPROC_NUM:
   /*
    * code here to compute the number of users
    * and put in variable nusers
    */
   nusers2 = nusers;
   if (rqstp->rq_vers == RUSERSVERS_ORIG)
         if (!svc_sendreply(transp, xdr_u_long, &nusers) {
      fprintf(stderr, "couldn't reply to RPC call\n");
      exit(1);
       }
   else
      if (!svc_sendreply(transp, xdr_u_short, &nusers2) {
      fprintf(stderr, "couldn't reply to RPC call\n");
      exit(1);
   return;
    default:
   svcerr_noproc(transp);
   return;
   }
}
```

TCP

Here is an example that is essentially `rcp`. The initiator of the RPC `snd()` call takes its standard input and sends it to the server `rcv()`, which prints it on standard output. The RPC call uses TCP. This also illustrates an XDR procedure that behaves differently on serialization than on deserialization.

```
/*
 * The xdr routine:
 *
 * on decode, read from wire, write onto fp
 * on encode, read from fp, write onto wire
 */
#include <stdio.h>
#include <rpc/rpc.h>
```

```
xdr_rcp(xdrs, fp)
    XDR *xdrs;
    FILE *fp;
{
    unsigned long size;
    char buf[MAXCHUNK], *p;

    if (xdrs->x_op == XDR_FREE)/* nothing to free */
          return 1;
   while (1) {
     if (xdrs->x_op == XDR_ENCODE) {
            if ((size = fread (buf, sizeof(char),
         MAXCHUNK, fp))
          == 0 && ferror(fp)) {
           fprintf(stderr, "couldn't
             fread\n");
         exit(1);
        }
      }
   p = buf;
   if (!xdr_bytes(xdrs, &p, &size, MAXCHUNK))
      return 0;
   if (size == 0)
       return 1;
   if (xdrs->x_op == XDR_DECODE) {
      if (fwrite(buf, sizeof(char), size,
      fp)
        != size) {
      fprintf(stderr, "couldn't
          fwrite\n");
      exit(1);
   }
      }
    }
}

/*
 * The sender routines
 */
#include <stdio.h>
#include <netdb.h>
#include <rpc/rpc.h>
#include <sys/socket.h>
#include <sys/time.h>

main(argc, argv)
   int argc;
   char **argv;
{
   int err;

   if (argc < 2) {
     fprintf(stderr, "usage: %s server-name\n",
             argv[0]);
```

```
        exit(-1);
    }
    if ((err = callrpctcp(argv[1], RCPPROG,
          RCPPROC_FP,
        RCPVERS,
         xdr_rcp, stdin, xdr_void, 0) != 0)) {
           clnt_perrno(err);
           fprintf(stderr, " couldn't make RPC
                      call\n");
        exit(1);
    }
}

callrpctcp(host, prognum, procnum, versnum, inproc, in,
   outproc, out)
     char *host, *in, *out;
     xdrproc_t inproc, outproc;
{
     struct sockaddr_in server_addr;
     int socket = RPC_ANYSOCK;
     enum clnt_stat clnt_stat;
     struct hostent *hp;
     register CLIENT *client;
     struct timeval total_timeout;

     if ((hp = gethostbyname(host)) == NULL) {
          fprintf(stderr, "cannot get addr for '%s'\n",
             host);
      exit(-1);
     }
     bcopy(hp->h_addr, (caddr_t)&server_addr.sin_addr,
           hp->h_length);
      server_addr.sin_family = AF_INET;
      server_addr.sin_port =  0;
      if ((client = clnttcp_create(&server_addr, prognum,
        versnum, &socket, BUFSIZ, BUFSIZ)) == NULL) {
       perror("rpctcp_create");
       exit(-1);
     }
    total_timeout.tv_sec = 20;
    total_timeout.tv_usec = 0;
    clnt_stat = clnt_call(client, procnum, inproc, in,
        outproc, out, total_timeout);
   clnt_destroy(client)
   return (int)clnt_stat;
}

/*
 * The receiving routines
 */
#include <stdio.h>
#include <rpc/rpc.h>

main()
{
```

```
    register SVCXPRT *transp;

    if ((transp = svctcp_create(RPC_ANYSOCK, 1024, 1024))
       == NULL) {
        fprintf("svctcp_create: error\n");
        exit(1);
    }
    pmap_unset(RCPPROG, RCPVERS);
    if (!svc_register(transp, RCPPROG, RCPVERS, rcp_service,
        IPPROTO_TCP)) {
           fprintf(stderr, "svc_register: error\n");
       exit(1);
    }
    svc_run();  /* never returns */
    fprintf(stderr, "svc_run should never return\n");
}


rcp_service(rqstp, transp)
    register struct svc_req *rqstp;
    register SVCXPRT *transp;
{
    switch (rqstp->rq_proc) {
    case NULLPROC:
   if (svc_sendreply(transp, xdr_void, 0) == 0) {
      fprintf(stderr, "err: rcp_service");
      exit(1);
   }
   return;
     case RCPPROC_FP:
   if (!svc_getargs(transp, xdr_rcp, stdout)) {
      svcerr_decode(transp);
      return;
   }
   if (!svc_sendreply(transp, xdr_void, 0)) {
      fprintf(stderr, "can't reply\n");
      return;
   }
   exit(0);
     default:
   svcerr_noproc(transp);
   return;
     }
}
```

Callback Procedures

Occasionally, it is useful to have a server become a client, and make an RPC call back the process which is its client. An example is remote debugging, where the client is a window system program, and the server is a debugger running on the remote machine. Most of the time, the user clicks a mouse button at the debugging window, which converts this to a debugger command, and then makes an RPC call to the server (where the debugger is actually running), telling it to execute that command. However, when the debugger hits a breakpoint, the roles are reversed, and the

debugger wants to make an RPC call to the window program, so that it can inform the user that a breakpoint has been reached.

In order to do an RPC callback, you need a program number to make the RPC call on. Since this will be a dynamically generated program number, it should be in the transient range, 0x40000000 - 0x5fffffff. The routine `gettransient()` returns a valid program number in the transient range, and registers it with the portmapper. It only talks to the portmapper running on the same machine as the `gettransient()` routine itself. The call to `pmap_set()` is a test and set operation, in that it indivisibly tests whether a program number has already been registered, and if it has not, then reserves it. On return, the `sockp` argument will contain a socket that can be used as the argument to an `svcudp_create()` or `svctcp_create()` call.

```
#include <stdio.h>
#include <rpc/rpc.h>
#include <sys/socket.h>

gettransient(proto, vers, sockp)
int *sockp;
{
    static int prognum = 0x40000000;
    int s, len, socktype;
    struct sockaddr_in addr;

    switch(proto) {
   case IPPROTO_UDP:
      socktype = SOCK_DGRAM;
      break;
   case IPPROTO_TCP:
      socktype = SOCK_STREAM;
      break;
   default:
      fprintf(stderr, "unknown
          protocol type\n");
      return 0;
    }
    if (*sockp == RPC_ANYSOCK) {
   if ((s = socket(AF_INET, socktype, 0))
      < 0) {
      perror("socket");
      return (0);
   }
   *sockp = s;
}

else
   s = *sockp;
    addr.sin_addr.s_addr = 0;
    addr.sin_family = AF_INET;
    addr.sin_port = 0;
    len = sizeof(addr);
    /*
     * may be already bound, so don't check for err
     */
    bind(s, &addr, len);
    if (getsockname(s, &addr, &len)< 0) {
   perror("getsockname");
   return (0);
}
    while (pmap_set(prognum++, vers, proto,
       addr.sin_port) == 0)
       continue;
    return (prognum-1);
}
```

The following pair of programs illustrate how to use the `gettransient()` routine. The client makes an RPC call to the server, passing it a transient program number. Then the client waits around to receive a callback from the server at that program number. The server registers the program EXAMPELPROG, so that it can receive the RPC call informing it of the callback program number. Then at some random time (on receiving an ALRM signal in this example), it sends a callback RPC call, using the program number it received earlier.

