```
60  long                    nextjob;
61  pthread_mutex_t         joblock = PTHREAD_MUTEX_INITIALIZER;
62  pthread_cond_t          jobwait = PTHREAD_COND_INITIALIZER;
63  /*
64   * Function prototypes.
65   */
66  void        init_request(void);
67  void        init_printer(void);
68  void        update_jobno(void);
69  long        get_newjobno(void);
70  void        add_job(struct printreq *, long);
71  void        replace_job(struct job *);
72  void        remove_job(struct job *);
73  void        build_qonstart(void);
74  void        *client_thread(void *);
75  void        *printer_thread(void *);
76  void        *signal_thread(void *);
77  ssize_t     readmore(int, char **, int, int *);
78  int         printer_status(int, struct job *);
79  void        add_worker(pthread_t, int);
80  void        kill_workers(void);
81  void        client_cleanup(void *);
82  /*
83   * Main print server thread.  Accepts connect requests from
84   * clients and spawns additional threads to service requests.
85   *
86   * LOCKING: none.
87   */
88  int
89  main(int argc, char *argv[])
90  {
91      pthread_t           tid;
92      struct addrinfo     *ailist, *aip;
93      int                 sockfd, err, i, n, maxfd;
94      char                *host;
95      fd_set              rendezvous, rset;
96      struct sigaction    sa;
97      struct passwd       *pwdp;
```

[60–62] `nextjob` is the ID of the next print job to be received. The `joblock` mutex protects the linked list of jobs, as well as the condition represented by the `jobwait` condition variable.

[63–81] We declare the function prototypes for the remaining functions in this file. Doing this up front allows us to place the functions in the file without worrying about the order in which each is called.

[82–97] The `main` function for the printer spooling daemon has two tasks to perform: initialize the daemon and then process connect requests from clients.

```
 98      if (argc != 1)
 99          err_quit("usage: printd");
100      daemonize("printd");

101      sigemptyset(&sa.sa_mask);
102      sa.sa_flags = 0;
103      sa.sa_handler = SIG_IGN;
104      if (sigaction(SIGPIPE, &sa, NULL) < 0)
105          log_sys("sigaction failed");
106      sigemptyset(&mask);
107      sigaddset(&mask, SIGHUP);
108      sigaddset(&mask, SIGTERM);
109      if ((err = pthread_sigmask(SIG_BLOCK, &mask, NULL)) != 0)
110          log_sys("pthread_sigmask failed");
111      init_request();
112      init_printer();

113    #ifdef _SC_HOST_NAME_MAX
114      n = sysconf(_SC_HOST_NAME_MAX);
115      if (n < 0)  /* best guess */
116    #endif
117          n = HOST_NAME_MAX;

118      if ((host = malloc(n)) == NULL)
119          log_sys("malloc error");
120      if (gethostname(host, n) < 0)
121          log_sys("gethostname error");
```

[98–100] The daemon doesn't have any options, so if `argc` is not 1, we call `err_quit` to print an error message and exit. We call the `daemonize` function from Figure 13.1 to become a daemon. After this point, we can't print error messages to standard error; we need to log them instead.

[101–112] We arrange to ignore `SIGPIPE`. We will be writing to socket file descriptors, and we don't want a write error to trigger `SIGPIPE`, because the default action is to kill the process. Next, we set the signal mask of the thread to include `SIGHUP` and `SIGTERM`. All threads we create will inherit this signal mask. We'll use `SIGHUP` to tell the daemon to reread the configuration file and `SIGTERM` to tell the daemon to clean up and exit gracefully. We call `init_request` to initialize the job requests and ensure that only one copy of the daemon is running, and we call `init_printer` to initialize the printer information (we'll see both of these functions shortly).

[113–121] If the platform defines the `_SC_HOST_NAME_MAX` symbol, we call `sysconf` to get the maximum size of a host name. If `sysconf` fails or the limit is undefined, we use `HOST_NAME_MAX` as a best guess. Sometimes, this is defined for us by the platform, but if it isn't, we chose our own value in `print.h`. We allocate memory to hold the host name and call `gethostname` to retrieve it.

```
122     if ((err = getaddrlist(host, "print", &ailist)) != 0) {
123         log_quit("getaddrinfo error: %s", gai_strerror(err));
124         exit(1);
125     }
126     FD_ZERO(&rendezvous);
127     maxfd = -1;
128     for (aip = ailist; aip != NULL; aip = aip->ai_next) {
129         if ((sockfd = initserver(SOCK_STREAM, aip->ai_addr,
130           aip->ai_addrlen, QLEN)) >= 0) {
131             FD_SET(sockfd, &rendezvous);
132             if (sockfd > maxfd)
133                 maxfd = sockfd;
134         }
135     }
136     if (maxfd == -1)
137         log_quit("service not enabled");
138     pwdp = getpwnam("lp");
139     if (pwdp == NULL)
140         log_sys("can't find user lp");
141     if (pwdp->pw_uid == 0)
142         log_quit("user lp is privileged");
143     if (setuid(pwdp->pw_uid) < 0)
144         log_sys("can't change IDs to user lp");
```

[122–135] Next, we try to find the network address that the daemon is supposed to use to provide printer spooling service. We clear the `rendezvous fd_set` variable that we will use with `select` to wait for client connect requests. We initialize the maximum file descriptor to –1 so that the first file descriptor we allocate is sure to be greater than `maxfd`. For each network address on which we need to provide service, we call `initserver` (from Figure 16.20) to allocate and initialize a socket. If `initserver` succeeds, we add the file descriptor to the `fd_set`; if it is greater than the maximum, we set `maxfd` equal to the socket file descriptor.

[136–137] If `maxfd` is still –1 after stepping through the list of `addrinfo` structures, we can't enable the printer spooling service, so we log a message and exit.

[138–144] Our daemon needs superuser privileges to bind a socket to a reserved port number. Now that this is done, we can lower its privileges by changing its user ID to the one associated with user `lp` (recall the security discussion in Section 21.4). We want to follow the principles of least privilege to avoid exposing the system to any potential vulnerabilities in the daemon. We call `getpwnam` to find the password entry associated with user `lp`. If no such user account exists, or if it exists with the same user ID as the superuser, we log a message and exit. Otherwise, we call `setuid` to change both the real and effective user IDs to the user ID for `lp`. To avoid exposing our system, we choose to provide no service at all if we can't reduce our privileges.

```
145     pthread_create(&tid, NULL, printer_thread, NULL);
146     pthread_create(&tid, NULL, signal_thread, NULL);
147     build_qonstart();
148     log_msg("daemon initialized");
149     for (;;) {
150         rset = rendezvous;
151         if (select(maxfd+1, &rset, NULL, NULL, NULL) < 0)
152             log_sys("select failed");
153         for (i = 0; i <= maxfd; i++) {
154             if (FD_ISSET(i, &rset)) {
155                 /*
156                  * Accept the connection and handle
157                  * the request.
158                  */
159                 sockfd = accept(i, NULL, NULL);
160                 if (sockfd < 0)
161                     log_ret("accept failed");
162                 pthread_create(&tid, NULL, client_thread,
163                   (void *)sockfd);
164             }
165         }
166     }
167     exit(1);
168 }
```

[145–148] We call `pthread_create` twice to create one thread to handle signals and one thread to communicate with the printer. (By restricting printer communication to one thread, we can simplify the locking of the printer-related data structures.) Then we call `build_qonstart` to search the directories in `/var/spool/printer` for any pending jobs. For each job that we find on disk, we will create a structure to let the printer thread know that it should send the file to the printer. At this point, we are done setting up the daemon, so we log a message to indicate that the daemon has initialized successfully.

[149–168] We copy the `rendezvous` `fd_set` structure to `rset` and call `select` to wait for one of the file descriptors to become readable. We have to copy `rendezvous`, because `select` will modify the `fd_set` structure that we pass to it to include only those file descriptors that satisfy the event. Since the sockets have been initialized for use by a server, a readable file descriptor means that a connect request is pending. After `select` returns, we check `rset` for a readable file descriptor. If we find one, we call `accept` to accept the connection. If this fails, we log a message and continue checking for more readable file descriptors. Otherwise, we create a thread to handle the client connection. The `main` thread loops, farming requests out to other threads for processing, and should never reach the `exit` statement.

```
169  /*
170   * Initialize the job ID file.  Use a record lock to prevent
171   * more than one printer daemon from running at a time.
172   *
173   * LOCKING: none, except for record-lock on job ID file.
174   */
175  void
176  init_request(void)
177  {
178      int     n;
179      char    name[FILENMSZ];

180      sprintf(name, "%s/%s", SPOOLDIR, JOBFILE);
181      jobfd = open(name, O_CREAT|O_RDWR, S_IRUSR|S_IWUSR);
182      if (write_lock(jobfd, 0, SEEK_SET, 0) < 0)
183          log_quit("daemon already running");

184      /*
185       * Reuse the name buffer for the job counter.
186       */
187      if ((n = read(jobfd, name, FILENMSZ)) < 0)
188          log_sys("can't read job file");
189      if (n == 0)
190          nextjob = 1;
191      else
192          nextjob = atol(name);
193  }
```

[169–183] The `init_request` function does two things: it places a record lock on the job file, `/var/spool/printer/jobno`, and it reads the file to determine the next job number to assign. We place a write lock on the entire file to indicate that the daemon is running. If someone tries to start additional copies of the printer spooling daemon while one is already running, these additional daemons will fail to obtain the write lock and will exit. Thus, only one copy of the daemon can be running at a time. (Recall that we used this technique in Figure 13.6; we discussed the `write_lock` macro in Section 14.3.)

[184–193] The job file contains an ASCII integer string representing the next job number. If the file was just created and therefore is empty, we set `nextjob` to 1. Otherwise, we use `atol` to convert the string to an integer and use this as the next job number. We leave `jobfd` open to the job file so that we can update the job number as jobs are created. We can't close the file, because this would release the write lock that we've placed on it.

On a system where a long integer is 64 bits wide, we need a buffer at least 21 bytes in size to fit a string representing the largest possible long integer. We are safe reusing the filename buffer, because `FILENMSZ` is defined to be 64 in `print.h`.

```
194  /*
195   * Initialize printer information.
196   *
197   * LOCKING: none.
198   */
199  void
200  init_printer(void)
201  {
202    printer = get_printaddr();
203    if (printer == NULL) {
204        log_msg("no printer device registered");
205        exit(1);
206    }
207    printer_name = printer->ai_canonname;
208    if (printer_name == NULL)
209        printer_name = "printer";
210    log_msg("printer is %s", printer_name);
211  }

212  /*
213   * Update the job ID file with the next job number.
214   *
215   * LOCKING: none.
216   */
217  void
218  update_jobno(void)
219  {
220    char    buf[32];

221    lseek(jobfd, 0, SEEK_SET);
222    sprintf(buf, "%ld", nextjob);
223    if (write(jobfd, buf, strlen(buf)) < 0)
224        log_sys("can't update job file");
225  }
```

[194–211] The `init_printer` function is used to set the printer name and address. We get the printer address by calling `get_printaddr` (from `util.c`). If this fails, we log a message and exit. We can't do this by calling `log_sys`, because `get_printaddr` can fail without setting `errno`. When it fails and does set `errno`, however, `get_printaddr` logs its own error message. We set the printer name to the `ai_canonname` field in the `addrinfo` structure. If this field is null, we set the printer name to a default value of `printer`. Note that we log the name of the printer we are using to aid administrators in diagnosing problems with the spooling system.

[212–225] The `update_jobno` function is used to write the next job number to the job file, `/var/spool/printer/jobno`. First, we seek to the beginning of the file. Then we convert the integer job number into a string and write it to the file. If the write fails, we log an error message and exit.

```
226  /*
227   * Get the next job number.
228   *
229   * LOCKING: acquires and releases joblock.
230   */
231  long
232  get_newjobno(void)
233  {
234    long    jobid;

235    pthread_mutex_lock(&joblock);
236    jobid = nextjob++;
237    if (nextjob <= 0)
238        nextjob = 1;
239    pthread_mutex_unlock(&joblock);
240    return(jobid);
241  }

242  /*
243   * Add a new job to the list of pending jobs.  Then signal
244   * the printer thread that a job is pending.
245   *
246   * LOCKING: acquires and releases joblock.
247   */
248  void
249  add_job(struct printreq *reqp, long jobid)
250  {
251    struct job  *jp;

252    if ((jp = malloc(sizeof(struct job))) == NULL)
253        log_sys("malloc failed");
254    memcpy(&jp->req, reqp, sizeof(struct printreq));
```

[226–241] The `get_newjobno` function is used to get the next job number. We first lock the `joblock` mutex. We increment the `nextjob` variable and handle the case where it wraps around. Then we unlock the mutex and return the value `nextjob` had before we incremented it. Multiple threads can call `get_newjobno` at the same time; we need to serialize access to the next job number so that each thread gets a unique job number. (Refer to Figure 11.9 to see what could happen if we don't serialize the threads in this case.)

