## Chapter 15

**15.1** If the write end of the pipe is never closed, the reader never sees an end of file. The pager program blocks forever reading from its standard input.

**15.2** The parent terminates right after writing the last line to the pipe. The read end of the pipe is automatically closed when the parent terminates. But the parent is probably running ahead of the child by one pipe buffer, since the child (the pager program) is waiting for us to look at a page of output. If we're running a shell, such as the Korn shell, with interactive command-line editing enabled, the shell probably changes the terminal mode when our parent terminates and the shell prints a prompt. This undoubtedly interferes with the pager program, which has also modified the terminal mode. (Most pager programs set the terminal to noncanonical mode when awaiting input to proceed to the next page.)

**15.3** The `popen` function returns a file pointer because the shell is executed. But the shell can't execute the nonexistent command, so it prints

```
sh: line 1: ./a.out: No such file or directory
```

on the standard error and terminates with an exit status of 127. `pclose` returns the termination status of the command as it is returned by `waitpid`.

**15.4** When the parent terminates, look at its termination status with the shell. For the Bourne shell, Bourne-again shell, and Korn shell, the command is `echo $?`. The number printed is 128 plus the signal number.

**15.5** First add the declaration

```
FILE    *fpin, *fpout;
```

Then use `fdopen` to associate the pipe descriptors with a standard I/O stream, and set the streams to be line buffered. Do this before the `while` loop that reads from standard input:

```
if ((fpin = fdopen(fd2[0], "r")) == NULL)
    err_sys("fdopen error");
if ((fpout = fdopen(fd1[1], "w")) == NULL)
    err_sys("fdopen error");
if (setvbuf(fpin, NULL, _IOLBF, 0) < 0)
    err_sys("setvbuf error");
if (setvbuf(fpout, NULL, _IOLBF, 0) < 0)
    err_sys("setvbuf error");
```

The `write` and `read` in the `while` loop are replaced with

```
if (fputs(line, fpout) == EOF)
    err_sys("fputs error to pipe");
if (fgets(line, MAXLINE, fpin) == NULL) {
    err_msg("child closed pipe");
    break;
}
```

**15.6** The `system` function calls `wait`, and the first child to terminate is the child generated by `popen`. Since that's not the child that `system` created, it calls `wait` again and blocks until the `sleep` is done. Then `system` returns. When `pclose` calls `wait`, an error is returned, since there are no more children to `wait` for. Then `pclose` returns an error.

**15.7** The `select` function indicates that the descriptor is readable. When we call `read` after all the data has been read, it returns 0 to indicate the end of file. But with `poll` (assuming a STREAMS-based pipe), the `POLLHUP` event is returned, and this event may be returned while there is still data to be read. Once we have read all the data, however, `read` returns 0 to indicate the end of file. After all the data has been read, the `POLLIN` event is not returned, even though we need to issue a `read` to receive the end-of-file notification (the return of 0).

With an output descriptor that refers to a pipe that has been closed by the reader, `select` indicates that the descriptor is writable. But when we call `write`, the `SIGPIPE` signal is generated. If we either ignore this signal or return from its signal handler, `write` returns an error of `EPIPE`. With `poll`, however, if the pipe is STREAMS based, `poll` returns with a `POLLHUP` event for the descriptor.

**15.8** Anything written by the child to standard error appears wherever the parent's standard error would appear. To send standard error back to the parent, include the shell redirection `2>&1` in the *cmdstring*.

**15.9** The `popen` function `fork`s a child, and the child executes the shell. The shell in turn calls `fork`, and the child of the shell executes the command string. When *cmdstring* terminates, the shell is waiting for this to happen. The shell then exits, which is what the `waitpid` in `pclose` is waiting for.

**15.10** The trick is to `open` the FIFO twice: once for reading and once for writing. We never use the descriptor that is opened for writing, but leaving that descriptor open prevents an end of file from being generated when the number of clients goes from 1 to 0. Opening the FIFO twice requires some care, as a nonblocking `open` is required. We have to do a nonblocking, read-only `open` first, followed by a blocking `open` for write-only. (If we tried a nonblocking `open` for write-only first, it would return an error.) We then turn off nonblocking for the read descriptor. Figure C.18 shows the code for this.

```
#include "apue.h"
#include <fcntl.h>

#define FIFO    "temp.fifo"

int
main(void)
{
    int     fdread, fdwrite;

    unlink(FIFO);
    if (mkfifo(FIFO, FILE_MODE) < 0)
        err_sys("mkfifo error");
    if ((fdread = open(FIFO, O_RDONLY | O_NONBLOCK)) < 0)
```

```
        err_sys("open error for reading");
    if ((fdwrite = open(FIFO, O_WRONLY)) < 0)
        err_sys("open error for writing");
    clr_fl(fdread, O_NONBLOCK);
    exit(0);
}
```

**Figure C.18** Opening a FIFO for reading and writing, without blocking

**15.11** Randomly reading a message from an active queue would interfere with the client–server protocol, as either a client request or a server's response would be lost. To read the queue, all that is needed is for the process to know the identifier for the queue and for the queue to allow world-read access.