```
/*
 * client
 */
#include <stdio.h>
#include <rpc/rpc.h>

int callback();
char hostname[256];

main(argc, argv)
    char **argv;
{
    int x, ans, s;
    SVCXPRT *xprt;

    gethostname(hostname, sizeof(hostname));
    s = RPC_ANYSOCK;
    x = gettransient(IPPROTO_UDP, 1, &s);
    fprintf(stderr, "client gets prognum %d\n", x);

    if ((xprt = svcudp_create(s)) == NULL) {
        fprintf(stderr, "rpc_server: svcudp_create\n");
        exit(1);
    }
(    (void)svc_register(xprt, x, 1, callback, 0);

    ans = callrpc(hostname, EXAMPLEPROG, EXAMPLEPROC_CALLBACK,
        EXAMPLEVERS, xdr_int, &x, xdr_void, 0);
    if (ans != 0) {
      fprintf(stderr, "call: ");
   clnt_perrno(ans);
   fprintf(stderr, "\n");
    }
    svc_run();
    fprintf(stderr,
        "Error: svc_run shouldn't have returned\n");
}

callback(rqstp, transp)
    register struct svc_req *rqstp;
    register SVCXPRT *transp;
{
    switch (rqstp->rq_proc) {
    case 0:
     if (svc_sendreply(transp, xdr_void, 0)  ==
          FALSE) {
   fprintf(stderr, "err: rusersd\n");
   exit(1);
       }
   exit(0);
    case 1:
     if (!svc_getargs(transp, xdr_void, 0)) {
      svcerr_decode(transp);
   exit(1);
```

```
        }
         fprintf(stderr, "client got callback\n");
    if (svc_sendreply(transp, xdr_void, 0)  ==
              FALSE) {
          fprintf(stderr, "err: rusersd");
          exit(1);
          }
     }
}

/*
 * server
 */
#include <stdio.h>
#include <rpc/rpc.h>
#include <sys/signal.h>

char *getnewprog();
char hostname[256];
int docallback();
int pnum;
    /*program number for callback routine */

main(argc, argv)
    char **argv;
{
    gethostname(hostname, sizeof(hostname));
    registerrpc(EXAMPLEPROG, EXAMPLEPROC_CALLBACK,
              EXAMPLEVERS,
         getnewprog, xdr_int, xdr_void);
    fprintf(stderr, "server going into svc_run\n");
    alarm(10);
    signal(SIGALRM, docallback);
    svc_run();
    fprintf(stderr, "Error: svc_run shouldn't have
         returned\n");
}


char *
getnewprog(pnump)
    char *pnump;
{
    pnum = *(int *)pnump;
    return NULL;
}

docallback()
{
    int ans;

    ans = callrpc(hostname, pnum, 1, 1, xdr_void,
         0, xdr_void, 0);
    if (ans != 0) {
       fprintf(stderr, "server: ");
```

```
        clnt_perrno(ans);
        fprintf(stderr, "\n");
    }
}
```

Synopsis of RPC Routines

auth_destroy()

```
void
auth_destroy(auth)
   AUTH *auth;
```

A macro that destroys the authentication information associated with `auth`. Destruction usually involves deallocation of private data structures. The use of `auth` is undefined after calling `auth_destroy()`.

authnone_create()

```
AUTH *
authnone_create()
```

Creates and returns an RPC authentication handle that passes no usable authentication information with each remote procedure call.

authunix_create()

```
AUTH *
authunix_create(host, uid, gid, len, aup_gids)
   char *host;
   int uid, gid, len, *aup_gids;
```

Creates and returns an RPC authentication handle that contains UNIX authentication information. The parameter `host` is the name of the machine on which the information was created; `uid` is the user's user ID; `gid` is the user's current group ID; `len` and `aup_gids` refer to a counted array of groups to which the user belongs. It is easy to impersonate a user.

authunix_create_default()

```
AUTH *
authunix_create_default()
```

Calls `authunix_create()` with the appropriate parameters.

callrpc()

```
callrpc(host, prognum, versnum, procnum, inproc, in,
    outproc, out) char *host;
   u_long prognum, versnum, procnum;
   char *in, *out;
   xdrproc_t inproc, outproc;
```

Calls the remote procedure associated with `prognum`, `versnum`, and `procnum` on the machine, `host`. The parameter `in` is the address of the procedure's argument(s), and `out` is the address of where to place the result(s); `inproc` is used to encode the procedure's parameters, and `outproc` is used to decode the procedure's results. This routine returns zero if it succeeds, or the value of `enum clnt_stat` cast to an integer if it fails. The routine `clnt_perrno()` is handy for translating failure statuses into messages. Warning: calling remote procedures with this routine uses UDP/IP as a transport; see `clntudp_create()` for restrictions.

clnt_broadcast()

```
enum clnt_stat
clnt_broadcast(prognum, versnum, procnum, inproc, in,
    outproc, out, eachresult)
   u_long prognum, versnum, procnum;
   char *in, *out;
   xdrproc_t inproc, outproc;
   resultproc_t eachresult;
```

Like `callrpc()`, except the call message is broadcast to all locally connected broadcast nets. Each time it receives a response, this routine calls `eachresult`, whose form is

```
eachresult(out, addr)
   char *out;
   struct sockaddr_in *addr;
```

where `out` is the same as `out` passed to `clnt_broadcast()`, except that the remote procedure's output is decoded there; `addr` points to the address of the machine that sent the results. If `eachresult()` returns zero, `clnt_broadcast()` waits for more replies; otherwise it returns with appropriate status.

clnt_call()

```
   enum clnt_stat
clnt_call(clnt, procnum, inproc, in, outproc, out, tout)
   CLIENT *clnt; long procnum;
   xdrproc_t inproc, outproc;
   char *in, *out;
   struct timeval tout;
```

A macro that calls the remote procedure `procnum` associated with the client handle, `clnt`, which is obtained with an RPC client creation routine such as `clntudp_create`. The parameter `in` is the address of the procedure's argument(s), and `out` is the address of where to place the result(s); `inproc` is used to encode the procedure's parameters, and `outproc` is used to decode the procedure's results; `tout` is the time allowed for results to come back.

clnt_destroy()

```
clnt_destroy(clnt)
   CLIENT *clnt;
```

A macro that destroys the client's RPC handle. Destruction usually involves deallocation of private data structures, including `clnt` itself. Use of `clnt` is undefined after calling `clnt_destroy()`. Warning: client destruction routines do not close sockets associated with `clnt`; this is the responsibility of the user.

clnt_freeres()

```
clnt_freeres(clnt, outproc, out)
   CLIENT *clnt;
   xdrproc_t outproc;
   char *out;
```

A macro that frees any data allocated by the RPC/XDR system when it decoded the results of an RPC call. The parameter `out` is the address of the results, and `outproc` is the XDR routine describing the results in simple primitives. This routine returns one if the results were successfully freed, and zero otherwise.

clnt_geterr()

```
void
clnt_geterr(clnt, errp)
   CLIENT *clnt;
   struct rpc_err *errp;
```

A macro that copies the error structure out of the client handle to the structure at address `errp`.

clnt_pcreateerror()

```
void
clnt_pcreateerror(s)
   char *s;
```

Prints a message to standard error indicating why a client RPC handle could not be created. The message is prepended with string `s` and a colon.

clnt_perrno()

```
void
clnt_perrno(stat)
   enum clnt_stat;
```

Prints a message to standard error corresponding to the condition indicated by `stat`.

clnt_perror()

```
clnt_perror(clnt, s)
   CLIENT *clnt;
   char *s;
```

Prints a message to standard error indicating why an RPC call failed; `clnt` is the handle used to do the call. The message is prepended with string `s` and a colon.

clntraw_create()

```
CLIENT *
clntraw_create(prognum, versnum)
   u_long prognum, versnum;
```

This routine creates a toy RPC client for the remote program `prognum`, version `versnum`. The transport used to pass messages to the service is actually a buffer within the process's address space, so the corresponding RPC server should live in the same address space; see `svcraw_create()`. This allows simulation of RPC and acquisition of RPC overheads, such as round trip times, without any kernel interference. This routine returns NULL if it fails.

clnttcp_create()