[242–254] The `add_job` function is used to add a new print request to the end of the list of pending print jobs. We start by allocating space for the `job` structure. If this fails, we log a message and exit. At this point, the print request is stored safely on disk; when the printer spooling daemon is restarted, it will pick the request up. After we allocate memory for the new job, we copy the request structure from the client into the job structure. Recall from `print.h` that a `job` structure consists of a pair of list pointers, a job ID, and a copy of the `printreq` structure sent to us by the client `print` command.

```
255     jp->jobid = jobid;
256     jp->next = NULL;
257     pthread_mutex_lock(&joblock);
258     jp->prev = jobtail;
259     if (jobtail == NULL)
260         jobhead = jp;
261     else
262         jobtail->next = jp;
263     jobtail = jp;
264     pthread_mutex_unlock(&joblock);
265     pthread_cond_signal(&jobwait);
266   }

267   /*
268    * Replace a job back on the head of the list.
269    *
270    * LOCKING: acquires and releases joblock.
271    */
272   void
273   replace_job(struct job *jp)
274   {
275     pthread_mutex_lock(&joblock);
276     jp->prev = NULL;
277     jp->next = jobhead;
278     if (jobhead == NULL)
279         jobtail = jp;
280     else
281         jobhead->prev = jp;
282     jobhead = jp;
283     pthread_mutex_unlock(&joblock);
284   }
```

[255–266] We save the job ID and lock the `joblock` mutex to gain exclusive access to the linked list of print jobs. We are about to add the new job structure to the end of the list. We set the new structure's previous pointer to the last job on the list. If the list is empty, we set `jobhead` to point to the new structure. Otherwise, we set the next pointer in the last entry on the list to point to the new structure. Then we set `jobtail` to point to the new structure. We unlock the mutex and signal the printer thread that another job is available.

[267–284] The `replace_job` function is used to insert a job at the head of the pending job list. We acquire the `joblock` mutex, set the previous pointer in the `job` structure to null, and set the next pointer in the `job` structure to point to the head of the list. If the list is empty, we set `jobtail` to point to the `job` structure we are replacing. Otherwise, we set the previous pointer in the first `job` structure on the list to point to the `job` structure we are replacing. Then we set the `jobhead` pointer to the `job` structure we are replacing. Finally, we release the `joblock` mutex.

```
285   /*
286    * Remove a job from the list of pending jobs.
287    *
288    * LOCKING: caller must hold joblock.
289    */
290   void
291   remove_job(struct job *target)
292   {
293     if (target->next != NULL)
294         target->next->prev = target->prev;
295     else
296         jobtail = target->prev;
297     if (target->prev != NULL)
298         target->prev->next = target->next;
299     else
300         jobhead = target->next;
301   }

302   /*
303    * Check the spool directory for pending jobs on start-up.
304    *
305    * LOCKING: none.
306    */
307   void
308   build_qonstart(void)
309   {
310     int             fd, err, nr;
311     long            jobid;
312     DIR             *dirp;
313     struct dirent   *entp;
314     struct printreq req;
315     char            dname[FILENMSZ], fname[FILENMSZ];

316     sprintf(dname, "%s/%s", SPOOLDIR, REQDIR);
317     if ((dirp = opendir(dname)) == NULL)
318         return;
```

[285–301] `remove_job` removes a job from the list of pending jobs given a pointer to the job to be removed. The caller must already hold the `joblock` mutex. If the next pointer is non-null, we set the next entry's previous pointer to the target's previous pointer. Otherwise, the entry is the last one on the list, so we set `jobtail` to the target's previous pointer. If the target's previous pointer is non-null, we set the previous entry's next pointer equal to the target's next pointer. Otherwise, this is the first entry in the list, so we set `jobhead` to point to the next entry in the list after the target.

[302–318] When the daemon starts, it calls `build_qonstart` to build an in-memory list of print jobs from the disk files stored in `/var/spool/printer/reqs`. If we can't open the directory, no print jobs are pending, so we return.

```
319     while ((entp = readdir(dirp)) != NULL) {
320         /*
321          * Skip "." and ".."
322          */
323         if (strcmp(entp->d_name, ".") == 0 ||
324           strcmp(entp->d_name, "..") == 0)
325             continue;
326         /*
327          * Read the request structure.
328          */
329         sprintf(fname, "%s/%s/%s", SPOOLDIR, REQDIR, entp->d_name);
330         if ((fd = open(fname, O_RDONLY)) < 0)
331             continue;
332         nr = read(fd, &req, sizeof(struct printreq));
333         if (nr != sizeof(struct printreq)) {
334             if (nr < 0)
335                 err = errno;
336             else
337                 err = EIO;
338             close(fd);
339             log_msg("build_qonstart: can't read %s: %s",
340               fname, strerror(err));
341             unlink(fname);
342             sprintf(fname, "%s/%s/%s", SPOOLDIR, DATADIR,
343               entp->d_name);
344             unlink(fname);
345             continue;
346         }
347         jobid = atol(entp->d_name);
348         log_msg("adding job %ld to queue", jobid);
349         add_job(&req, jobid);
350     }
351     closedir(dirp);
352 }
```

[319–325] We read each entry in the directory, one at a time. We skip the entries for dot and dot-dot.

[326–346] For each entry, we create the full pathname of the file and open it for reading. If the open call fails, we just skip the file. Otherwise, we read the `printreq` structure stored in it. If we don't read the entire structure, we close the file, log a message, and unlink the file. Then we create the full pathname of the corresponding data file and unlink it, too.

[347–352] If we were able to read a complete `printreq` structure, we convert the filename into a job ID (the name of the file is its job ID), log a message, and then add the request to the list of pending print jobs. When we are done reading the directory, `readdir` will return `NULL`, and we close the directory and return.

```
353  /*
354   * Accept a print job from a client.
355   *
356   * LOCKING: none.
357   */
358  void *
359  client_thread(void *arg)
360  {
361    int                 n, fd, sockfd, nr, nw, first;
362    long                jobid;
363    pthread_t           tid;
364    struct printreq     req;
365    struct printresp    res;
366    char                name[FILENMSZ];
367    char                buf[IOBUFSZ];

368    tid = pthread_self();
369    pthread_cleanup_push(client_cleanup, (void *)tid);
370    sockfd = (int)arg;
371    add_worker(tid, sockfd);

372    /*
373     * Read the request header.
374     */
375    if ((n = treadn(sockfd, &req, sizeof(struct printreq), 10)) !=
376      sizeof(struct printreq)) {
377        res.jobid = 0;
378        if (n < 0)
379            res.retcode = htonl(errno);
380        else
381            res.retcode = htonl(EIO);
382        strncpy(res.msg, strerror(res.retcode), MSGLEN_MAX);
383        writen(sockfd, &res, sizeof(struct printresp));
384        pthread_exit((void *)1);
385    }
```

[353–371] The `client_thread` is spawned from the `main` thread when a connect request is accepted. Its job is to receive the file to be printed from the client `print` command. We create a separate thread for each client print request.

The first thing we do is install a thread cleanup handler (see Section 11.5 for a discussion of thread cleanup handlers). The cleanup handler is `client_cleanup`, which we will see later. It takes a single argument: our thread ID. Then we call `add_worker` to create a `worker_thread` structure and add it to the list of active client threads.

[372–385] At this point, we are done with the thread's initialization tasks, so we read the request header from the client. If the client sends less than we expect or we encounter an error, we respond with a message indicating the reason for the error and call `pthread_exit` to terminate the thread.

```
386     req.size = ntohl(req.size);
387     req.flags = ntohl(req.flags);
388     /*
389      * Create the data file.
390      */
391     jobid = get_newjobno();
392     sprintf(name, "%s/%s/%ld", SPOOLDIR, DATADIR, jobid);
393     if ((fd = creat(name, FILEPERM)) < 0) {
394         res.jobid = 0;
395         if (n < 0)
396             res.retcode = htonl(errno);
397         else
398             res.retcode = htonl(EIO);
399         log_msg("client_thread: can't create %s: %s", name,
400           strerror(res.retcode));
401         strncpy(res.msg, strerror(res.retcode), MSGLEN_MAX);
402         writen(sockfd, &res, sizeof(struct printresp));
403         pthread_exit((void *)1);
404     }
405     /*
406      * Read the file and store it in the spool directory.
407      */
408     first = 1;
409     while ((nr = tread(sockfd, buf, IOBUFSZ, 20)) > 0) {
410         if (first) {
411             first = 0;
412             if (strncmp(buf, "%!PS", 4) != 0)
413                 req.flags |= PR_TEXT;
414         }
```

[386–404] We convert the integer fields in the request header to host byte order and call `get_newjobno` to reserve the next job ID for this print request. We create the job data file, named `/var/spool/printer/data/`*jobid*, where *jobid* is the request's job ID. We use permissions that prevent others from being able read the files (`FILEPERM` is defined as `S_IRUSR|S_IWUSR` in `print.h`). If we can't create the file, we log an error message, send a failure response back to the client, and terminate the thread by calling `pthread_exit`.

[405–414] We read the file contents from the client, with the intent of writing the contents out to our private copy of the data file. But before we write anything, we need to check if this is a PostScript file the first time through the loop. If the file doesn't begin with the pattern `%!PS`, we can assume that the file is plaintext, so we set the `PR_TEXT` flag in the request header in this case. (Recall that the client can also set this flag if the `-t` flag is included when the `print` command is executed.) Although PostScript programs are not required to start with the pattern `%!PS`, the document formatting guidelines (Adobe Systems [1999]) strongly recommends that they do.

```
415             nw = write(fd, buf, nr);
416             if (nw != nr) {
417                 if (nw < 0)
418                     res.retcode = htonl(errno);
419                 else
420                     res.retcode = htonl(EIO);
421                 log_msg("client_thread: can't write %s: %s", name,
422                   strerror(res.retcode));
423                 close(fd);
424                 strncpy(res.msg, strerror(res.retcode), MSGLEN_MAX);
425                 writen(sockfd, &res, sizeof(struct printresp));
426                 unlink(name);
427                 pthread_exit((void *)1);
428             }
429         }
430         close(fd);
431         /*
432          * Create the control file.
433          */
434         sprintf(name, "%s/%s/%ld", SPOOLDIR, REQDIR, jobid);
435         fd = creat(name, FILEPERM);
436         if (fd < 0) {
437             res.jobid = 0;
438             if (n < 0)
439                 res.retcode = htonl(errno);
440             else
441                 res.retcode = htonl(EIO);
442             log_msg("client_thread: can't create %s: %s", name,
443               strerror(res.retcode));
444             strncpy(res.msg, strerror(res.retcode), MSGLEN_MAX);
445             writen(sockfd, &res, sizeof(struct printresp));
446             sprintf(name, "%s/%s/%ld", SPOOLDIR, DATADIR, jobid);
447             unlink(name);
448             pthread_exit((void *)1);
449         }
```

[415–430] We write the data that we read from the client to the data file. If `write` fails, we log an error message, close the file descriptor for the data file, send an error message back to the client, delete the data file, and terminate the thread by calling `pthread_exit`. Note that we do not explicitly close the socket file descriptor. This is done for us by our thread cleanup handler as part of the processing that occurs when we call `pthread_exit`.

When we receive all the data to be printed, we close the file descriptor for the data file.

[431–449] Next, we create a file, `/var/spool/printer/reqs/`*jobid*, to remember the print request. If this fails, we log an error message, send an error response to the client, remove the data file, and terminate the thread.

```
450     nw = write(fd, &req, sizeof(struct printreq));
451     if (nw != sizeof(struct printreq)) {
452         res.jobid = 0;
453         if (nw < 0)
454             res.retcode = htonl(errno);
455         else
456             res.retcode = htonl(EIO);
457         log_msg("client_thread: can't write %s: %s", name,
458           strerror(res.retcode));
459         close(fd);
460         strncpy(res.msg, strerror(res.retcode), MSGLEN_MAX);
461         writen(sockfd, &res, sizeof(struct printresp));
462         unlink(name);
463         sprintf(name, "%s/%s/%ld", SPOOLDIR, DATADIR, jobid);
464         unlink(name);
465         pthread_exit((void *)1);
466     }
467     close(fd);
468     /*
469      * Send response to client.
470      */
471     res.retcode = 0;
472     res.jobid = htonl(jobid);
473     sprintf(res.msg, "request ID %ld", jobid);
474     writen(sockfd, &res, sizeof(struct printresp));
475     /*
476      * Notify the printer thread, clean up, and exit.
477      */
478     log_msg("adding job %ld to queue", jobid);
479     add_job(&req, jobid);
480     pthread_cleanup_pop(1);
481     return((void *)0);
482 }
```

[450–466] We write the `printreq` structure to the control file. On error, we log a message, close the descriptor for the control file, send a failure response back to the client, remove the data and control files, and terminate the thread.