**15.13** We never store actual addresses in a shared memory segment, since it's possible for the server and all the clients to attach the segment at different addresses. Instead, when a linked list is built in a shared memory segment, the list pointers should be stored as offsets to other objects in the shared memory segment. These offsets are formed by subtracting the start of the shared memory segment from the actual address of the object.

**15.14** Figure C.19 shows the relevant events.

| Parent `i` set to | Child `i` set to | Shared value set to | `update` returns | Comment |
|---|---|---|---|---|
| | | 0 | | initialized by `mmap` |
| | 1 | | | child runs first, then is blocked |
| 0 | | | | parent runs |
| | | 1 | | |
| | | | 0 | then parent is blocked |
| | | 2 | | child resumes |
| | | | 1 | |
| | 3 | | | then child is blocked |
| 2 | | | | parent resumes |
| | | 3 | | |
| | | | 2 | then parent is blocked |
| | | 4 | | |
| | | | 3 | |
| | 5 | | | then child is blocked |
| 4 | | | | parent resumes |

**Figure C.19** Alternation between parent and child in Figure 15.33

## Chapter 16

**16.1** Figure C.20 shows a program that prints the system's byte order.

**16.3** For each endpoint we will be listening on, we need to bind the proper address and record an entry in an `fd_set` structure corresponding to each file descriptor.

```
#include <stdio.h>
#include <stdlib.h>
#include <inttypes.h>

int
main(void)
{
    uint32_t i;
    unsigned char *cp;

    i = 0x04030201;
    cp = (unsigned char *)&i;
    if (*cp == 1)
        printf("little-endian\n");
    else if (*cp == 4)
        printf("big-endian\n");
    else
        printf("who knows?\n");
    exit(0);
}
```

**Figure C.20** Determine byte order on system

We will use `select` to wait for connect requests to arrive on multiple endpoints. Recall from Section 16.4 that a passive endpoint will appear to be readable when a connect request arrives on it. When a connect request does arrive, we will accept the request and process it as before.

**16.5** In the `main` procedure, we need to arrange to catch `SIGCHLD` by calling our `signal` function (Figure 10.18), which will use `sigaction` to install the handler specifying the restartable system call option. Next, we need to remove the call to `waitpid` from our `serve` function. After `forking` the child to service the request, the parent closes the new file descriptor and resumes listening for additional connect requests. Finally, we need a signal handler for `SIGCHLD`, as follows:

```
void
sigchld(int signo)
{
    while (waitpid((pid_t)-1, NULL, WNOHANG) > 0)
        ;
}
```

**16.6** To enable asynchronous socket I/O, we need to establish socket ownership using the `F_SETOWN fcntl` command, and then enable asynchronous signaling using the `FIOASYNC ioctl` command. To disable asynchronous socket I/O, we simply need to disable asynchronous signaling. The reason we mix `fcntl` and `ioctl` commands is to find the methods that are most portable. The code is shown in Figure C.21.

```
#include "apue.h"
#include <errno.h>
#include <fcntl.h>
#include <sys/socket.h>
#include <sys/ioctl.h>
#if defined(BSD) || defined(MACOS) || defined(SOLARIS)
#include <sys/filio.h>
#endif

int
setasync(int sockfd)
{
    int n;

    if (fcntl(sockfd, F_SETOWN, getpid()) < 0)
        return(-1);
    n = 1;
    if (ioctl(sockfd, FIOASYNC, &n) < 0)
        return(-1);
    return(0);
}

int
clrasync(int sockfd)
{
    int n;

    n = 0;
    if (ioctl(sockfd, FIOASYNC, &n) < 0)
        return(-1);
    return(0);
}
```

**Figure C.21** Enable and disable asynchronous socket I/O

## Chapter 17

**17.3** A *declaration* specifies the attributes (such as the data type) of a set of identifiers. If the declaration also causes storage to be allocated, it is called a *definition*.

In the `opend.h` header, we declare the three global variables with the `extern` storage class. These declarations do not cause storage to be allocated for the variables. In the `main.c` file, we define the three global variables. Sometimes, we'll also initialize a global variable when we define it, but we typically let the C default apply.

**17.5** Both `select` and `poll` return the number of ready descriptors as the value of the function. The loop that goes through the `client` array can terminate when the number of ready descriptors have been processed.