```
   CLIENT *
clnttcp_create(addr, prognum, versnum, sockp, sendsz, recvsz)
   struct sockaddr_in *addr;
   u_long prognum, versnum;
   int *sockp;
   u_int sendsz, recvsz;
```

This routine creates an RPC client for the remote program `prognum`, version `versnum`; the client uses TCP/IP as a transport. The remote program is located at Internet address `*addr`. If `addr->sin_port` is zero, then it is set to the actual port that the remote program is listening on (the remote *portmap* service is consulted for this information). The parameter `*sockp` is a socket; if it is RPC_ANYSOCK, then this routine opens a new one and sets `*sockp` . Since TCP-based RPC uses buffered I/O, the user may specify the size of the send and receive buffers with the parameters `sendsz` and `recvsz` ; values of zero choose suitable defaults. This routine returns NULL if it fails.

clntudp_create()

```
CLIENT *
clntudp_create(addr, prognum, versnum, wait, sockp)
   struct sockaddr_in *addr;
   u_long prognum, versnum;
   struct timeval wait;
   int *sockp;
```

This routine creates an RPC client for the remote program `prognum`, version `versnum`; the client uses use UDP/IP as a transport. The remote program is located at Internet address `*addr`. If `addr->sin_port` is zero, then it is set to actual port that the remote program is listening on (the remote *portmap* service is consulted for this information). The parameter `*sockp` is a socket; if it is RPC_ANYSOCK, then this routine opens a new one and sets `*sockp`. The UDP transport resends the call message in intervals of `wait` time until a response is received or until the call times out. Warning: since UDP-based RPC messages can only hold up to 8 Kbytes of encoded data, this transport cannot be used for procedures that take large arguments or return huge results.

get_myaddress()

```
void
get_myaddress(addr)
   struct sockaddr_in *addr;
```

Stuffs the machine's IP address into `*addr`, without consulting the library routines that deal with */etc/hosts*. The port number is always set to `htons(PMAPPORT)`.

pmap_getmaps()

```
struct pmaplist *
pmap_getmaps(addr)
   struct sockaddr_in *addr;
```

A user interface to the *portmap* service, which returns a list of the current RPC program-to-port mappings on the host located at IP address `*addr`. This routine can return NULL. The command `rpcinfo -p` uses this routine.

pmap_getport()

```
u_short
pmap_getport(addr, prognum, versnum, protocol)
   struct sockaddr_in *addr;
   u_long prognum, versnum, protocol;
```

A user interface to the *portmap* service, which returns the port number on which waits a service that supports program number `prognum` , version `versnum`, and speaks the transport protocol associated with protocol. A return value of zero means that the mapping does not exist or that the RPC system failured to contact the remote *portmap* service. In the latter case, the global variable `rpc_createerr` contains the RPC status.

pmap_rmtcall()

```
enum clnt_stat
pmap_rmtcall(addr, prognum, versnum, procnum,
        inproc, in, outproc, out, tout, portp)
   struct sockaddr_in *addr;
   u_long prognum, versnum, procnum;
   char *in, *out;
   xdrproc_t inproc, outproc;
   struct timeval tout;
   u_long *portp;
```

A user interface to the *portmap* service, which instructs *portmap* on the host at IP address `*addr` to make an RPC call on your behalf to a procedure on that host. The parameter `*portp` will be modified to the program's port number if the procedure succeeds. The definitions of other parameters are discussed in `callrpc()` and `clnt_call()`; see also `clnt_broadcast()`.

pmap_set()

```
pmap_set(prognum, versnum, protocol, port)
   u_long prognum, versnum, protocol;
   u_short port;
```

A user interface to the *portmap* service, which establishes a mapping between the triple `[prognum,versnum,protocol]` and `port` on the machine's *portmap* service. The value of protocol is most likely IPPROTO_UDP or IPPROTO_TCP. This routine returns one if it succeeds, zero otherwise.

pmap_unset()

```
pmap_unset(prognum, versnum)
   u_long prognum, versnum;
```

A user interface to the *portmap* service, which destroys all mappings between the triple `[prognum,versnum,*]` and `ports` on the machine's *portmap* service. This routine returns one if it succeeds, zero otherwise.

registerrpc()

```
registerrpc(prognum, versnum, procnum, procname, inproc, outproc)
   u_long prognum, versnum, procnum;
   char *(*procname)();
   xdrproc_t inproc, outproc;
```

Registers procedure `procname` with the RPC service package. If a request arrives for program `prognum`, version `versnum`, and procedure `procnum`, `procname` is called with a pointer to its parameter(s); `progname` should return a pointer to its static result(s); `inproc` is used to decode the parameters while `outproc` is used to encode the results. This routine returns zero if the registration succeeded, −1 otherwise.

Warning: remote procedures registered in this form are accessed using the UDP/IP transport; see `svcudp_create()` for restrictions.

rpc_createerr

```
struct rpc_createerr rpc_createerr;
```

A global variable whose value is set by any RPC client creation routine that does not succeed. Use the routine `clnt_pcreateerror()` to print the reason why.

svc_destroy()

```
svc_destroy(xprt)
   SVCXPRT *xprt;
```

A macro that destroys the RPC service transport handle, `xprt`. Destruction usually involves deallocation of private data structures, including `xprt` itself. Use of `xprt` is undefined after calling this routine.

svc_fds

```
int   svc_fds;
```

A global variable reflecting the RPC service side's read file descriptor bit mask; it is suitable as a parameter to the `select` system call. This is only of interest if a service implementor does not call `svc_run()`, but rather does his own asynchronous event processing. This variable is read-only (do not pass its address to `select`!), yet it may change after calls to `svc_getreq()` or any creation routines.

svc_freeargs()

```
svc_freeargs(xprt, inproc, in)
   SVCXPRT *xprt;
   xdrproc_t inproc;
   char *in;
```

A macro that frees any data allocated by the RPC/XDR system when it decoded the arguments to a service procedure using `svc_getargs()`. This routine returns one if the results were successfully freed, and zero otherwise.

svc_getargs()

```
svc_getargs(xprt, inproc, in)
   SVCXPRT *xprt;
   xdrproc_t inproc;
   char *in;
```

A macro that decodes the arguments of an RPC request associated with the RPC service transport handle, `xprt`. The parameter `in` is the address where the arguments will be placed; `inproc` is the XDR routine used to decode the arguments. This routine returns one if decoding succeeds, and zero otherwise.

svc_getcaller()

```
struct sockaddr_in
svc_getcaller(xprt)
   SVCXPRT *xprt;
```

The approved way of getting the network address of the caller of a procedure associated with the RPC service transport handle, `xprt`.

svc_getreq()

```
svc_getreq(rdfds)
   int rdfds;
```

This routine is only of interest if a service implementor does not call `svc_run()`, but instead implements custom asynchronous event processing. It is called when the `select` system call has determined that an RPC request has arrived on some RPC socket(s); `rdfds` is the resultant read file descriptor bit mask. The routine returns when all sockets associated with the value of `rdfds` have been serviced.

svc_register()

```
svc_register(xprt, prognum, versnum, dispatch, protocol)
   SVCXPRT *xprt;
   u_long prognum, versnum;
   void (*dispatch) ();
   u_long protocol;
```

Associates `prognum` and `versnum` with the service dispatch procedure, `dispatch()`. If `protocol` is zero, the service is not registered with the *portmap* service. If `protocol` is non-zero, then a mapping of the triple `[prognum, versnum, protocol]` to `xprt->xp_port` is established with the local *portmap* service (generally `protocol` is zero, `IPPROTO_UDP` or `IPPROTO_TCP`. The procedure `dispatch()` has the following form:

```
dispatch(request, xprt)
   struct svc_req *request;
   SVCXPRT *xprt;
```

The `svc_register()` routine returns one if it succeeds, and zero otherwise.

svc_run()

```
svc_run()
```

This routine never returns. It waits for RPC requests to arrive, and calls the appropriate service procedure using `svc_getreq()` when one arrives. This procedure is usually waiting for a `select()` system call to return.

svc_sendreply()

```
svc_sendreply(xprt, outproc, out)
   XVCXPRT *xprt;
   xdrproc_t outproc;
   char *out;
```

Called by an RPC service's dispatch routine to send the results of a remote procedure call. The parameter `xprt` is the caller's associated transport handle; `outproc` is the XDR routine which is used to encode the results; and `out` is the address of the results. This routine returns one if it succeeds, zero otherwise.

svc_unregister()