[467–474] We close the file descriptor for the control file and send a message containing the job ID and a successful status (`retcode` set to 0) back to the client.

[475–482] We call `add_job` to add the received job to the list of pending print jobs and call `pthread_cleanup_pop` to complete the cleanup processing. The thread terminates when we return.

Note that before the thread exits, we must close any file descriptors we no longer need. Unlike process termination, file descriptors are not closed automatically when a thread ends if other threads exist in the process. If we didn't close unneeded file descriptors, we'd eventually run out of resources.

```
483   /*
484   * Add a worker to the list of worker threads.
485   *
486   * LOCKING: acquires and releases workerlock.
487   */
488   void
489   add_worker(pthread_t tid, int sockfd)
490   {
491     struct worker_thread    *wtp;

492     if ((wtp = malloc(sizeof(struct worker_thread))) == NULL) {
493         log_ret("add_worker: can't malloc");
494         pthread_exit((void *)1);
495     }
496     wtp->tid = tid;
497     wtp->sockfd = sockfd;
498     pthread_mutex_lock(&workerlock);
499     wtp->prev = NULL;
500     wtp->next = workers;
501     if (workers == NULL)
502         workers = wtp;
503     else
504         workers->prev = wtp;
505     pthread_mutex_unlock(&workerlock);
506   }

507   /*
508    * Cancel (kill) all outstanding workers.
509    *
510    * LOCKING: acquires and releases workerlock.
511    */
512   void
513   kill_workers(void)
514   {
515     struct worker_thread    *wtp;

516     pthread_mutex_lock(&workerlock);
517     for (wtp = workers; wtp != NULL; wtp = wtp->next)
518         pthread_cancel(wtp->tid);
519     pthread_mutex_unlock(&workerlock);
520   }
```

[483–506] `add_worker` adds a `worker_thread` structure to the list of active threads. We allocate memory for the structure, initialize it, lock the `workerlock` mutex, add the structure to the head of the list, and unlock the mutex.

[507–520] The `kill_workers` function walks the list of worker threads and cancels each one. We hold the `workerlock` mutex while we walk the list. Recall that `pthread_cancel` merely schedules a thread for cancellation; actual cancellation happens when each thread reaches the next cancellation point.

```
521  /*
522   * Cancellation routine for the worker thread.
523   *
524   * LOCKING: acquires and releases workerlock.
525   */
526  void
527  client_cleanup(void *arg)
528  {
529      struct worker_thread    *wtp;
530      pthread_t               tid;

531      tid = (pthread_t)arg;
532      pthread_mutex_lock(&workerlock);
533      for (wtp = workers; wtp != NULL; wtp = wtp->next) {
534          if (wtp->tid == tid) {
535              if (wtp->next != NULL)
536                  wtp->next->prev = wtp->prev;
537              if (wtp->prev != NULL)
538                  wtp->prev->next = wtp->next;
539              else
540                  workers = wtp->next;
541              break;
542          }
543      }
544      pthread_mutex_unlock(&workerlock);
545      if (wtp != NULL) {
546          close(wtp->sockfd);
547          free(wtp);
548      }
549  }
```

[521–543] The `client_cleanup` function is the thread cleanup handler for the worker threads that communicate with client commands. This function is called when the thread calls `pthread_exit`, calls `pthread_cleanup_pop` with a nonzero argument, or responds to a cancellation request. The argument is the thread ID of the thread terminating.

We lock the `workerlock` mutex and search the list of worker threads until we find a matching thread ID. When we find a match, we remove the worker thread structure from the list and stop the search.

[544–549] We unlock the `workerlock` mutex, close the socket file descriptor used by the thread to communicate with the client, and free the memory backing the `worker_thread` structure.

Since we try to acquire the `workerlock` mutex, if a thread reaches a cancellation point while the `kill_workers` function is still walking the list, we will have to wait until `kill_workers` releases the mutex before we can proceed.

```
550  /*
551   * Deal with signals.
552   *
553   * LOCKING: acquires and releases configlock.
554   */
555  void *
556  signal_thread(void *arg)
557  {
558      int     err, signo;

559      for (;;) {
560          err = sigwait(&mask, &signo);
561          if (err != 0)
562              log_quit("sigwait failed: %s", strerror(err));
563          switch (signo) {
564          case SIGHUP:
565              /*
566               * Schedule to re-read the configuration file.
567               */
568              pthread_mutex_lock(&configlock);
569              reread = 1;
570              pthread_mutex_unlock(&configlock);
571              break;

572          case SIGTERM:
573              kill_workers();
574              log_msg("terminate with signal %s", strsignal(signo));
575              exit(0);

576          default:
577              kill_workers();
578              log_quit("unexpected signal %d", signo);
579          }
580      }
581  }
```

[550–563] The `signal_thread` function is run by the thread that is responsible for handling signals. In the `main` function, we initialized the signal mask to include `SIGHUP` and `SIGTERM`. Here, we call `sigwait` to wait for one of these signals to occur. If `sigwait` fails, we log an error message and exit.

[564–571] If we receive `SIGHUP`, we acquire the `configlock` mutex, set the `reread` variable to 1, and release the mutex. This tells the printer daemon to reread the configuration file on the next iteration in its processing loop.

[572–575] If we receive `SIGTERM`, we call `kill_workers` to kill all the worker threads, log a message, and call `exit` to terminate the process.

[576–581] If we receive a signal we are not expecting, we kill the worker threads and call `log_quit` to log a message and exit.

```
582  /*
583   * Add an option to the IPP header.
584   *
585   * LOCKING: none.
586   */
587  char *
588  add_option(char *cp, int tag, char *optname, char *optval)
589  {
590    int     n;
591    union {
592        int16_t s;
593        char c[2];
594    }       u;

595    *cp++ = tag;
596    n = strlen(optname);
597    u.s = htons(n);
598    *cp++ = u.c[0];
599    *cp++ = u.c[1];
600    strcpy(cp, optname);
601    cp += n;
602    n = strlen(optval);
603    u.s = htons(n);
604    *cp++ = u.c[0];
605    *cp++ = u.c[1];
606    strcpy(cp, optval);
607    return(cp + n);
608  }
```

[582–594] The `add_option` function is used to add an option to the IPP header that we build to send to the printer. Recall from Figure 21.3 that the format of an attribute is a 1-byte tag describing the type of the attribute, followed by the length of the attribute name stored in binary as a 2-byte integer, followed by the name, the size of the attribute value, and finally the value itself.

IPP makes no attempt to control the alignment of the binary integers embedded in the header. Some processor architectures, such as the SPARC, can't load an integer from an arbitrary address. This means that we can't store the integers in the header by casting a pointer to an `int16_t` to the address in the header where the integer is to be stored. Instead, we need to copy the integer 1 byte at a time. This is why we define the `union` containing a 16-bit integer and 2 bytes.

[595–608] We store the tag in the header and convert the length of the attribute name to network byte order. We copy the length 1 byte at a time to the header. Then we copy the attribute name. We repeat this process for the attribute value and return the address in the header where the next part of the header should begin.

```
609  /*
610   * Single thread to communicate with the printer.
611   *
612   * LOCKING: acquires and releases joblock and configlock.
613   */
614  void *
615  printer_thread(void *arg)
616  {
617    struct job      *jp;
618    int             hlen, ilen, sockfd, fd, nr, nw;
619    char            *icp, *hcp;
620    struct ipp_hdr  *hp;
621    struct stat     sbuf;
622    struct iovec    iov[2];
623    char            name[FILENMSZ];
624    char            hbuf[HBUFSZ];
625    char            ibuf[IBUFSZ];
626    char            buf[IOBUFSZ];
627    char            str[64];

628    for (;;) {
629        /*
630         * Get a job to print.
631         */
632        pthread_mutex_lock(&joblock);
633        while (jobhead == NULL) {
634            log_msg("printer_thread: waiting...");
635            pthread_cond_wait(&jobwait, &joblock);
636        }
637        remove_job(jp = jobhead);
638        log_msg("printer_thread: picked up job %ld", jp->jobid);
639        pthread_mutex_unlock(&joblock);

640        update_jobno();
```

[609–627] The `printer_thread` function is run by the thread that communicates with the network printer. We'll use `icp` and `ibuf` to build the IPP header. We'll use `hcp` and `hbuf` to build the HTTP header. We need to build the headers in separate buffers. The HTTP header includes a length field in ASCII, and we won't know how much space to reserve for it until we assemble the IPP header. We'll use `writev` to write these two headers in one call.

[628–640] The printer thread runs in an infinite loop that waits for jobs to transmit to the printer. We use the `joblock` mutex to protect the list of jobs. If a job is not pending, we use `pthread_cond_wait` to wait for one to arrive. When a job is ready, we remove it from the list by calling `remove_job`. We still hold the mutex at this point, so we release it and call `update_jobno` to write the next job number to `/var/spool/printer/jobno`.

```
641         /*
642          * Check for a change in the config file.
643          */
644         pthread_mutex_lock(&configlock);
645         if (reread) {
646             freeaddrinfo(printer);
647             printer = NULL;
648             printer_name = NULL;
649             reread = 0;
650             pthread_mutex_unlock(&configlock);
651             init_printer();
652         } else {
653             pthread_mutex_unlock(&configlock);
654         }
655         /*
656          * Send job to printer.
657          */
658         sprintf(name, "%s/%s/%ld", SPOOLDIR, DATADIR, jp->jobid);
659         if ((fd = open(name, O_RDONLY)) < 0) {
660             log_msg("job %ld canceled - can't open %s: %s",
661               jp->jobid, name, strerror(errno));
662             free(jp);
663             continue;
664         }
665         if (fstat(fd, &sbuf) < 0) {
666             log_msg("job %ld canceled - can't fstat %s: %s",
667               jp->jobid, name, strerror(errno));
668             free(jp);
669             close(fd);
670             continue;
671         }
```

[641–654] Now that we have a job to print, we check for a change in the configuration file. We lock the `configlock` mutex and check the `reread` variable. If it is nonzero, then we free the old printer `addrinfo` list, clear the pointers, unlock the mutex, and call `init_printer` to reinitialize the printer information. Since only this context looks at and potentially changes the printer information after the `main` thread initialized it, we don't need any synchronization other than using the `configlock` mutex to protect the state of the `reread` flag.

Note that although we acquire and release two different mutex locks in this function, we never hold both at the same time, so we don't need to establish a lock hierarchy (Section 11.6).

[655–671] If we can't open the data file, we log a message, free the `job` structure, and continue. After opening the file, we call `fstat` to find the size of the file. If this fails, we log a message, clean up, and continue.

```
672         if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
673             log_msg("job %ld deferred - can't create socket: %s",
674               jp->jobid, strerror(errno));
675             goto defer;
676         }
677         if (connect_retry(sockfd, printer->ai_addr,
678           printer->ai_addrlen) < 0) {
679             log_msg("job %ld deferred - can't contact printer: %s",
680               jp->jobid, strerror(errno));
681             goto defer;
682         }
683         /*
684          * Set up the IPP header.
685          */
686         icp = ibuf;
687         hp = (struct ipp_hdr *)icp;
688         hp->major_version = 1;
689         hp->minor_version = 1;
690         hp->operation = htons(OP_PRINT_JOB);
691         hp->request_id = htonl(jp->jobid);
692         icp += offsetof(struct ipp_hdr, attr_group);
693         *icp++ = TAG_OPERATION_ATTR;
694         icp = add_option(icp, TAG_CHARSET, "attributes-charset",
695           "utf-8");
696         icp = add_option(icp, TAG_NATULANG,
697           "attributes-natural-language", "en-us");
698         sprintf(str, "http://%s:%d", printer_name, IPP_PORT);
699         icp = add_option(icp, TAG_URI, "printer-uri", str);
```

[672–682] We open a stream socket to communicate with the printer. If the `socket` call fails, we jump down to `defer`, where we will clean up, delay, and try again later. If we can create a socket, we call `connect_retry` to connect to the printer.