## Chapter 18

**18.1** Note that you have to terminate the `reset` command with a line feed character, not a return, since the terminal is in noncanonical mode.

**18.2** It builds a table for each of the 128 characters and sets the high-order bit (the parity bit) according to the user's specification. It then uses 8-bit I/O, handling the parity generation itself.

**18.3** If you happen to be on a windowing terminal, you don't need to log in twice. You can do this experiment between two separate windows. Under Solaris, execute `stty -a` with standard input redirected from the terminal window running `vi`. This shows that `vi` sets MIN to 1 and TIME to 1. A call to `read` will wait for at least one character to be typed, but after that character is entered, `read` waits only one-tenth of a second for additional characters before returning.

## Chapter 19

**19.1** Both servers, `telnetd` and `rlogind`, run with superuser privileges, so their calls to `chown` and `chmod` succeed.

**19.3** Execute `pty -n stty -a` to prevent the slave's `termios` structure and `winsize` structure from being initialized.

**19.5** Unfortunately, the `F_SETFL` command of `fcntl` doesn't allow the read–write status to be changed.

**19.6** There are three process groups: (1) the login shell, (2) the `pty` parent and child, and (3) the `cat` process. The first two process groups constitute a session with the login shell as the session leader. The second session contains only the `cat` process. The first process group (the login shell) is a background process group, and the other two are foreground process groups.

**19.7** First, `cat` terminates when it receives the end of file from its line discipline. This causes the PTY slave to terminate, which causes the PTY master to terminate. This in turn generates an end of file for the `pty` parent that's reading from the PTY master. The parent sends `SIGTERM` to the child, so the child terminates next. (The child doesn't catch this signal.) Finally, the parent calls `exit(0)` at the end of the `main` function.

The relevant output from the program shown in Figure 8.29 is

```
cat     e =   270, chars =   274, stat =   0:
pty     e =   262, chars =    40, stat =  15: F     X
pty     e =   288, chars =   188, stat =   0:
```

**19.8** This can be done with the shell's `echo` command and the `date`(1) command, all in a subshell:

```
#!/bin/sh
( echo "Script started on " `date`;
  pty "${SHELL:-/bin/sh}";
  echo "Script done on " `date` ) | tee typescript
```

**19.9** The line discipline above the PTY slave has echo enabled, so whatever `pty` reads on its standard input and writes to the PTY master gets echoed by default. This echoing is done by the line discipline module above the slave even though the program (`ttyname`) never reads the data.

## Chapter 20

**20.1** Our conservative locking in `_db_dodelete` is to avoid race conditions with `db_nextrec`. If the call to `_db_writedat` were not protected with a write lock, it would be possible to erase the data record while `db_nextrec` was reading that data record: `db_nextrec` would read an index record, determine that it was not blank, and then read the data record, which could be erased by `_db_dodelete` between the calls to `_db_readidx` and `_db_readdat` in `db_nextrec`.

**20.2** Assume that `db_nextrec` calls `_db_readidx`, which reads the key into the index buffer for the process. This process is then stopped by the kernel, and another process runs. This other process calls `db_delete`, and the record being read by the other process is deleted. Both its key and its data are rewritten in the two files as all blanks. The first process resumes and calls `_db_readdat` (from `db_nextrec`) and reads the all-blank data record. The read lock by `db_nextrec` allows it to do the read of the index record, followed by the read of the data record, as an atomic operation (with regard to other cooperating processes using the same database).

**20.3** With mandatory locking, other readers and writers are affected. Other reads and writes are blocked by the kernel until the locks placed by `_db_writeidx` and `_db_writedat` are removed.

**20.5** By writing the data record before the index record, we protect ourselves from generating a corrupt record if the process should be killed in between the two writes. If the process were to write the index record first, but be killed before writing the data record, then we'd have a valid index record that pointed to invalid data.

## Chapter 21

**21.5** Here are some hints. There are two places to check for queued jobs: the printer spooling daemon's queue and the network printer's internal queue. Take care to prevent one user from being able to cancel someone else's print job. Of course, the superuser should be able to cancel any job.

# Index

The function subentries labeled "definition of" point to where the function prototype appears and, when applicable, to the source code for the function. Functions defined in the text that are used in later examples, such as the `set_fl` function in Figure 3.11, are included in this index. The definitions of external functions that are part of the larger examples (Chapters 17, 19, 20, and 21) are also included in this index, to help in going through these larger examples. Also, significant functions and constants that occur in any of the examples in the text, such as `select` and `poll`, are also included in this index. Trivial functions that occur in almost every example, such as `exit`, are not referenced when they occur in examples.