```
void
svc_unregister(prognum, versnum)
   u_long prognum, versnum;
```

Removes all mapping of the double `[prognum, versnum]` to dispatch routines, and of the triple `[prognum, versnum, *]` to port number.

svcerr_auth()

```
void
svcerr_auth(xprt, why)
   SVCXPRT *xprt;
   enum auth_stat why;
```

Called by the service dispatch routine that refuses to perform a remote procedure call due to an authentication error.

svcerr_decode()

```
void
svcerr_decode(xprt)
   SVCXPRT *xprt;
```

Called by a service dispatch routine that can't successfully decode its parameters. See also `svc_getargs()`.

svcerr_noproc()

```
void
svcerr_noproc(xprt)
   SVCXPRT *xprt;
```

Called by a service dispatch routine that doesn't implement the desired procedure number the caller requests.

svcerr_noprog()

```
void
svcerr_noprog(xprt)
   SVCXPRT *xprt;
```

Called when the desired program is not registered with the RPC package. Service implementors usually don't need this routine.

svcerr_progvers()

```
void
svcerr_progvers(xprt)
   SVCXPRT *xprt;
```

Called when the desired version of a program is not registered with the RPC package. Service implementors usually don't need this routine.

svcerr_systemerr()

```
void
svcerr_systemerr(xprt)
   SVCXPRT *xprt;
```

Called by a service dispatch routine when it detects a system error not covered by any particular protocol. For example, if a service can no longer allocate storage, it may call this routine.

svcerr_weakauth()

```
void
svcerr_weakauth(xprt)
   SVCXPRT *xprt;
```

Called by a service dispatch routine that refuses to perform a remote procedure call due to insufficent (but correct) authentication parameters. The routine calls `svcerr_auth(xprt, AUTH_TOOWEAK)`.

svcraw_create()

```
SCVXPRT *
svcraw_create()
```

This routine creates a toy RPC service transport, to which it returns a pointer. The transport is really a buffer whithin the process' address space, to the corresponding RPC client should line in the same address space; see `clntraw_create()`. This routine allows simulation of RPC and acquisition of RPC overheads (such as round trip times), without any kernel interference. This routine returns `NULL` if it fails.

svctcp_create()

```
SVCXPRT *
svctcp_create(sock, send_buf_size, recv_buf_size)
   int sock;
   u_int send_buf_size, recv_buf_size;
```

This routine creates a TCP/IP-based RPC service transport, to which it returns a pointer. The transport is associated with the socket `sock`, which may be `RPC_ANYSOCK`, in which care a new socket is created. If the socket is not bound to a local TCP port, then this routine binds it to an arbitrary port. Upon completion, `xprt->xp` is the transport's socket number, and `xprt->xp_port` is the transport's

port number. This routine returns NULL if it fails. Since TCP-based RPC uses buffered I/O, users may specify the size of the send and receive buffers; values of zero choose suitable defaults.

svcudp_create()

```
SVCXPRT *
svcudp_create(sock)
   int sock;
```

This routine create a UDP/IP-based RPC service transport, to which it returns a pointer. The transport is accociated with the socket socket, which may be RPC_ANYSOCK, in which case a new socket is created. If the socket is not bound to a lockal UDP port, then this routine binds it to an arbitrary port. Upon completion, xprt->xp_sock is the transport's socket number, and xprt->xp_port is the transport's port number. This routine returns NULL if it fails. Warning: since UDP-based RPC messages can only hold up to 8 Kbytes of encoded data, this transport cannot be used for procedures that take large arguments or return huge results.

xdr_accepted_reply()

```
xdr_accepted_reply(xdrs, ar)
   XDR *xdrs;
   struct accepted_reply *ar;
```

Used for describing RPS messages, externally. This routine is useful for users who wish to generate RPC-style messages without using the RPC package.

xdr_array()

```
xdr_array(xdrs, arrp, sizep, maxsize, elsize, elproc)
   XDR *xdrs;
   char **arrp;
   u_int *sizep, maxsize, elsize;
   xdrproc_t elproc;
```

A filter primitive that translates between arrays and their corresponding external representations. The parameter arrp is the address of the pointer to the array, while sizep is the address of the element count of the array; this element count cannot exceed maxsize. The parameter elsize is the sizeof() each of the array's elements, and elproc is an XDR filter that translates between the array elements' C form, and their external representation. This routine returns one of it succeeds, zero otherwise.

xdr_authunix_parms()

```
xdr_authunix_parms(xdrs, aupp)
   XDR *xrds;
   struct authunix_parms *aupp;
```

Used for describing UNIX credrentials, externally. This routine is useful for users who wish to generate these credentials without using the RPC authentication package.

xdr_bool()

```
xdr_bool(xdrs, bp)
   XDR *xdrs;
   bool_t *bp;
```

A filter primitive that tranlates between booleans (C integers) and their external representations. When encoding data, this filter produces values of either one or zero. This routine returns one if it suceeds, zero otherwise.

```
xdr_bytes(xdrs, sp, sizep, maxsize)
   XDR *xdrs;
   char **sp;
   u_int *sizep, maxsize;
```

A filter primitive that translates between counted byte strings and their external representations. The parameter `sp` is the address of the string pointer. The length of the string is located at address `sizep`; strings cannot be longer than `maxsize`. This routine returns one if it succeeds, zero otherwise.

xdr_callhdr()

```
void
xdr_callhdr(xdrs, chdr)
   XDR *xdrs;
   struct rpc_msg *chdr;
```

Used for describing RPC messages, externally. This routine is useful for users who wich to generate RPC-style messages without using the RPC package.

xdr_callmsg()

```
xdr_callmsg(xdrs, cmsg)
   XDR *xdrs;
   struct rpc_ms *cmsg;
```

Used for describing RPC messages, externally. This routine is useful for users who wish to generate RPC-sytle messages without using the RPC package.

xdr_double()

```
xdr_double(xdrs, dp)
   XDR *xdrs;
   double *dp;
```

A filter primitive that translates between C `double` precision numbers and their external representations. This routine returns one if it succeds, zero otherwise.

xdr_enum()

```
xdr_enum(xdrs, ep)
   XDR *xdrs;
   enum_t *ep;
```

A filter primitive that translates between C `enums` (actually integers) and their external representations. This routine returns one if it succeeds, zero otherwise.

xdr_float()

```
xdr_float(xdrs, fp)
   XDR *xdrs;
   float *fp;
```

A filter primitive that translates between C `floats` and their external representations. This routine returns one if it succeeds, zero otherwise.

xdr_inline()

```
long*
xdr_inline(xdrs, len)
   XDR *xdrs;
   int len;
```

A macro that invokes the in-line routine associated with the XDR stream, `xdrs`. The routine returns a pointer to a contigous piece of the stream's buffer; `len` is the byte length of the desired buffer. Note that pointer is cast to `long *`. Warning: `xdr_inline()` may return `NULL(0)` if it cannot allocate a contigous piece of a buffer. Therefore the behavior may vay among stream instances; if exists for the sake of efficiency.

xdr_int()

```
xdr_int(xdrs, ip)
   XDR *xdrs;
   int *ip;
```

A filter primitive that translates between C integers and their external representations. This routine returns one if it succeeds, zero otherwise.

xdr_long()

```
xdr_long(xdrs, lp)
   XDR *xdrs;
   long *lp;
```

A filter primitive that translates between C `long` integers and their external representations. This routine returns one if it succeeds, zero otherwise.

xdr_opaque()

```
xdr_opaque(xdrs, cp, cnt)
   XDR *xdrs;
   char * cp;
   u_int cnt;
```

A filter primitive that translates between fized size opaque data and its external representation. The parameter `cp` is the address of the opaque object, and `cnt` is its size in bytes. This routine returns one if it succeeds, zero otherwise.

xdr_opaque_auth()

```
xdr_opaque_auth(xdrs, ap)
   XDR *xdrs;
   struct opaque_auth *ap;
```

Used for describing messages externally. This routine is useful for users who wish to generate RPC-style messages without using the RPC package.

xdr_pmap()