[683–699] Next, we set up the IPP header. The operation is a print-job request. We use `htons` to convert the 2-byte operation ID from host to network byte order and `htonl` to convert the 4-byte job ID from host to network byte order. After the initial portion of the header, we set the tag value to indicate that operation attributes follow. We call `add_option` to add attributes to the message. Figure 21.4 lists the required and optional attributes for print-job requests. The first three are required. We specify the character set to be UTF-8, which the printer must support. We specify the language as `en-us`, which represents U.S. English. Another required attribute is the printer Universal Resource Identifier (URI). We set it to `http://`*printer_name*`:631`. (We really should ask the printer for a list of supported URIs and select one from that list, but that would complicate this example without adding much value.)

```
700         icp = add_option(icp, TAG_NAMEWOLANG,
701           "requesting-user-name", jp->req.usernm);
702         icp = add_option(icp, TAG_NAMEWOLANG, "job-name",
703           jp->req.jobnm);
704         if (jp->req.flags & PR_TEXT) {
705             icp = add_option(icp, TAG_MIMETYPE, "document-format",
706               "text/plain");
707         } else {
708             icp = add_option(icp, TAG_MIMETYPE, "document-format",
709               "application/postscript");
710         }
711         *icp++ = TAG_END_OF_ATTR;
712         ilen = icp - ibuf;

713         /*
714          * Set up the HTTP header.
715          */
716         hcp = hbuf;
717         sprintf(hcp, "POST /%s/ipp HTTP/1.1\r\n", printer_name);
718         hcp += strlen(hcp);
719         sprintf(hcp, "Content-Length: %ld\r\n",
720           (long)sbuf.st_size + ilen);
721         hcp += strlen(hcp);
722         strcpy(hcp, "Content-Type: application/ipp\r\n");
723         hcp += strlen(hcp);
724         sprintf(hcp, "Host: %s:%d\r\n", printer_name, IPP_PORT);
725         hcp += strlen(hcp);
726         *hcp++ = '\r';
727         *hcp++ = '\n';
728         hlen = hcp - hbuf;
```

[700–712] The `requesting-user-name` attribute is recommended, but not required. The `job-name` attribute is optional. Recall that the `print` command sends the name of the file being printed as the job name, which can help users distinguish among multiple pending jobs. The last attribute we supply is the `document-format`. If we omit it, the printer will assume that the file conforms to the printer's default format. For a PostScript printer, this is probably PostScript, but some printers can autosense the format and choose between PostScript and text or PostScript and PCL (HP's Printer Command Language). If the `PR_TEXT` flag is set, we specify the document format as `text/plain`. Otherwise, we set it to `application/postscript`. Then we delimit the end of the attributes portion of the header with an end-of-attributes tag and calculate the size of the IPP header.

[713–728] Now that we know the IPP header size, we can set up the HTTP header. We set the `Context-Length` to the size in bytes of the IPP header plus the size of the file to be printed. The `Content-Type` is `application/ipp`. We mark the end of the HTTP header with a carriage return and a line feed.

```
729         /*
730          * Write the headers first.  Then send the file.
731          */
732         iov[0].iov_base = hbuf;
733         iov[0].iov_len = hlen;
734         iov[1].iov_base = ibuf;
735         iov[1].iov_len = ilen;
736         if ((nw = writev(sockfd, iov, 2)) != hlen + ilen) {
737             log_ret("can't write to printer");
738             goto defer;
739         }
740         while ((nr = read(fd, buf, IOBUFSZ)) > 0) {
741             if ((nw = write(sockfd, buf, nr)) != nr) {
742                 if (nw < 0)
743                   log_ret("can't write to printer");
744                 else
745                   log_msg("short write (%d/%d) to printer", nw, nr);
746                 goto defer;
747             }
748         }
749         if (nr < 0) {
750             log_ret("can't read %s", name);
751             goto defer;
752         }
753         /*
754          * Read the response from the printer.
755          */
756         if (printer_status(sockfd, jp)) {
757             unlink(name);
758             sprintf(name, "%s/%s/%ld", SPOOLDIR, REQDIR, jp->jobid);
759             unlink(name);
760             free(jp);
761             jp = NULL;
762         }
```

[729–739] We set the first element of the `iovec` array to refer to the HTTP header and the second element to refer to the IPP header. Then we use `writev` to send both headers to the printer. If the write fails, we log a message and jump to `defer`, where we will clean up and delay before trying again.

[740–752] Next, we send the data file to the printer. We read the data file in `IOBUFSZ` chunks and write it to the socket connected to the printer. If either `read` or `write` fails, we log a message and jump to `defer`.

[753–762] After sending the entire file to be printed, we call `printer_status` to receive the printer's response to our print request. If `printer_status` succeeds, it returns a positive value, and we delete the data and control files. Then we free the `job` structure, set its pointer to `NULL`, and fall through to the `defer` label.

```
763   defer:
764         close(fd);
765         if (sockfd >= 0)
766             close(sockfd);
767         if (jp != NULL) {
768             replace_job(jp);
769             sleep(60);
770         }
771     }
772   }
773   /*
774    * Read data from the printer, possibly increasing the buffer.
775    * Returns offset of end of data in buffer or -1 on failure.
776    *
777    * LOCKING: none.
778    */
779   ssize_t
780   readmore(int sockfd, char **bpp, int off, int *bszp)
781   {
782     ssize_t nr;
783     char    *bp = *bpp;
784     int     bsz = *bszp;
785     if (off >= bsz) {
786         bsz += IOBUFSZ;
787         if ((bp = realloc(*bpp, bsz)) == NULL)
788             log_sys("readmore: can't allocate bigger read buffer");
789         *bszp = bsz;
790         *bpp = bp;
791     }
792     if ((nr = tread(sockfd, &bp[off], bsz-off, 1)) > 0)
793         return(off+nr);
794     else
795         return(-1);
796   }
```

[763–772] At the `defer` label, we close the file descriptor for the open data file. If the socket descriptor is valid, we close it. On error, we place the job back on the head of the pending job list and delay for 1 minute. On success, `jp` is `NULL`, so we simply go back to the top of the loop to get the next job to print.

[773–796] The `readmore` function is used to read part of the response message from the printer. If we're at the end of the buffer, we reallocate a bigger buffer and return the new starting buffer address and buffer size through the `bpp` and `bszp` parameters, respectively. In either case, we read as much as the buffer will hold, starting at the end of the data already in the buffer. We return the new offset in the buffer corresponding to the end of the data read. If the `read` fails or the timeout expires, we return –1.

```
797  /*
798   * Read and parse the response from the printer.  Return 1
799   * if the request was successful, and 0 otherwise.
800   *
801   * LOCKING: none.
802   */
803  int
804  printer_status(int sockfd, struct job *jp)
805  {
806    int             i, success, code, len, found, bufsz;
807    long            jobid;
808    ssize_t         nr;
809    char            *statcode, *reason, *cp, *contentlen;
810    struct ipp_hdr  *hp;
811    char            *bp;

812    /*
813     * Read the HTTP header followed by the IPP response header.
814     * They can be returned in multiple read attempts.  Use the
815     * Content-Length specifier to determine how much to read.
816     */
817    success = 0;
818    bufsz = IOBUFSZ;
819    if ((bp = malloc(IOBUFSZ)) == NULL)
820        log_sys("printer_status: can't allocate read buffer");

821    while ((nr = tread(sockfd, bp, IOBUFSZ, 5)) > 0) {
822        /*
823         * Find the status.  Response starts with "HTTP/x.y"
824         * so we can skip the first 8 characters.
825         */
826        cp = bp + 8;
827        while (isspace((int)*cp))
828            cp++;
829        statcode = cp;
830        while (isdigit((int)*cp))
831            cp++;
832        if (cp == statcode) { /* Bad format; log it and move on */
833            log_msg(bp);
```

[797–811] The `printer_status` function reads the printer's response to a print-job request. We don't know how the printer will respond; it might send a response in multiple messages, send the complete response in one message, or include intermediate acknowledgements, such as HTTP `100 Continue` messages. We need to handle all these possibilities.

[812–833] We allocate a buffer and read from the printer, expecting a response to be available within about 5 seconds. We skip the `HTTP/1.1` and any white space that starts the message. The numeric status code should follow. If it doesn't, we log the contents of the message.

```
834         } else {
835             *cp++ = '\0';
836             reason = cp;
837             while (*cp != '\r' && *cp != '\n')
838                 cp++;
839             *cp = '\0';
840             code = atoi(statcode);
841             if (HTTP_INFO(code))
842                 continue;
843             if (!HTTP_SUCCESS(code)) { /* probable error: log it */
844                 bp[nr] = '\0';
845                 log_msg("error: %s", reason);
846                 break;
847             }
848             /*
849              * The HTTP request was okay, but we still
850              * need to check the IPP status.  First
851              * search for the Content-Length specifier.
852              */
853             i = cp - bp;
854             for (;;) {
855                 while (*cp != 'C' && *cp != 'c' && i < nr) {
856                     cp++;
857                     i++;
858                 }
859                 if (i >= nr &&  /* get more header */
860                   ((nr = readmore(sockfd, &bp, i, &bufsz)) < 0))
861                     goto out;
862                 cp = &bp[i];
```

[834–839] If we have found a numeric status code in the response, we convert the first nondigit character to a null byte. The reason string (a text message) should follow. We search for the terminating carriage return or line feed, also terminating the text string with a null byte.

[840–847] We convert the code to an integer. If this is an informational message only, we ignore it and continue the loop so we end up reading more. We expect to see either a success message or an error message. If we get an error message, we log the error and break out of the loop.

[848–862] If the HTTP request was successful, we need to check the IPP status. We search through the message until we find the `Content-Length` attribute, so we look for a `C` or `c`. HTTP header keywords are case-insensitive, so we need to check both lowercase and uppercase characters.

If we run out of buffer space, we read some more. Since `readmore` calls `realloc`, which might change the address of the buffer, we need to reset `cp` to point to the correct place in the buffer.

```
863                 if (strncasecmp(cp, "Content-Length:", 15) == 0) {
864                     cp += 15;
865                     while (isspace((int)*cp))
866                         cp++;
867                     contentlen = cp;
868                     while (isdigit((int)*cp))
869                         cp++;
870                     *cp++ = '\0';
871                     i = cp - bp;
872                     len = atoi(contentlen);
873                     break;
874                 } else {
875                     cp++;
876                     i++;
877                 }
878             }
879             if (i >= nr &&  /* get more header */
880               ((nr = readmore(sockfd, &bp, i, &bufsz)) < 0))
881                 goto out;
882             cp = &bp[i];
883             found = 0;
884             while (!found) {  /* look for end of HTTP header */
885                 while (i < nr - 2) {
886                     if (*cp == '\n' && *(cp + 1) == '\r' &&
887                       *(cp + 2) == '\n') {
888                         found = 1;
889                         cp += 3;
890                         i += 3;
891                         break;
892                     }
893                     cp++;
894                     i++;
895                 }
896                 if (i >= nr &&  /* get more header */
897                   ((nr = readmore(sockfd, &bp, i, &bufsz)) < 0))
898                     goto out;
899                 cp = &bp[i];
900             }
```

[863–882] If we find the `Content-Length` attribute string, we search for its value. We convert this numeric string into an integer, break out of the `for` loop, and read more from the printer if we've exhausted the contents of the buffer. If we reach the end of the buffer without finding the `Content-Length` attribute, we continue in the loop and read some more from the printer.

[883–900] Once we get the length of the message as specified by the `Content-Length` attribute, we search for the end of the HTTP header (a blank line). If we find it, we set the `found` flag and skip past the blank line in the message.

```
901             if (nr - i < len && /* get more header */
902               ((nr = readmore(sockfd, &bp, i, &bufsz)) < 0))
903                 goto out;
904             cp = &bp[i];
905             hp = (struct ipp_hdr *)cp;
906             i = ntohs(hp->status);
907             jobid = ntohl(hp->request_id);
908             if (jobid != jp->jobid) {
909                 /*
910                  * Different jobs.  Ignore it.
911                  */
912                 log_msg("jobid %ld status code %d", jobid, i);
913                 break;
914             }
915             if (STATCLASS_OK(i))
916                 success = 1;
917             break;
918         }
919     }
920 out:
921     free(bp);
922     if (nr < 0) {
923         log_msg("jobid %ld: error reading printer response: %s",
924           jobid, strerror(errno));
925     }
926     return(success);
927 }
```

[901–904] We continue searching for the end of the HTTP header. If we run out of space in the buffer, we read more. When we find the end of the HTTP header, we calculate the number of bytes that the HTTP header consumed. If the amount we've read minus the size of the HTTP header is not equal to the amount of data in the IPP message (the value we calculated from the content length), then we read some more.

[905–927] We get the status and job ID from the IPP header in the message. Both are stored as integers in network byte order, so we need to convert them to the host byte order by calling `ntohs` and `ntohl`. If the job IDs don't match, then this is not our response, so we log a message and break out of the outer `while` loop. If the IPP status indicates success, then we save the return value and break out of the loop. We return 1 if the print request was successful and 0 if it failed.

This concludes our look at the extended example in this chapter. The programs in this chapter were tested with a Xerox Phaser 860 network-attached PostScript printer. Unfortunately, this printer doesn't recognize the `text/plain` document format, but it

does support the ability to autosense between plaintext and PostScript. Therefore, with this printer, we can print PostScript files and text files, but we cannot print the source to a PostScript program as plaintext unless we use some other utility, such as `a2ps`(1) to encapsulate the PostScript program.