```
xdr_pmap(xdrs, regs)
   XDR *xdrs;
   struct pmap *regs;
```

Used for describing parameters to various *portmap* procedures, externally. This routine is usefule for users who wish to generate these parameters without using the pmap interface.

xdr_pmaplist()

```
xdr_pmaplist(xdrs, rp)
   XDR *xdrs;
   struct pmaplist **rp;
```

Used for describing a list of port mappings, externally. This routine is usefule for users who wish to generate these parameters without using the pmap interface.

xdr_reference()

```
xdr_reference(xdrs, pp, size, proc)
   XDR *xdrs;
   char **pp;
   u_int size;
   sdrproc_t proc;
```

A primitive that provides pointer chasing within structures. The parameter pp is the address of the pointer, size is the sizeof() the structure that **pp points to; and proc is an XDR procedure that filters the structure between its C form and its external representation. This routine returns one if it succeeds, zero otherwise.

xdr_rejected_reply()

```
xdr_rejected_reply(xdrs, rr)
   XDR *xdrs;
   struct rejected_reply *rr;
```

Used for describing RPC messages, externally. This routine is useful for users who wish to generate RPC-style messages without using the RPC package.

xdr_replymsg()

```
xdr_replymsg(xdrs, rmsg)
   XDR *xdrs;
   struct rpc_msg *rmsg;
```

Used for describing RPC messages, externally. This routine is useful for users who wich to generate RPC-style messages without using the RPC package.

xdr_short()

```
xdr_short(xdrs, sp)
   XDR *xdrs;
   short *sp;
```

A filter primitive that translates between C `short` integers and their external representations. This routine returns one if it succeeds, zero otherwise.

xdr_string()

```
xdr_string(xdrs, sp, maxsize)
   XDR *xdrs;
   char **sp;
   u_int maxsize;
```

A filter primitive that translates between C strings and their corresponding external representations. Strings cannot be longer than `maxsize`. Note that `sp` is the address of the string's pointer. This routine returns one if it succeeds, zero otherwise.

xdr_u_int()

```
xdr_u_int(xdrs, up)
   XDR *xdrs;
   unsigned *up;
```

A filter primitive that translates between C `unsigned` integers and their external representations. This routine returns one if it succeeds, zero otherwise.

xdr_u_long()

```
sxdr_u_long(xdrs, ulp)
   XDR *xdrs;
   unsigneds long *ulp;
```

A filter primitive that translates between C `unsigned long` integers and their external representations. This routine returns one if it succeeds, zero otherwise.

xdr_u_short()

```
xdr_u_short(xdrs, usp)
   XDR *xdrs;
   unsigned short *usp;
```

A filter primitive that translates between C `unsigned short` integers and their external representations. This routine returns one if it succeeds, zero otherwise.

xdr_union()

```
xdr_union(xdrs, dscmp, unp, choices, dfault)
   XDR *xdrs;
   int *dcsmp;
   char *unp;
   struct xdr_discrim *choices;
   xdrproc_t default;
```

A filter primitive that translates between a discriminated C `union` and its corresponding external representation. The parameter `dscmp` is the address of the union's discriminant, while `unp` in the address of the union. This routine returns one if it succeeds, zero otherwise.

xdr_void()

```
xdr_void
```

This routine always returns one.

xdr_wrapstring()

```
xdr_wrapstring(xdrs, sp)
   XDR *xdrs;
   char **sp;
```

A primitive that calls `xdr_string(xdrs, sp, MAXUNSIGNED);` where `MAXUNSIGNED` is the maximum value of an unsigned integer. This is handy because the RPC package passes only two parameters XDR routines, whereas `xdr_string()`, one of the most frequently used primitives, requires three parameters. This routine returns one if it succeeds, zero otherwise.

xprt_register()

```
void
xprt_register(xprt)
   SVCXPRT *xprt;
```

After RPC service transport handles are created, they should register themselves with the RPC service package. This routine modifies the global variable `svc_fds`. Service implementors usually don't need this routine.

xprt_unregister()