21.6 Summary

This chapter has examined in detail two complete programs: a print spooler daemon that sends a print job to a network printer and a command that can be used to submit a job to be printed to the spooling daemon. This has given us a chance to see lots of features that we described in earlier chapters used in a real program: threads, I/O multiplexing, file I/O, socket I/O, and signals.

Exercises

21.1 Translate the IPP error code values listed in `ipp.h` into error messages. Then modify the print spooler daemon to log a message at the end of the `printer_status` function when the IPP header indicates a printer error.

21.2 Add support to the `print` command and the `printd` daemon to allow users to request double-sided printing. Do the same for landscape and portrait page orientation.

21.3 Modify the print spooler daemon so that when it starts, it contacts the printer to find out what features are supported by the printer so that the daemon doesn't request an option that isn't supported.

21.4 Write a command to report on the status of pending print jobs.

21.5 Write a command to cancel a pending print job.

21.6 Add support for multiple printers to the printer spooler. Include a way to move print jobs from one printer to another.

Appendix A

Function Prototypes

This appendix contains the function prototypes for the standard ISO C, POSIX, and UNIX System functions described in the text. Often, we want to see only the arguments to a function ("Which argument is the file pointer for `fgets`?") or only the return value ("Does `sprintf` return a pointer or a count?"). These prototypes also show which headers need to be included to obtain the definitions of any special constants and to obtain the ISO C function prototype to help detect any compile-time errors.

The page number reference for each function prototype appears to the right of the first header file listed for the function. The page number reference gives the page containing the prototype for the function. That page should be consulted for additional information on the function.

Some functions are supported by only a few of the platforms described in this text. In addition, some platforms support function flags that other platforms don't support. In these cases, we usually list the platforms for which support is provided. In a few cases, however, we list platforms that lack support.

```
void      abort(void);
```
`<stdlib.h>` p. 340
This function never returns

```
int       accept(int sockfd, struct sockaddr *restrict addr,
              socklen_t *restrict len);
```
`<sys/socket.h>` p. 563
Returns: file (socket) descriptor if OK, −1 on error

```
int       access(const char *pathname, int mode);
                  <unistd.h>                                          p. 95
                  mode: R_OK, W_OK, X_OK, F_OK
                  Returns: 0 if OK, -1 on error
unsigned
int       alarm(unsigned int seconds);
                  <unistd.h>                                          p. 313
                  Returns: 0 or number of seconds until previously set alarm

char     *asctime(const struct tm *tmptr);
                  <time.h>                                            p. 175
                  Returns: pointer to null-terminated string

int       atexit(void (*func)(void));
                  <stdlib.h>                                          p. 182
                  Returns: 0 if OK, nonzero on error

int       bind(int sockfd, const struct sockaddr *addr, socklen_t len);
                  <sys/socket.h>                                      p. 560
                  Returns: 0 if OK, -1 on error

void     *calloc(size_t nobj, size_t size);
                  <stdlib.h>                                          p. 189
                  Returns: non-null pointer if OK, NULL on error

speed_t   cfgetispeed(const struct termios *termptr);
                  <termios.h>                                         p. 652
                  Returns: baud rate value

speed_t   cfgetospeed(const struct termios *termptr);
                  <termios.h>                                         p. 652
                  Returns: baud rate value

int       cfsetispeed(struct termios *termptr, speed_t speed);
                  <termios.h>                                         p. 652
                  Returns: 0 if OK, -1 on error

int       cfsetospeed(struct termios *termptr, speed_t speed);
                  <termios.h>                                         p. 652
                  Returns: 0 if OK, -1 on error

int       chdir(const char *pathname);
                  <unistd.h>                                          p. 125
                  Returns: 0 if OK, -1 on error

int       chmod(const char *pathname, mode_t mode);
                  <sys/stat.h>                                        p. 99
                  mode: S_IS[UG]ID, S_ISVTX, S_I[RWX](USR|GRP|OTH)
                  Returns: 0 if OK, -1 on error
```

```
int       chown(const char *pathname, uid_t owner, gid_t group);
                  <unistd.h>                                          p. 102
                  Returns: 0 if OK, -1 on error

void      clearerr(FILE *fp);
                  <stdio.h>                                           p. 141

int       close(int filedes);
                  <unistd.h>                                          p. 63
                  Returns: 0 if OK, -1 on error

int       closedir(DIR *dp);
                  <dirent.h>                                          p. 120
                  Returns: 0 if OK, -1 on error

void      closelog(void);
                  <syslog.h>                                          p. 430

unsigned
char     *CMSG_DATA(struct cmsghdr *cp);
                  <sys/socket.h>                                      p. 607
                  Returns: pointer to data associated with cmsghdr structure

struct
cmsghdr  *CMSG_FIRSTHDR(struct msghdr *mp);
                  <sys/socket.h>                                      p. 607
                  Returns: pointer to first cmsghdr structure associated with the msghdr
                           structure, or NULL if none exists

unsigned
int       CMSG_LEN(unsigned int nbytes);
                  <sys/socket.h>                                      p. 607
                  Returns: size to allocate for data object nbytes large

struct
cmsghdr  *CMSG_NXTHDR(struct msghdr *mp, struct cmsghdr *cp);
                  <sys/socket.h>                                      p. 607
                  Returns: pointer to next cmsghdr structure associated with the msghdr
                           structure given the current cmsghdr structure, or NULL if we're
                           at the last one

int       connect(int sockfd, const struct sockaddr *addr, socklen_t len);
                  <sys/socket.h>                                      p. 561
                  Returns: 0 if OK, -1 on error

int       creat(const char *pathname, mode_t mode);
                  <fcntl.h>                                           p. 62
                  mode: S_IS[UG]ID, S_ISVTX, S_I[RWX](USR|GRP|OTH)
                  Returns: file descriptor opened for write-only if OK, -1 on error

char      *ctermid(char *ptr);
                  <stdio.h>                                           p. 654
                  Returns: pointer to name of controlling terminal on success, pointer to
                           empty string on error
```

```
char    *ctime(const time_t *calptr);
                <time.h>                                        p. 175
                Returns: pointer to null-terminated string

int     dup(int filedes);
                <unistd.h>                                      p. 76
                Returns: new file descriptor if OK, -1 on error

int     dup2(int filedes, int filedes2);
                <unistd.h>                                      p. 76
                Returns: new file descriptor if OK, -1 on error

void    endgrent(void);
                <grp.h>                                         p. 167

void    endhostent(void);
                <netdb.h>                                       p. 553

void    endnetent(void);
                <netdb.h>                                       p. 554

void    endprotoent(void);
                <netdb.h>                                       p. 554

void    endpwent(void);
                <pwd.h>                                         p. 164

void    endservent(void);
                <netdb.h>                                       p. 555

void    endspent(void);
                <shadow.h>                                      p. 166
                Platforms: Linux 2.4.22, Solaris 9

int     execl(const char *pathname, const char *arg0, ... /* (char *) 0 */ );
                <unistd.h>                                      p. 231
                Returns: -1 on error, no return on success

int     execle(const char *pathname, const char *arg0, ... /* (char *) 0,
              char *const envp[] */ );
                <unistd.h>                                      p. 231
                Returns: -1 on error, no return on success

int     execlp(const char *filename, const char *arg0, ... /* (char *) 0 */ );
                <unistd.h>                                      p. 231
                Returns: -1 on error, no return on success

int     execv(const char *pathname, char *const argv[]);
                <unistd.h>                                      p. 231
                Returns: -1 on error, no return on success
```

```
int     execve(const char *pathname, char *const argv[], char *const envp[]);
```
<unistd.h> p. 231
Returns: –1 on error, no return on success

```
int     execvp(const char *filename, char *const argv[]);
```
<unistd.h> p. 231
Returns: –1 on error, no return on success

```
void    _Exit(int status);
```
<stdlib.h> p. 180
This function never returns

```
void    _exit(int status);
```
<unistd.h> p. 180
This function never returns

```
void    exit(int status);
```
<stdlib.h> p. 180
This function never returns

```
int     fattach(int filedes, const char *path);
```
<stropts.h> p. 589
Returns: 0 if OK, –1 on error
Platforms: Linux 2.4.22, Solaris 9

```
int     fchdir(int filedes);
```
<unistd.h> p. 125
Returns: 0 if OK, –1 on error

```
int     fchmod(int filedes, mode_t mode);
```
<sys/stat.h> p. 99
mode: S_IS[UG]ID, S_ISVTX, S_I[RWX](USR|GRP|OTH)
Returns: 0 if OK, –1 on error

```
int     fchown(int filedes, uid_t owner, gid_t group);
```
<unistd.h> p. 102
Returns: 0 if OK, –1 on error

```
int     fclose(FILE *fp);
```
<stdio.h> p. 139
Returns: 0 if OK, EOF on error

```
int     fcntl(int filedes, int cmd, ... /* int arg */ );
```
<fcntl.h> p. 78
cmd: F_DUPFD, F_GETFD, F_SETFD, F_GETFL, F_SETFL,
F_GETOWN, F_SETOWN, F_GETLK, F_SETLK, F_SETLKW
Returns: depends on *cmd* if OK, –1 on error

```
int       fdatasync(int filedes);
                  <unistd.h>                                          p. 77
                  Returns: 0 if OK, -1 on error
                  Platforms: Linux 2.4.22, Solaris 9

void      FD_CLR(int fd, fd_set *fdset);
                  <sys/select.h>                                      p. 476

int       fdetach(const char *path);
                  <stropts.h>                                         p. 590
                  Returns: 0 if OK, -1 on error
                  Platforms: Linux 2.4.22, Solaris 9

int       FD_ISSET(int fd, fd_set *fdset);
                  <sys/select.h>                                      p. 476
                  Returns: nonzero if fd is in set, 0 otherwise

FILE     *fdopen(int filedes, const char *type);
                  <stdio.h>                                           p. 138
                  type: "r", "w", "a", "r+", "w+", "a+",
                  Returns: file pointer if OK, NULL on error

void      FD_SET(int fd, fd_set *fdset);
                  <sys/select.h>                                      p. 476

void      FD_ZERO(fd_set *fdset);
                  <sys/select.h>                                      p. 476

int       feof(FILE *fp);
                  <stdio.h>                                           p. 141
                  Returns: nonzero (true) if end of file on stream, 0 (false) otherwise

int       ferror(FILE *fp);
                  <stdio.h>                                           p. 141
                  Returns: nonzero (true) if error on stream, 0 (false) otherwise

int       fflush(FILE *fp);
                  <stdio.h>                                           p. 137
                  Returns: 0 if OK, EOF on error

int       fgetc(FILE *fp);
                  <stdio.h>                                           p. 140
                  Returns: next character if OK, EOF on end of file or error

int       fgetpos(FILE *restrict fp, fpos_t *restrict pos);
                  <stdio.h>                                           p. 148
                  Returns: 0 if OK, nonzero on error
```

```
char     *fgets(char *restrict buf, int n, FILE *restrict fp);
                    <stdio.h>                                          p. 142
                    Returns: buf if OK, NULL on end of file or error

int      fileno(FILE *fp);
                    <stdio.h>                                          p. 153
                    Returns: file descriptor associated with the stream

void     flockfile(FILE *fp);
                    <stdio.h>                                          p. 403

FILE     *fopen(const char *restrict pathname, const char *restrict type);
                    <stdio.h>                                          p. 138
                    type: "r", "w", "a", "r+", "w+", "a+",
                    Returns: file pointer if OK, NULL on error

pid_t    fork(void);
                    <unistd.h>                                         p. 211
                    Returns: 0 in child, process ID of child in parent, -1 on error

long     fpathconf(int filedes, int name);
                    <unistd.h>                                         p. 41
                    name: _PC_ASYNC_IO, _PC_CHOWN_RESTRICTED,
                          _PC_FILESIZEBITS, _PC_LINK_MAX, _PC_MAX_CANON,
                          _PC_MAX_INPUT, _PC_NAME_MAX, _PC_NO_TRUNC,
                          _PC_PATH_MAX, 'u'_PC_PIPE_BUF, _PC_PRIO_IO,
                          _PC_SYNC_IO, _PC_SYMLINK_MAX, _PC_VDISABLE
                    Returns: corresponding value if OK, -1 on error

int      fprintf(FILE *restrict fp, const char *restrict format, ...);
                    <stdio.h>                                          p. 149
                    Returns: number of characters output if OK, negative value if output error

int      fputc(int c, FILE *fp);
                    <stdio.h>                                          p. 142
                    Returns: c if OK, EOF on error

int      fputs(const char *restrict str, FILE *restrict fp);
                    <stdio.h>                                          p. 143
                    Returns: non-negative value if OK, EOF on error

size_t   fread(void *restrict ptr, size_t size, size_t nobj, FILE *restrict fp);
                    <stdio.h>                                          p. 146
                    Returns: number of objects read

void     free(void *ptr);
                    <stdlib.h>                                         p. 189
```

```
void      freeaddrinfo(struct addrinfo *ai);
                    <sys/socket.h>                                    p. 555
                    <netdb.h>

FILE      *freopen(const char *restrict pathname, const char *restrict type,
                   FILE *restrict fp);
                    <stdio.h>                                         p. 138
                    type: "r", "w", "a", "r+", "w+", "a+",
                    Returns: file pointer if OK, NULL on error

int       fscanf(FILE *restrict fp, const char *restrict format, ...);
                    <stdio.h>                                         p. 151
                    Returns: number of input items assigned, EOF if input error or end of file
                             before any conversion

int       fseek(FILE *fp, long offset, int whence);
                    <stdio.h>                                         p. 147
                    whence: SEEK_SET, SEEK_CUR, SEEK_END
                    Returns: 0 if OK, nonzero on error

int       fseeko(FILE *fp, off_t offset, int whence);
                    <stdio.h>                                         p. 148
                    whence: SEEK_SET, SEEK_CUR, SEEK_END
                    Returns: 0 if OK, nonzero on error

int       fsetpos(FILE *fp, const fpos_t *pos);
                    <stdio.h>                                         p. 148
                    Returns: 0 if OK, nonzero on error

int       fstat(int filedes, struct stat *buf);
                    <sys/stat.h>                                      p. 87
                    Returns: 0 if OK, -1 on error

int       fsync(int filedes);
                    <unistd.h>                                        p. 77
                    Returns: 0 if OK, -1 on error

long      ftell(FILE *fp);
                    <stdio.h>                                         p. 147
                    Returns: current file position indicator if OK, -1L on error

off_t     ftello(FILE *fp);
                    <stdio.h>                                         p. 148
                    Returns: current file position indicator if OK, (off_t)-1 on error

key_t     ftok(const char *path, int id);
                    <sys/ipc.h>                                       p. 519
                    Returns: key if OK, (key_t)-1 on error
```

```
int        ftruncate(int filedes, off_t length);
                     <unistd.h>                                          p. 105
                     Returns: 0 if OK, –1 on error

int        ftrylockfile(FILE *fp);
                     <stdio.h>                                           p. 403
                     Returns: 0 if OK, nonzero if lock can't be acquired

void       funlockfile(FILE *fp);
                     <stdio.h>                                           p. 403

int        fwide(FILE *fp, int mode);
                     <stdio.h>                                           p. 134
                     <wchar.h>
                     Returns: positive if stream is wide oriented, negative if stream is
                              byte oriented, or 0 if stream has no orientation

size_t     fwrite(const void *restrict ptr, size_t size, size_t nobj,
                  FILE *restrict fp);
                     <stdio.h>                                           p. 146
                     Returns: number of objects written

const
char      *gai_strerror(int error);
                     <netdb.h>                                           p. 556
                     Returns: a pointer to a string describing the error

int        getaddrinfo(const char *restrict host, const char *restrict service,
                       const struct addrinfo *restrict hint,
                       struct addrinfo **restrict res);
                     <sys/socket.h>                                      p. 555
                     <netdb.h>
                     Returns: 0 if OK, nonzero error code on error

int        getc(FILE *fp);
                     <stdio.h>                                           p. 140
                     Returns: next character if OK, EOF on end of file or error

int        getchar(void);
                     <stdio.h>                                           p. 140
                     Returns: next character if OK, EOF on end of file or error

int        getchar_unlocked(void);
                     <stdio.h>                                           p. 403
                     Returns: the next character if OK, EOF on end of file or error

int        getc_unlocked(FILE *fp);
                     <stdio.h>                                           p. 403
                     Returns: the next character if OK, EOF on end of file or error
```

```
char      *getcwd(char *buf, size_t size);
```
<unistd.h> p. 126
Returns: *buf* if OK, NULL on error

```
gid_t     getegid(void);
```
<unistd.h> p. 210
Returns: effective group ID of calling process

```
char      *getenv(const char *name);