```
void
xprt_unregister(xprt)
   SVCXPRT *xprt;
```

Before an RPC service transport handle is destroyed, it should unregister itself with the RPC service package. This routine modifies the global variable `svc_fds`. Service implementors usually don't need this routine.

Glossary

Ada

Named after the Countess of Lovelace, the nineteenth century mathematician and computer pioneer, Ada is a high-level general-purpose programming language developed under the sponsorship of the U.S. Department of Defense. Ada was developed to provide consistency among programs originating in different branches of the military. Ada features include packages that make data objects visible only to the modules that need them, task objects that facilitate parallel processing, and an exception handling mechanism that encourages well-structured error processing.

ANSI standard

ANSI is the acronym for the American National Standards Institute. ANSI establishes guidelines in the computing industry, from the definition of ASCII to the determination of overall datacom system performance. ANSI standards have been established for both the Ada and FORTRAN programming languages, and a standard for C has been proposed.

a.out file

a.out is the default file name used by the link editor when it outputs a successfully compiled, executable file. **a.out** contains object files that are combined to create a complete working program. Object file format is described in the chapter "The Common Object File Format," and in **a.out**(4) in the *Programmer's Reference Manual*.

application program

An application program is a working program in a system. Such programs are usually unique to one type of users' work, although some application programs can be used in a variety of business situations. An accounting application, for example, may well be applicable to many different businesses.

archive

An archive file or archive library is a collection of data gathered from several files. Each of the files within an archive is called a member. The command **ar**(1) collects data for use as a library.

argument

An argument is additional information that is passed to a command or a function. On a command line, an argument is a character string or number that follows the command name and is separated from it by a space. There are two types of command-line arguments: options and operands. Options are immediately preceded by a minus sign (**–**) and change the execution or output of the command. Some options can themselves take arguments. Operands are preceded by a space and specify files or directories that will be operated on by the command. For example, in the command

pr –t –h Heading file

all of the elements after the **pr** are arguments. **–t** and **–h** are options, **Heading** is an argument to the **–h** option, and **file** is an operand.

For a function, arguments are enclosed within a pair of parentheses immediately following the function name. The

number of arguments can be zero or more; if more than two are present they are separated by commas and the whole list enclosed by the parentheses. The formal definition of a function, such as might be found on a page in Section 3 of the *Programmer's Reference Manual*, describes the number and data type of argument(s) expected by the function.

ASCII — ASCII is an acronym for American Standard Code for Information Interchange, a standard for data representation that is followed in the UNIX system. ASCII code represents alphanumeric characters as binary numbers. The code includes 128 upper- and lower-case letters, numerals, and special characters. Each alphanumeric and special character has an ASCII code (binary) equivalent that is one byte long.

assembler — The assembler is a translating program that accepts instructions written in the assembly language of the computer and translates them into the binary representation of machine instructions. In many cases, the assembly language instructions map 1 to 1 with the binary machine instructions.

assembly language — A programming language that uses the instruction set that applies to a particular computer.

BASIC — BASIC is a high-level conversational programming language that allows a computer to be used much like a complex electronic calculating machine. The name is an acronym for Beginner's All-purpose Symbolic Instruction Code.

branch table — A branch table is an implementation technique for fixing the addresses of text symbols, without forfeiting the ability to update code. Instead of being directly associated with function code, text symbols label jump instructions that transfer control to the real code. Branch table addresses do not change, even when one changes the code of a routine. Jump table is another name for branch table.

buffer — A buffer is a storage space in computer memory where data are stored temporarily into convenient units for system operations. Buffers are often used by programs, such as editors, that access and alter text or data frequently. When you edit a file, a copy of its contents are read into a buffer where you make changes to the text. For the changes to become part of the permanent file, you must write the buffer contents back into the permanent file. This replaces the contents of the file with the contents of the buffer. When you quit the editor, the contents of the buffer are flushed.

byte — A byte is a unit of storage in the computer. On many UNIX systems, a byte is eight bits (binary digits), the equivalent of one character of text.

byte order — Byte order refers to the order in which data are stored in computer memory.

C — The C programming language is a general-purpose programming language that features economy of expression, control flow, data structures, and a variety of operators. It can be used to perform both high-level and low-level tasks. Although

it has been called a system programming language, because it is useful for writing operating systems, it has been used equally effectively to write major numerical, text-processing, and data base programs. The C programming language was designed for and implemented on the UNIX system; however, the language is not limited to any one operating system or machine.

C compiler

The C compiler converts C programs into assembly language programs that are eventually translated into object files by the assembler.

C preprocessor

The C preprocessor is a component of the C Compilation System. In C source code, statements preceded with a pound sign (**#**) are directives to the preprocessor. Command line options of the **cc**(1) command may also be used to control the actions of the preprocessor. The main work of the preprocessor is to perform file inclusions and macro substitution.

CCS

CCS is an acronym for C Compilation System, which is a set of programming language utilities used to produce object code from C source code. The major components of a C Compilation System are a C preprocessor, C compiler, assembler, and link editor. The C preprocessor accepts C source code as input, performs any preprocessing required, then passes the processed code to the C compiler, which produces assembly language code that it passes to the assembler. The assembler in turn produces object code that can be linked to other object files by the link editor. The object files produced are in the Common Object File Format (COFF). Other components of CCS include a symbolic debugger, an optimizer that makes the code produced as efficient as possible, productivity tools, tools used to read and manipulate object files, and libraries that provide runtime support, access to system calls, input/output, string manipulation, mathematical functions, and other code processing functions.

COBOL

COBOL is an acronym for COmmon Business Oriented Language. COBOL is a high-level programming language designed for business and commercial applications. The English-language statements of COBOL provide a relatively machine-independent method of expressing a business-oriented problem to the computer.

COFF

COFF is an acronym for Common Object File Format. COFF refers to the format of the output file produced on some UNIX systems by the assembler and the link editor. This format is also used by other operating systems. The following are some of its key features:

- ☐ Applications may add system-dependent information to the object file without causing access utilities to become obsolete.
- ☐ Space is provided for symbolic information used by debuggers and other applications.

- ☐ Users may make some modifications in the object file construction at compile time.

command — A command is the term commonly used to refer to an instruction that a user types at a computer terminal keyboard. It can be the name of a file that contains an executable program or a shell script that can be processed or executed by the computer on request. A command is composed of a word or string of letters and/or special characters that can continue for several (terminal) lines, up to 256 characters. A command name is sometimes used interchangeably with a program name.

command line — A command line is composed of the command name followed by any argument(s) required by the command or optionally included by the user. The manual page for a command includes a command line synopsis in a notation designed to show the correct way to type in a command, with or without options and arguments.

compiler — A compiler transforms the high-level language instructions in a program (the source code) into object code or assembly language. Assembly language code may then be passed to the assembler for further translation into machine instructions..

core — Core is a (mostly archaic) synonym for primary memory.

core file — A core file is an image of a terminated process saved for debugging. A core file is created under the name "core" in the current directory of the process when an abnormal event occurs resulting in the process' termination. A list of these events is found in the **signal**(2) manual page in section 2 of the *Programmer's Reference Manual.*

core image — Core image is a copy of all the segments of a running or terminated program. The copy may exist in main storage, in the swap area. or in a core file.

curses — **curses**(3X) is a library of C routines that are designed to handle input, output, and other operations in screen management programs. The name **curses** comes from the cursor optimization that the routines provide. When a screen management program is run, cursor optimization minimizes the amount of time a cursor has to move about a screen to update its contents. The program refers to the **terminfo**(4) data base at run time to obtain the information that it needs about the screen (terminal) being used. See **terminfo**(4) in the *Programmer's Reference Manual.*

data symbol — A data symbol names a variable that may or may not be initialized. Normally, these variables reside in read/write memory during execution. See text symbol.

data base — A data base is a bank of information on a particular subject or subjects. On-line data bases are designed so that by using subject headings, key words, or key phrases you can search for, analyze, update, and print out data.

debug

Debugging is the process of locating and correcting errors in computer programs.

default

A default is the way a computer will perform a task in the absence of other instructions.

delimiter

A delimiter is an initial character that identifies the next character or character string as a particular kind of argument. Delimiters are typically used for option names on a command line; they identify the associated word as an option (or as a string of several options if the options are bundled). In the UNIX system command syntax, a minus sign (**−**) is most often the delimiter for option names, for example, **−s** or **−n**, although some commands also use a plus sign (**+**).

directory

A directory is a type of file used to group and organize other files or directories. A directory consists of entries that specify further files (including directories) and constitutes a node of the file system. A subdirectory is a directory that is pointed to by a directory one level above it in the file system organization.

The **ls**(1) command is used to list the contents of a directory. When you first log onto the system, you are in your home directory ($HOME). You can move to another directory by using the **cd**(1) command and you can print the name of the current directory by using the **pwd**(1) command. You can also create new directories with the **mkdir**(1) command and remove empty directories with **rmdir**(1).

A directory name is a string of characters that identifies a directory. It can be a simple directory name, the relative path name or the full path name of a directory.

dynamic linking

Dynamic linking refers to the ability to resolve symbolic references at run time. Systems that use dynamic linking can execute processes without resolving unused references. See static linking.

environment

An environment is a collection of resources used to support a function. In the UNIX system, the shell environment is composed of variables whose values define the way you interact with the system. For example, your environment includes your shell prompt string, specifics for backspace and erase characters, and commands for sending output from your terminal to the computer.