```
<stdlib.h> p. 192
Returns: pointer to *value* associated with *name*, NULL if not found

```
uid_t     geteuid(void);
```
<unistd.h> p. 210
Returns: effective user ID of calling process

```
gid_t     getgid(void);
```
<unistd.h> p. 210
Returns: real group ID of calling process

```
struct
group     *getgrent(void);
```
<grp.h> p. 167
Returns: pointer if OK, NULL on error or end of file

```
struct
group     *getgrgid(gid_t gid);
```
<grp.h> p. 166
Returns: pointer if OK, NULL on error

```
struct
group     *getgrnam(const char *name);
```
<grp.h> p. 166
Returns: pointer if OK, NULL on error

```
int       getgroups(int gidsetsize, gid_t grouplist[]);
```
<unistd.h> p. 168
Returns: number of supplementary group IDs if OK, –1 on error

```
struct
hostent   *gethostent(void);
```
<netdb.h> p. 553
Returns: pointer if OK, NULL on error

```
int       gethostname(char *name, int namelen);
```
<unistd.h> p. 172
Returns: 0 if OK, –1 on error

```
char      *getlogin(void);
```
<unistd.h> p. 256
Returns: pointer to string giving login name if OK, NULL on error

```
int       getmsg(int filedes, struct strbuf *restrict ctlptr,
                 struct strbuf *restrict dataptr, int *restrict flagptr);
```
p. 469
<stropts.h>
*flagptr: 0, RS_HIPRI
Returns: non-negative value if OK, –1 on error
Platforms: Linux 2.4.22, Solaris 9

```
int       getnameinfo(const struct sockaddr *restrict addr, socklen_t alen,
                      char *restrict host, socklen_t hostlen, char *restrict service,
                      socklen_t servlen, unsigned int flags);
```
p. 556
<sys/socket.h>
<netdb.h>
Returns: 0 if OK, nonzero on error

```
struct
netent    *getnetbyaddr(uint32_t net, int type);
```
p. 554
<netdb.h>
Returns: pointer if OK, NULL on error

```
struct
netent    *getnetbyname(const char *name);
```
p. 554
<netdb.h>
Returns: pointer if OK, NULL on error

```
struct
netent    *getnetent(void);
```
p. 554
<netdb.h>
Returns: pointer if OK, NULL on error

```
int       getopt(int argc, const * const argv[], const char *options);
```
p. 774
<fcntl.h>
extern int optind, opterr, optopt;
extern char *optarg;
Returns: the next option character, or –1 when all options have been processed

```
int       getpeername(int sockfd, struct sockaddr *restrict addr,
                      socklen_t *restrict alenp);
```
p. 561
<sys/socket.h>
Returns: 0 if OK, –1 on error

```
pid_t     getpgid(pid_t pid);
```
p. 269
<unistd.h>
Returns: process group ID if OK, –1 on error

```
pid_t     getpgrp(void);
```
p. 269
<unistd.h>
Returns: process group ID of calling process

```
pid_t     getpid(void);
```
p. 210
<unistd.h>
Returns: process ID of calling process

```
int       getpmsg(int filedes, struct strbuf *restrict ctlptr,
                  struct strbuf *restrict dataptr, int *restrict bandptr,
                  int *restrict flagptr);
```
<stropts.h> p. 469
*flagptr: 0, MSG_HIPRI, MSG_BAND, MSG_ANY
Returns: non-negative value if OK, –1 on error
Platforms: Linux 2.4.22, Solaris 9

```
pid_t     getppid(void);
```
<unistd.h> p. 210
Returns: parent process ID of calling process

```
struct
protoent *getprotobyname(const char *name);
```
<netdb.h> p. 554
Returns: pointer if OK; NULL on error

```
struct
protoent *getprotobynumber(int proto);
```
<netdb.h> p. 554
Returns: pointer if OK, NULL on error

```
struct
protoent *getprotoent(void);
```
<netdb.h> p. 554
Returns: pointer if OK, NULL on error

```
struct
passwd   *getpwent(void);
```
<pwd.h> p. 164
Returns: pointer if OK, NULL on error or end of file

```
struct
passwd   *getpwnam(const char *name);
```
<pwd.h> p. 163
Returns: pointer if OK, NULL on error

```
struct
passwd   *getpwuid(uid_t uid);
```
<pwd.h> p. 163
Returns: pointer if OK, NULL on error

```
int       getrlimit(int resource, struct rlimit *rlptr);
```
<sys/resource.h> p. 202
Returns: 0 if OK, nonzero on error

```
char     *gets(char *buf);
```
<stdio.h> p. 142
Returns: buf if OK, NULL on end of file or error

```
struct
servent  *getservbyname(const char *name, const char *proto);
```
<netdb.h> p. 555
Returns: pointer if OK, NULL on error

```
struct
servent  *getservbyport(int port, const char *proto);
```
<netdb.h> p. 555
Returns: pointer if OK, NULL on error

```
struct
servent  *getservent(void);
```
<netdb.h> p. 555
Returns: pointer if OK, NULL on error

```
pid_t    getsid(pid_t pid);
```
<unistd.h> p. 271
Returns: session leader's process group ID if OK, –1 on error

```
int      getsockname(int sockfd, struct sockaddr *restrict addr,
                     socklen_t *restrict alenp);
```
<sys/socket.h> p. 561
Returns: 0 if OK, –1 on error

```
int      getsockopt(int sockfd, int level, int option, void *restrict val,
                    socklen_t *restrict lenp);
```
<sys/socket.h> p. 579
Returns: 0 if OK, –1 on error

```
struct
spwd     *getspent(void);
```
<shadow.h> p. 166
Returns: pointer if OK, NULL on error
Platforms: Linux 2.4.22, Solaris 9

```
struct
spwd     *getspnam(const char *name);
```
<shadow.h> p. 166
Returns: pointer if OK, NULL on error
Platforms: Linux 2.4.22, Solaris 9

```
int      gettimeofday(struct timeval *restrict tp, void *restrict tzp);
```
<sys/time.h> p. 173
Returns: 0 always

```
uid_t    getuid(void);
```
<unistd.h> p. 210
Returns: real user ID of calling process

```
struct
tm       *gmtime(const time_t *calptr);
```
<time.h> p. 175
Returns: pointer to broken-down time

```
int      grantpt(int filedes);
```
<stdlib.h> p. 682
Returns: 0 on success, –1 on error
Platforms: FreeBSD 5.2.1, Linux 2.4.22, Solaris 9

```
uint32_t  htonl(uint32_t hostint32);
                    <arpa/inet.h>                                    p. 550
                    Returns: 32-bit integer in network byte order

uint16_t  htons(uint16_t hostint16);
                    <arpa/inet.h>                                    p. 550
                    Returns: 16-bit integer in network byte order

const
char     *inet_ntop(int domain, const void *restrict addr, char *restrict str,
                    socklen_t size);
                    <arpa/inet.h>                                    p. 552
                    Returns: pointer to address string on success, NULL on error

int       inet_pton(int domain, const char *restrict str, void *restrict addr);
                    <arpa/inet.h>                                    p. 552
                    Returns: 1 on success, 0 if the format is invalid, or -1 on error

int       initgroups(const char *username, gid_t basegid);
                    <grp.h>     /* Linux & Solaris */                p. 168
                    <unistd.h> /* FreeBSD & Mac OS X */
                    Returns: 0 if OK, -1 on error

int       ioctl(int filedes, int request, ...);
                    <unistd.h>      /* System V */                   p. 83
                    <sys/ioctl.h>  /* BSD and Linux */
                    <stropts.h>     /* XSI STREAMS */
                    Returns: -1 on error, something else if OK

int       isastream(int filedes);
                    <stropts.h>                                      p. 465
                    Returns: 1 (true) if STREAMS device, 0 (false) otherwise
                    Platforms: Linux 2.4.22, Solaris 9

int       isatty(int filedes);
                    <unistd.h>                                       p. 655
                    Returns: 1 (true) if terminal device, 0 (false) otherwise

int       kill(pid_t pid, int signo);
                    <signal.h>                                       p. 312
                    Returns: 0 if OK, -1 on error

int       lchown(const char *pathname, uid_t owner, gid_t group);
                    <unistd.h>                                       p. 102
                    Returns: 0 if OK, -1 on error

int       link(const char *existingpath, const char *newpath);
                    <unistd.h>                                       p. 109
                    Returns: 0 if OK, -1 on error
```

```
int        listen(int sockfd, int backlog);
                  <sys/socket.h>                                          p. 563
                  Returns: 0 if OK, -1 on error

struct
tm        *localtime(const time_t *calptr);
                  <time.h>                                                p. 175
                  Returns: pointer to broken-down time

void       longjmp(jmp_buf env, int val);
                  <setjmp.h>                                              p. 197
                  This function never returns

off_t      lseek(int filedes, off_t offset, int whence);
                  <unistd.h>                                              p. 63
                  whence: SEEK_SET, SEEK_CUR, SEEK_END
                  Returns: new file offset if OK, -1 on error

int        lstat(const char *restrict pathname, struct stat *restrict buf);
                  <sys/stat.h>                                            p. 87
                  Returns: 0 if OK, -1 on error

void      *malloc(size_t size);
                  <stdlib.h>                                              p. 189
                  Returns: non-null pointer if OK, NULL on error

int        mkdir(const char *pathname, mode_t mode);
                  <sys/stat.h>                                            p. 119
                  mode: S_IS[UG]ID, S_ISVTX, S_I[RWX](USR|GRP|OTH)
                  Returns: 0 if OK, -1 on error

int        mkfifo(const char *pathname, mode_t mode);
                  <sys/stat.h>                                            p. 514
                  mode: S_IS[UG]ID, S_ISVTX, S_I[RWX](USR|GRP|OTH)
                  Returns: 0 if OK, -1 on error

int        mkstemp(char *template);
                  <stdlib.h>                                              p. 158
                  Returns: file descriptor if OK, -1 on error

time_t     mktime(struct tm *tmptr);
                  <time.h>                                                p. 175
                  Returns: calendar time if OK, -1 on error

caddr_t    mmap(void *addr, size_t len, int prot, int flag, int filedes, off_t off);
                  <sys/mman.h>                                            p. 487
                  prot: PROT_READ, PROT_WRITE, PROT_EXEC, PROT_NONE
                  flag: MAP_FIXED, MAP_SHARED, MAP_PRIVATE
                  Returns: starting address of mapped region if OK, MAP_FAILED on error
```

```
int       mprotect(void *addr, size_t len, int prot);
                   <sys/mman.h>                                          p. 489
                   Returns: 0 if OK, -1 on error

int       msgctl(int msqid, int cmd, struct msqid_ds *buf);
                   <sys/msg.h>                                           p. 524
                   cmd: IPC_STAT, IPC_SET, IPC_RMID
                   Returns: 0 if OK, -1 on error
                   Platforms: FreeBSD 5.2.1, Linux 2.4.22, Solaris 9

int       msgget(key_t key, int flag);
                   <sys/msg.h>                                           p. 524
                   flag: 0, IPC_CREAT, IPC_EXCL
                   Returns: message queue ID if OK, -1 on error
                   Platforms: FreeBSD 5.2.1, Linux 2.4.22, Solaris 9

ssize_t   msgrcv(int msqid, void *ptr, size_t nbytes, long type, int flag);
                   <sys/msg.h>                                           p. 526
                   flag: 0, IPC_NOWAIT, MSG_NOERROR
                   Returns: size of data portion of message if OK, -1 on error
                   Platforms: FreeBSD 5.2.1, Linux 2.4.22, Solaris 9

int       msgsnd(int msqid, const void *ptr, size_t nbytes, int flag);
                   <sys/msg.h>                                           p. 525
                   flag: 0, IPC_NOWAIT
                   Returns: 0 if OK, -1 on error
                   Platforms: FreeBSD 5.2.1, Linux 2.4.22, Solaris 9

int       msync(void *addr, size_t len, int flags);
                   <sys/mman.h>                                          p. 490
                   Returns: 0 if OK, -1 on error

int       munmap(caddr_t addr, size_t len);
                   <sys/mman.h>                                          p. 490
                   Returns: 0 if OK, -1 on error

uint32_t  ntohl(uint32_t netint32);
                   <arpa/inet.h>                                         p. 550
                   Returns: 32-bit integer in host byte order

uint16_t  ntohs(uint16_t netint16);
                   <arpa/inet.h>                                         p. 550
                   Returns: 16-bit integer in host byte order

int       open(const char *pathname, int oflag, ... /* mode_t mode */ );
                   <fcntl.h>                                             p. 60
                   oflag: O_RDONLY, O_WRONLY, O_RDWR;
                          O_APPEND, O_CREAT, O_DSYNC, O_EXCL, O_NOCTTY,
                          O_NONBLOCK, O_RSYNC, O_SYNC, O_TRUNC
                   mode: S_IS[UG]ID, S_ISVTX, S_I[RWX](USR|GRP|OTH)
                   Returns: file descriptor if OK, -1 on error
                   Platforms: O_FSYNC flag on FreeBSD 5.2.1 and Mac OS X 10.3
```

```
DIR     *opendir(const char *pathname);
```

```
ssize_t  pread(int filedes, void *buf, size_t nbytes, off_t offset);
                    <unistd.h>                                          p. 75
                    Returns: number of bytes read, 0 if end of file, -1 on error

int      printf(const char *restrict format, ...);
                    <stdio.h>                                           p. 149
                    Returns: number of characters output if OK, negative value if output error

int      pselect(int maxfdp1, fd_set *restrict readfds, fd_set *restrict writefds,
                 fd_set *restrict exceptfds, const struct timespec *restrict tsptr,
                 const sigset_t *restrict sigmask);
                    <sys/select.h>                                      p. 478
                    Returns: count of ready descriptors, 0 on timeout, -1 on error
                    Platforms: FreeBSD 5.2.1, Linux 2.4.22, Mac OS X 10.3

void     psignal(int signo, const char *msg);
                    <signal.h>                                          p. 352
                    <siginfo.h>    /* on Solaris */

int      pthread_atfork(void (*prepare)(void), void (*parent)(void),
                        void (*child)(void));
                    <pthread.h>                                         p. 417
                    Returns: 0 if OK, error number on failure

int      pthread_attr_destroy(pthread_attr_t *attr);
                    <pthread.h>                                         p. 389
                    Returns: 0 if OK, error number on failure

int      pthread_attr_getdetachstate(const pthread_attr_t *restrict attr,
                                     int *detachstate);
                    <pthread.h>                                         p. 390
                    Returns: 0 if OK, error number on failure

int      pthread_attr_getguardsize(const pthread_attr_t *restrict attr,
                                   size_t *restrict guardsize);
                    <pthread.h>                                         p. 392
                    Returns: 0 if OK, error number on failure

int      pthread_attr_getstack(const pthread_attr_t *restrict attr, void
                               **restrict stackaddr, size_t *restrict stacksize);
                    <pthread.h>                                         p. 391
                    Returns: 0 if OK, error number on failure

int      pthread_attr_getstacksize(const pthread_attr_t *restrict attr,
                                   size_t *restrict stacksize);
                    <pthread.h>                                         p. 392
                    Returns: 0 if OK, error number on failure

int      pthread_attr_init(pthread_attr_t *attr);
                    <pthread.h>                                         p. 389
                    Returns: 0 if OK, error number on failure
```

```
int     pthread_attr_setdetachstate(pthread_attr_t *attr, int detachstate);
                <pthread.h>                                              p. 390
```
Returns: 0 if OK, error number on failure

```
int     pthread_attr_setguardsize(pthread_attr_t *attr, size_t guardsize);
                <pthread.h>                                              p. 392
```
Returns: 0 if OK, error number on failure

```
int     pthread_attr_setstack(const pthread_attr_t *attr, void *stackaddr,
                    size_t *stacksize);
                <pthread.h>                                              p. 391
```
Returns: 0 if OK, error number on failure

```
int     pthread_attr_setstacksize(pthread_attr_t *attr, size_t stacksize);
                <pthread.h>                                              p. 392
```
Returns: 0 if OK, error number on failure

```
int     pthread_cancel(pthread_t tid);
                <pthread.h>                                              p. 365
```
Returns: 0 if OK, error number on failure

```
void    pthread_cleanup_pop(int execute);
                <pthread.h>                                              p. 365
```

```
void    pthread_cleanup_push(void (*rtn)(void *), void *arg);
                <pthread.h>                                              p. 365
```

```
int     pthread_condattr_destroy(pthread_condattr_t *attr);
                <pthread.h>                                              p. 401
```
Returns: 0 if OK, error number on failure

```
int     pthread_condattr_getpshared(const pthread_condattr_t *restrict attr
                    int *restrict pshared);
                <pthread.h>                                              p. 401
```
Returns: 0 if OK, error number on failure

```
int     pthread_condattr_init(pthread_condattr_t *attr);
                <pthread.h>                                              p. 401
```
Returns: 0 if OK, error number on failure

```
int     pthread_condattr_setpshared(pthread_condattr_t *attr, int pshared);
                <pthread.h>                                              p. 401
```
Returns: 0 if OK, error number on failure

```
int     pthread_cond_broadcast(pthread_cond_t *cond);
                <pthread.h>                                              p. 384
```