An environment variable is a shell variable such as $HOME (which stands for your login directory) or $PATH (which is a list of directories the shell will search through for executable commands) that is part of your environment. When you log in, the system executes programs that create most of the environmental variables that you need for the commands to work. These variables come from **/etc/profile**, a file that defines a general working environment for all users when they log onto a system. In addition, you can define and set variables in your personal **.profile** file, which you create in your

login directory to tailor your own working environment. You can also temporarily set variables at the shell level.

executable file — An executable file is a file that can be processed or executed by the computer without any further translation. That is, when you type in the file name, the commands in the file are executed. An object file that is ready to run (ready to be copied into the address space of a process to run as the code of that process) is an executable file. Files containing shell commands are also executable. A file may be given execute permission by using the **chmod**(1) command. In addition to being ready to run, a file in the UNIX system needs to have execute permission.

exit — A specific system call that causes the termination of a process. The **exit**(2) call will close any open files and clean up most other information and memory which was used by the process.

exit status: return code — An exit status or return code is a code number returned to the shell when a command is terminated that indicates the cause of termination.

exported symbol — A symbol that a shared library defines and makes available outside the library. See imported symbol.

expression — An expression is a mathematical or logical symbol or meaningful combination of symbols. See regular expression.

file — A file is an identifiable collection of information that, in the UNIX system, is a member of a file system. A file is known to the UNIX system as an inode plus the information the inode contains that tells whether the file is a plain file, a special file, or a directory. A plain file may contain text, data, programs or other information that forms a coherent unit. A special file is a hardware device or portion thereof, such as a disk partition. A directory is a type of file that contains the names and inode addresses of other plain, special or directory files.

file and record locking — The phrase "file and record locking" refers to software that protects records in a data file against the possibility of being changed by two users at the same time. Records (or the entire file) may be locked by one authorized user while changes are made. Other users are thus prevented from working with the same record until the changes are completed.

file descriptor — A file descriptor is a number assigned by the operating system to a file when the file is opened by a process. File descriptors 0, 1, and 2 are reserved; file descriptor 0 is reserved for standard input (**stdin**), 1 is reserved for standard output (**stdout**), and 2 is reserved for standard error output (**stderr**).

file system — A UNIX file system is a hierarchical collection of directories and other files that are organized in a tree structure. The base of the structure is the root (**/**) directory; other directories, all subordinate to the root, are branches. The collection of files can be mounted on a block special file. Each file of a file

system appears exactly once in the inode list of the file system and is accessible via a single, unique path from the root directory of the file system.

filter — A filter is a program that reads information from standard input, acts on it in some way, and sends its results to standard output. It is called a filter because it can be used as a data transformer in a pipeline. Filters are different from editors and other commands because filters do not change the contents of a file. Examples of filters are **grep**(1) and **tail**(1), which select and output part of the input; **sort**(1), which sorts the input; and **wc**(1), which counts the number of words, characters, and lines in the input. **sed**(1) and **awk**(1) are also filters but they are called programmable filters or data transformers because a program must be supplied as input in addition to the data to be transformed.

flag — A flag or option is used on a command line to signal a specific condition to a command or to request particular processing. UNIX system flags are usually indicated by a leading hyphen (**-**). The word option is sometimes used interchangeably with flag. Flag is also used as a verb to mean to point out or to draw attention to. See option.

fork — **fork**(2) is a system call that divides a new process into two, the parent and child processes, with separate, but initially identical, text, data, and stack segments. After the duplication, the child (created) process is given a return code of 0 and the parent is given the process id of the newly created child as the return code.

FORTRAN — FORTRAN is an acronym for FORmula TRANslator. FORTRAN is a high-level programming language originally designed for scientific and engineering calculations but now also widely adapted for many business uses.

function — A function is a task done by a computer. In most modern programming languages, programs are made up of functions and procedures which perform small parts of the total job to be done.

header file — A header file is used in programming and in document formatting. In a programming context, a header file is a file that usually contains shared data declarations that are to be copied into source programs as they are compiled. A header file includes symbolic names for constants, macro definitions, external variable references and inclusion of other header files. The name of a header file customarily ends with '.h' (dot-h). Similarly, in a document formatting context, header files contain general formatting macros that describe a common document type and can be used with many different document bodies.

high-level language — A high-level language is a computer programming language such as C, FORTRAN, COBOL, or PASCAL that uses symbols and command statements representing actions the computer is to perform, the exact steps for a machine to follow. A high-level language must be translated into machine

language by a compilation system before a computer can execute it. A characteristic of a high-level language is that each statement usually translates into a series of machine language instructions. The low-level details of the computer's internal organization are left to the compilation system.

host machine — A host machine is the machine on which an **a.out** file is built.

imported symbol — A symbol used but not defined by a shared library. See exported symbol.

interpreted language — An interpreted language is a high-level language that is not translated by a compilation system and stored in an executable object file. The statements of a program in an interpreted language are translated each time the program is executed.

Interprocess Communication — Interprocess Communication describes software that enables independent processes running at the same time, to exchange information through messages, semaphores, or shared memory.

interrupt — An interrupt is a break in the normal flow of a system or program. Interrupts are initiated by signals that are generated by a hardware condition or a peripheral device indicating that a certain event has happened. When the interrupt is recognized by the hardware, an interrupt handling routine is executed. An interrupt character is a character (normally ASCII) that, when typed on a terminal, causes an interrupt. You can usually interrupt UNIX programs by pressing the delete or break keys, by typing Control-d, or by using the **kill**(1) command.

I/O (Input/Output) — I/O is the process by which information enters (input) and leaves (output) the computer system.

kernel — The kernel (comprising 5 to 10 percent of the operating system software) is the basic resident software on which the UNIX system relies. It is responsible for most operating system functions. It schedules and manages the work done by the computer and maintains the file system. The kernel has its own text, data, and stack areas.

lexical analysis — Lexical analysis is the process by which a stream of characters (often comprising a source program) is subdivided into its elementary words and symbols (called tokens). The tokens include the reserved words of the language, its identifiers and constants, and special symbols such as **=**, **:=**, and **;**. Lexical analysis enables you to recognize, for example, that the stream of characters 'print("hello, universe")' is to be analyzed into a series of tokens beginning with the word 'print' and not with, say, the string 'print("h.' In compilers, a lexical analyzer is often called by the compiler's syntactic analyzer or parser, which determines the statements of the program (that is, the proper arrangements of its tokens).

library — A library is an archive file that contains object code and/or files for programs that perform common tasks. The library provides a common source for object code, thus saving space by providing one copy of the code instead of requiring every

program that wants to incorporate the functions in the code to have its own copy. The link editor may select functions and data as needed.

link editor — A link editor, or loader, collects and merges separately compiled object files by linking together object files and the libraries that are referenced into executable load modules. The result is an **a.out** file. Link editing may be done automatically when you use the compilation system to process your programs on the UNIX system, but you can also link edit previously compiled files by using the **ld**(1) command.

magic number — The magic number is contained in the header of an **a.out** file. It indicates what the type of the file is, whether shared or non-shared text, and on which processor the file is executable.

makefile — A makefile is a file that lists dependencies among the source code files of a software product and methods for updating them, usually by recompilation. The **make**(1) command uses the makefile to maintain self-consistent software.

manual page — A manual page, or "man page" in UNIX system jargon, is the repository for the detailed description of a command, a system call, subroutine or other UNIX system component.

null pointer — A null pointer is a C pointer with a value of 0.

object code — Object code is executable machine-language code produced from source code or from other object files by an assembler or a compilation system. An object file is a file of object code and associated data. An object file that is ready to run is an executable file.

optimizer — An optimizer, an optional step in the compilation process, improves the efficiency of the assembly language code. The optimizer reduces the space used by and speeds the execution time of the code.

option — An option is an argument used in a command line to modify program output by modifying the execution of a command. An option is usually one character preceded by a hyphen (**–**). When you do not specify any options, the command will execute according to its default options. For example, in the command line

ls –a –l directory

–a and **–l** are the options that modify the **ls**(1) command to list all **directory** entries, including entries whose names begin with a period (.), in the long format (including permissions, size, and date).

parent process — A parent process occurs when a process is split into two, a parent process and a child process, with separate, but initially identical text, data, and stack segments.

parse — To parse is to analyze a sentence in order identify its components and to determine their grammatical relationship. In computer terminology the word has a similar meaning, but instead of sentences, program statements or commands are

analyzed.

PASCAL

PASCAL is a multipurpose high-level programming language often used to teach programming. It is based on the ALGOL programming language and emphasizes structured programming.

path name

A path name is a way of designating the exact location of a file in a file system. It is made up of a series of directory names that proceed down the hierarchical path of the file system. The directory names are separated by a slash character (/). The last name in the path is either a file or another directory. If the path name begins with a slash, it is called a full path name; the initial slash means that the path begins at the **root** directory.

A path name that does not begin with a slash is known as a relative path name, meaning relative to the present working directory. A relative path name may begin either with a directory name or with two dots followed by a slash (**../**). One that begins with a directory name indicates that the ultimate file or directory is below the present working directory in the hierarchy. One that begins with **../** indicates that the path first proceeds up the hierarchy; **../** is the parent of the present working directory.

permissions

Permissions are a means of defining a right to access a file or directory in the UNIX file system. Permissions are granted separately to you, the owner of the file or directory, your group, and all others. There are three basic permissions:

- ☐ Read permission (r) includes permission to cat, pg, lp, and **cp** a file.
- ☐ Write permission (w) is the permission to change a file.
- ☐ Execute permission (x) is the permission to run an executable file.

Permissions can be changed with the UNIX system **chmod**(1) command.

pipe

A pipe causes the output of one command to be used as the input for the next command so that the two run in sequence. You can do this by preceding each command after the first command with the pipe symbol (|), which indicates that the output from the process on the left should be routed to the process on the right. For example, in the command

who | wc –l,

the output from the **who**(1) command, which lists the users who are logged on to the system, is used as input for the word-count command, **wc**(1), with the **l** option. The result of this pipeline (succession of commands connected by pipes) is the number of people who are currently logged on to the system.

portable

Portability describes the degree of ease with which a program or a library can be moved or ported from one system to another. Portability is desirable because once a program is developed it is used on many systems. If the program writer must change the program in many different ways before it can be distributed to the other systems, time is wasted, and each modification increases the chances for an error.

preprocessor

Preprocessor is a generic name for a program that prepares an input file for another program. For example, **neqn**(1) and **tbl**(1) are preprocessors for **nroff**(1). **grap**(1) is a preprocessor for **pic**(1). **cpp**(1) is a preprocessor for the C compiler.

process

A process is a program that is at some stage of execution. In the UNIX system, it also refers to the execution of a computer environment, including contents of memory, register values, name of the current working directory, status of files, information recorded at login time, etc. Every time you type the name of a file that contains an executable program, you initiate a new process. Shell programs can cause the initiation of many processes because they can contain many command lines.

The process id is a unique system-wide identification number that identifies an active process. The process status command, **ps**(1), prints the process ids of the processes that belong to you.

program

A program is a sequence of instructions or commands that cause the computer to perform a specific task, for example, changing text, making a calculation, or reporting on the status of the system. A subprogram is part of a larger program and can be compiled independently.

regular expression

A regular expression is a string of alphanumeric characters and special characters that describe a character string. It is a shorthand way of describing a pattern to be searched for in a file. The pattern-matching functions of **ed**(1) and **grep**(1), for example, use regular expressions.

routine

A routine is a discrete section of a program to accomplish a set of related tasks

semaphore

In the UNIX system, a semaphore is a sharable short unsigned integer maintained through a family of system calls which include calls for increasing the value of the semaphore, setting its value, and for blocking waiting for its value to reach some value. Semaphores are part of the UNIX system IPC facility.

shared library

Shared libraries include object modules that may be shared among several processes at execution time.

shared memory

Shared memory is an IPC (interprocess communication) facility in which two or more processes can share the same data space.

shell

The shell is the UNIX system program—**sh**(1)—responsible for handling all interaction between you and the system. It is a command language interpreter that understands your commands and causes the computer to act on them. The shell also establishes the environment at your terminal. A shell normally is started for you as part of the login process. Three shells, the Bourne shell, the Korn shell and the C shell, are popular. The shell can also be used as a programming language to write procedures for a variety of tasks.

signal: signal number

A signal is a message that you send to processes or processes send to one another. The most common signals you might send to a process are ones that would cause the process to stop: for example, interrupt, quit, or kill. A signal sent by a running process is usually a sign of an an exceptional occurrence that has caused the process to terminate or divert from the normal flow of control.

source code

Source code is the programming-language version of a program. Before the computer can execute the program, the source code must be translated to machine language by a compilation system or an interpreter.

standard error

Standard error is an output stream from a program. It is normally used to convey error messages. In the UNIX system, the default case is to associate standard error with the user's terminal.

standard input

Standard input is an input stream to a program. In the UNIX system, the default case is to associate standard input with the user's terminal.

standard output

Standard output is an output stream from a program. In the UNIX system, the default case is to associate standard output with the user's terminal.

stdio: standard input-output

stdio(3S) is a collection of functions for formatted and character-by-character input-output at a higher level than the basic read, write, and open operations.

static linking

Static linking refers to the requirement that symbolic references be resolved before run time. See dynamic linking.

stream

- A stream is an open file with buffering provided by the stdio package.
- A stream is a full duplex, processing and data transfer path in the kernel. It implements a connection between a driver in kernel space and a process in user space, providing a general character input/output interface for the user processes.

string
: A string is a contiguous sequence of characters treated as a unit. Strings are normally bounded by white space(s), tab(s), or a character designated as a separator. A string value is a specified group of characters symbolized to the shell by a variable.

strip
: **strip**(1) is a command that removes the symbol table and relocation bits from an executable file.

subroutine
: A subroutine is a program that defines desired operations and may be used in another program to produce the desired operations. A subroutine can be arranged so that control may be transferred to it from a master routine and so that, at the conclusion of the subroutine, control reverts to the master routine. Such a subroutine is usually called a closed subroutine. A single routine may be simultaneously a subroutine with respect to another routine and a master routine with respect to a third.

symbol table
: A symbol table describes information in an object file about the names and functions in that file. The symbol table and relocation bits are used by the link editor and by the debuggers.

symbol value
: The value of a symbol, typically its virtual address, used to resolve references.

syntax

- ☐ Command syntax is the order in which command names, options, option arguments, and operands are put together to form a command on the command line. The command name is first, followed by options and operands. The order of the options and the operands varies from command to command.
- ☐ Language syntax is the set of rules that describe how the elements of a programming language may legally be used.

system call
: A system call is a request by an active process for a service performed by the UNIX system kernel, such as I/O, process creation, etc. All system operations are allocated, initiated, monitored, manipulated, and terminated through system calls. System calls allow you to request the operating system to do some work that the program would not normally be able to do. For example, the **getuid**(2) system call allows you to inspect information that is not normally available since it resides in the operating system's address space.

target machine
: A target machine is the machine on which an **a.out** file is run. While it may be the same machine on which the **a.out** file was produced, the term implies that it may be a different machine.

TCP/IP (Transmission Control Protocol/Internetwork Protocol)
: TCP/IP is a connection-oriented, end-to-end reliable protocol designed to fit into a layered hierarchy of protocols that support multi-network applications. It is the Department of Defense standard in packet networks.

terminal definition

A terminal definition is an entry in the **terminfo**(4) data base that describes the characteristics of a terminal. See **terminfo**(4) and **curses**(3X) in the *Programmer's Reference Manual*.

terminfo

- a group of routines within the curses library that handle certain terminal capabilities. For example, if your terminal has programmable function keys, you can use these routines to program the keys.
- a data base containing the compiled descriptions of many terminals that can be used with **curses**(3X) screen management programs. These descriptions specify the capabilities of a terminal and how it performs various operations — for example, how many lines and columns it has and how its control characters are interpreted. A **curses**(3X) program refers to the data base at run time to obtain the information that it needs about the terminal being used.

See **curses**(3X) in the *Programmer's Reference Manual*. **terminfo**(4) routines can be used in shell programs, as well as C programs.

text symbol

A text symbol is a symbol, usually a function name, that is defined in the **.text** portion of an **a.out** file.

tool

A tool is a program, or package of programs, that performs a given task.

trap

A trap is a condition caused by an error where a process state transition occurs and a signal is sent to the currently running process.

UNIX operating system

The UNIX operating system is a general-purpose, multiuser, interactive, time-sharing operating system developed by AT&T. An operating system is the software on the computer under which all other software runs. The UNIX operating system has two basic parts:

- The kernel is the program that is responsible for most operating system functions. It schedules and manages all the work done by the computer and maintains the file system. It is always running and is invisible to users.
- The shell is the program responsible for handling all interaction between users and the computer. It includes a powerful command language called shell language.

The utility programs or UNIX system commands are executed using the shell, and allow users to communicate with each other, edit and manipulate files, and write and execute programs in several programming languages.

userid

A userid is an integer value, usually associated with a login name, used by the system to identify owners of files and directories. The userid of a process becomes the owner of files created by the process and descendent (forked) processes.

utility

A utility is a standard, permanently available program used to perform routine functions or to assist a programmer in the diagnosis of hardware and software errors, for example, a loader, editor, debugging, or diagnostics package.

variable

- A variable in a computer program is an object whose value may change during the execution of the program, or from one execution to the next.
- A variable in the shell is a name representing a string of characters (a string value).
- A variable normally set only on a command line is called a parameter (positional parameter and keyword parameter).
- A variable may be simply a name to which the user (user-defined variable) or the shell itself may assign string values.

white space

White space is one or more spaces, tabs, or newline characters. White space is normally used to separate strings of characters, and is required to separate the command from its arguments on a command line.

window

A window is a screen within your terminal screen that is set off from the rest of the screen. If you have two windows on your screen, they are independent of each other and the rest of the screen.

The most common way to create windows on a UNIX system is by using the layers capability of the TELETYPE 5620 Dot-Mapped Display. Each window you create with this program has a separate shell running it. Each one of these shells is called a layer.

If you do not have this facility, the **shl**(1) command, which stands for shell layer, offers a function similar to the layers program. You cannot create windows using **shl**(1), but you can start different shells that are independent of each other. Each of the shells you create with **shl**(1) is called a layer.

word

A word is a unit of storage in a computer that is composed of bytes of information. The number of bytes in a word depends on the computer you are using. The MIPS Computers, for example, have 32 bits or 4 bytes per word, and 16 bits or 2 bytes per half word.

Index

Customer Response Card

Your comments, which can assist us in improving our products and our publications, are welcome.

If you wish to reply, be sure to include your name and address, *and the name and part number that appears on the first page of this manual.*

Thank you for your cooperation.

No postage necessary if mailed in the U. S. A.

After writing comments, detach this page and then fold, seal, and mail.

Comments

Name of manual: _____

Part number: _____

MIPS may use and distribute any of the information you supply in any way it believes appropriate without incurring any obligation whatever. You may, of course, continue to use the information you supply.

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED
STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 1659 SUNNYVALE, CA

POSTAGE WILL BE PAID BY ADDRESSEE:

MIPS Computer Systems
928 Arques Avenue
Sunnyvale, CA 94086-9756