Returns: 0 if OK, error number on failure

```
int     pthread_cond_destroy(pthread_cond_t *cond);
                <pthread.h>                                              p. 383
```
Returns: 0 if OK, error number on failure

```
int     pthread_cond_init(pthread_cond_t *restrict cond,
                  pthread_condattr_t *restrict attr);
```
<pthread.h> p. 383
Returns: 0 if OK, error number on failure

```
int     pthread_cond_signal(pthread_cond_t *cond);
```
<pthread.h> p. 384
Returns: 0 if OK, error number on failure

```
int     pthread_cond_timedwait(pthread_cond_t *restrict cond,
                  pthread_mutex_t *restrict mutex,
                  const struct timespec *restrict timeout);
```
<pthread.h> p. 383
Returns: 0 if OK, error number on failure

```
int     pthread_cond_wait(pthread_cond_t *restrict cond,
                  pthread_mutex_t *restrict mutex);
```
<pthread.h> p. 383
Returns: 0 if OK, error number on failure

```
int     pthread_create(pthread_t *restrict tidp,
                  const pthread_attr_t *restrict attr,
                  void *(*start_rtn)(void), void *restrict arg);
```
<pthread.h> p. 357
Returns: 0 if OK, error number on failure

```
int     pthread_detach(pthread_t tid);
```
<pthread.h> p. 368
Returns: 0 if OK, error number on failure

```
int     pthread_equal(pthread_t tid1, pthread_t tid2);
```
<pthread.h> p. 357
Returns: nonzero if equal, 0 otherwise

```
void    pthread_exit(void *rval_ptr);
```
<pthread.h> p. 361

```
int     pthread_getconcurrency(void);
```
<pthread.h> p. 393
Returns: current concurrency level

```
void   *pthread_getspecific(pthread_key_t key);
```
<pthread.h> p. 408
Returns: thread-specific data value or NULL if no value has been associated with the key

```
int     pthread_join(pthread_t thread, void **rval_ptr);
```
<pthread.h> p. 361
Returns: 0 if OK, error number on failure

```
int     pthread_key_create(pthread_key_t *keyp, void (*destructor)(void *));
```
<pthread.h> p. 406
Returns: 0 if OK, error number on failure

```
int     pthread_key_delete(pthread_key_t *key);
```
<pthread.h> p. 407
Returns: 0 if OK, error number on failure

```
int     pthread_kill(pthread_t thread, int signo);
```
<signal.h> p. 414
Returns: 0 if OK, error number on failure

```
int     pthread_mutexattr_destroy(pthread_mutexattr_t *attr);
```
<pthread.h> p. 393
Returns: 0 if OK, error number on failure

```
int     pthread_mutexattr_getpshared(const pthread_mutexattr_t *restrict attr,
                                     int *restrict pshared);
```
<pthread.h> p. 394
Returns: 0 if OK, error number on failure

```
int     pthread_mutexattr_gettype(const pthread_mutexattr_t *restrict attr,
                                  int *restrict type);
```
<pthread.h> p. 395
Returns: 0 if OK, error number on failure

```
int     pthread_mutexattr_init(pthread_mutexattr_t *attr);
```
<pthread.h> p. 393
Returns: 0 if OK, error number on failure

```
int     pthread_mutexattr_setpshared(pthread_mutexattr_t *attr, int pshared);
```
<pthread.h> p. 394
Returns: 0 if OK, error number on failure

```
int     pthread_mutexattr_settype(pthread_mutexattr_t *attr, int type);
```
<pthread.h> p. 395
Returns: 0 if OK, error number on failure

```
int     pthread_mutex_destroy(pthread_mutex_t *mutex);
```
<pthread.h> p. 371
Returns: 0 if OK, error number on failure

```
int     pthread_mutex_init(pthread_mutex_t *restrict mutex,
                           const pthread_mutexattr_t *restrict attr);
```
<pthread.h> p. 371
Returns: 0 if OK, error number on failure

```
int     pthread_mutex_lock(pthread_mutex_t *mutex);
```
<pthread.h> p. 371
Returns: 0 if OK, error number on failure

```
int     pthread_mutex_trylock(pthread_mutex_t *mutex);
                <pthread.h>                                          p. 371
                Returns: 0 if OK, error number on failure

int     pthread_mutex_unlock(pthread_mutex_t *mutex);
                <pthread.h>                                          p. 371
                Returns: 0 if OK, error number on failure

int     pthread_once(pthread_once_t *initflag, void (*initfn)(void));
                <pthread.h>                                          p. 408
                pthread_once_t initflag = PTHREAD_ONCE_INIT;
                Returns: 0 if OK, error number on failure

int     pthread_rwlockattr_destroy(pthread_rwlockattr_t *attr);
                <pthread.h>                                          p. 400
                Returns: 0 if OK, error number on failure

int     pthread_rwlockattr_getpshared(const pthread_rwlockattr_t *restrict attr,
                                      int *restrict pshared);
                <pthread.h>                                          p. 400
                Returns: 0 if OK, error number on failure

int     pthread_rwlockattr_init(pthread_rwlockattr_t *attr);
                <pthread.h>                                          p. 400
                Returns: 0 if OK, error number on failure

int     pthread_rwlockattr_setpshared(pthread_rwlockattr_t *attr, int pshared);
                <pthread.h>                                          p. 400
                Returns: 0 if OK, error number on failure

int     pthread_rwlock_destroy(pthread_rwlock_t *rwlock);
                <pthread.h>                                          p. 379
                Returns: 0 if OK, error number on failure

int     pthread_rwlock_init(pthread_rwlock_t *restrict rwlock,
                            const pthread_rwlockattr_t *restrict attr);
                <pthread.h>                                          p. 379
                Returns: 0 if OK, error number on failure

int     pthread_rwlock_rdlock(pthread_rwlock_t *rwlock);
                <pthread.h>                                          p. 379
                Returns: 0 if OK, error number on failure

int     pthread_rwlock_tryrdlock(pthread_rwlock_t *rwlock);
                <pthread.h>                                          p. 379
                Returns: 0 if OK, error number on failure

int     pthread_rwlock_trywrlock(pthread_rwlock_t *rwlock);
                <pthread.h>                                          p. 379
                Returns: 0 if OK, error number on failure
```

```
int        pthread_rwlock_unlock(pthread_rwlock_t *rwlock);
                    <pthread.h>                                          p. 379
                    Returns: 0 if OK, error number on failure

int        pthread_rwlock_wrlock(pthread_rwlock_t *rwlock);
                    <pthread.h>                                          p. 379
                    Returns: 0 if OK, error number on failure

pthread_t  pthread_self(void);
                    <pthread.h>                                          p. 357
                    Returns: thread ID of the calling thread

int        pthread_setcancelstate(int state, int *oldstate);
                    <pthread.h>                                          p. 410
                    Returns: 0 if OK, error number on failure

int        pthread_setcanceltype(int type, int *oldtype);
                    <pthread.h>                                          p. 411
                    Returns: 0 if OK, error number on failure

int        pthread_setconcurrency(int level);
                    <pthread.h>                                          p. 393
                    Returns: 0 if OK, error number on failure

int        pthread_setspecific(pthread_key_t key, const void *value);
                    <pthread.h>                                          p. 408
                    Returns: 0 if OK, error number on failure

int        pthread_sigmask(int how, const sigset_t *restrict set,
                        sigset_t *restrict oset);
                    <signal.h>                                           p. 413
                    Returns: 0 if OK, error number on failure

void       pthread_testcancel(void);
                    <pthread.h>                                          p. 411

char       *ptsname(int filedes);
                    <stdlib.h>                                           p. 682
                    Returns: pointer to name of PTY slave if OK, NULL on error
                    Platforms: FreeBSD 5.2.1, Linux 2.4.22, Solaris 9

int        putc(int c, FILE *fp);
                    <stdio.h>                                            p. 142
                    Returns: c if OK, EOF on error

int        putchar(int c);
                    <stdio.h>                                            p. 142
                    Returns: c if OK, EOF on error
```

```
int        putchar_unlocked(int c);
```
<stdio.h> p. 403
Returns: *c* if OK, EOF on error

```
int        putc_unlocked(int c, FILE *fp);
```
<stdio.h> p. 403
Returns: *c* if OK, EOF on error

```
int        putenv(char *str);
```
<stdlib.h> p. 194
Returns: 0 if OK, nonzero on error

```
int        putmsg(int filedes, const struct strbuf *ctlptr,
               const struct strbuf *dataptr, int flag);
```
<stropts.h> p. 463
flag: 0, RS_HIPRI
Returns: 0 if OK, −1 on error
Platforms: Linux 2.4.22, Solaris 9

```
int        putpmsg(int filedes, const struct strbuf *ctlptr,
               const struct strbuf *dataptr, int band, int flag);
```
<stropts.h> p. 463
flag: 0, MSG_HIPRI, MSG_BAND
Returns: 0 if OK, −1 on error
Platforms: Linux 2.4.22, Solaris 9

```
int        puts(const char *str);
```
<stdio.h> p. 143
Returns: non-negative value if OK, EOF on error

```
ssize_t    pwrite(int filedes, const void *buf, size_t nbytes, off_t offset);
```
<unistd.h> p. 75
Returns: number of bytes written if OK, −1 on error

```
int        raise(int signo);
```
<signal.h> p. 312
Returns: 0 if OK, −1 on error

```
ssize_t    read(int filedes, void *buf, size_t nbytes);
```
<unistd.h> p. 67
Returns: number of bytes read if OK, 0 if end of file, −1 on error

```
struct
dirent    *readdir(DIR *dp);
```
<dirent.h> p. 120
Returns: pointer if OK, NULL at end of directory or error

```
int        readlink(const char *restrict pathname, char *restrict buf,
               size_t bufsize);
```
<unistd.h> p. 115
Returns: number of bytes read if OK, −1 on error

```
ssize_t   readv(int filedes, const struct iovec *iov, int iovcnt);
                    <sys/uio.h>                                          p. 483
                    Returns: number of bytes read if OK, -1 on error

void      *realloc(void *ptr, size_t newsize);
                    <stdlib.h>                                           p. 189
                    Returns: non-null pointer if OK, NULL on error

ssize_t   recv(int sockfd, void *buf, size_t nbytes, int flags);
                    <sys/socket.h>                                       p. 567
                    flags: 0, MSG_PEEK, MSG_OOB, MSG_WAITALL
                    Returns: length of message in bytes, 0 if no messages are available
                        and peer has done an orderly shutdown, or -1 on error
                    Platforms: MSG_TRUNC flag on Linux 2.4.22

ssize_t   recvfrom(int sockfd, void *restrict buf, size_t len, int flags,
                    struct sockaddr *restrict addr, socklen_t *restrict addrlen);
                    <sys/socket.h>                                       p. 567
                    flags: 0, MSG_PEEK, MSG_OOB, MSG_WAITALL
                    Returns: length of message in bytes, 0 if no messages are available
                        and peer has done an orderly shutdown, or -1 on error
                    Platforms: MSG_TRUNC flag on Linux 2.4.22

ssize_t   recvmsg(int sockfd, struct msghdr *msg, int flags);
                    <sys/socket.h>                                       p. 568
                    flags: 0, MSG_PEEK, MSG_OOB, MSG_WAITALL
                    Returns: length of message in bytes, 0 if no messages are available
                        and peer has done an orderly shutdown, or -1 on error
                    Platforms: MSG_TRUNC flag on Linux 2.4.22

int       remove(const char *pathname);
                    <stdio.h>                                            p. 111
                    Returns: 0 if OK, -1 on error

int       rename(const char *oldname, const char *newname);
                    <stdio.h>                                            p. 111
                    Returns: 0 if OK, -1 on error

void      rewind(FILE *fp);
                    <stdio.h>                                            p. 147

void      rewinddir(DIR *dp);
                    <dirent.h>                                           p. 120

int       rmdir(const char *pathname);
                    <unistd.h>                                           p. 120
                    Returns: 0 if OK, -1 on error

int       scanf(const char *restrict format, ...);
                    <stdio.h>                                            p. 151
                    Returns: number of input items assigned, EOF if input error or
                        end of file before any conversion
```

```
void      seekdir(DIR *dp, long loc);
                  <dirent.h>                                        p. 120

int       select(int maxfdp1, fd_set *restrict readfds, fd_set *restrict writefds,
                  fd_set *restrict exceptfds, struct timeval *restrict tvptr);
                  <sys/select.h>                                    p. 475
                  Returns: count of ready descriptors, 0 on timeout, -1 on error

int       semctl(int semid, int semnum, int cmd, ... /* union semun arg */ );
                  <sys/sem.h>                                       p. 529
                  cmd: IPC_STAT, IPC_SET, IPC_RMID, GETPID, GETNCNT,
                       GETZCNT, GETVAL, SETVAL, GETALL, SETALL
                  Returns: (depends on command)

int       semget(key_t key, int nsems, int flag);
                  <sys/sem.h>                                       p. 529
                  flag: 0, IPC_CREAT, IPC_EXCL
                  Returns: semaphore ID if OK, -1 on error

int       semop(int semid, struct sembuf semoparray[], size_t nops);
                  <sys/sem.h>                                       p. 530
                  Returns: 0 if OK, -1 on error

ssize_t   send(int sockfd, const void *buf, size_t nbytes, int flags);
                  <sys/socket.h>                                    p. 565
                  flags: 0, MSG_DONTROUTE, MSG_EOR, MSG_OOB
                  Returns: number of bytes sent if OK, -1 on error
                  Platforms: MSG_DONTWAIT flag on FreeBSD 5.2.1, Linux 2.4.22, Mac OS X 10.3
                             MSG_EOR flag not on Solaris 9

ssize_t   sendmsg(int sockfd, const struct msghdr *msg, int flags);
                  <sys/socket.h>                                    p. 566
                  flags: 0, MSG_DONTROUTE, MSG_EOR, MSG_OOB
                  Returns: number of bytes sent if OK, -1 on error
                  Platforms: MSG_DONTWAIT flag on FreeBSD 5.2.1, Linux 2.4.22, Mac OS X 10.3
                             MSG_EOR flag not on Solaris 9

ssize_t   sendto(int sockfd, const void *buf, size_t nbytes, int flags,
                  const struct sockaddr *destaddr, socklen_t destlen);
                  <sys/socket.h>                                    p. 566
                  flags: 0, MSG_DONTROUTE, MSG_EOR, MSG_OOB
                  Returns: number of bytes sent if OK, -1 on error
                  Platforms: MSG_DONTWAIT flag on FreeBSD 5.2.1, Linux 2.4.22, Mac OS X 10.3
                             MSG_EOR flag not on Solaris 9

void      setbuf(FILE *restrict fp, char *restrict buf);
                  <stdio.h>                                         p. 136

int       setegid(gid_t gid);
                  <unistd.h>                                        p. 241
                  Returns: 0 if OK, -1 on error
```

```
int     setenv(const char *name, const char *value, int rewrite);
                <stdlib.h>                                          p. 194
                Returns: 0 if OK, nonzero on error

int     seteuid(uid_t uid);
                <unistd.h>                                          p. 241
                Returns: 0 if OK, -1 on error

int     setgid(gid_t gid);
                <unistd.h>                                          p. 237
                Returns: 0 if OK, -1 on error

void    setgrent(void);
                <grp.h>                                             p. 167

int     setgroups(int ngroups, const gid_t grouplist[]);
                <grp.h>     /* on Linux */                          p. 168
                <unistd.h> /* on FreeBSD, Mac OS X, and Solaris */
                Returns: 0 if OK, -1 on error

void    sethostent(int stayopen);
                <netdb.h>                                           p. 553

int     setjmp(jmp_buf env);
                <setjmp.h>                                          p. 197
                Returns: 0 if called directly, nonzero if returning from a call to longjmp

int     setlogmask(int maskpri);
                <syslog.h>                                          p. 430
                Returns: previous log priority mask value

void    setnetent(int stayopen);
                <netdb.h>                                           p. 554

int     setpgid(pid_t pid, pid_t pgid);
                <unistd.h>                                          p. 269
                Returns: 0 if OK, -1 on error

void    setprotoent(int stayopen);
                <netdb.h>                                           p. 554

void    setpwent(void);
                <pwd.h>                                             p. 164

int     setregid(gid_t rgid, gid_t egid);
                <unistd.h>                                          p. 240
                Returns: 0 if OK, -1 on error

int     setreuid(uid_t ruid, uid_t euid);
                <unistd.h>                                          p. 240
                Returns: 0 if OK, -1 on error
```

```
int       setrlimit(int resource, const struct rlimit *rlptr);
                    <sys/resource.h>                                    p. 202
                    Returns: 0 if OK, nonzero on error

void      setservent(int stayopen);
                    <netdb.h>                                           p. 555

pid_t     setsid(void);
                    <unistd.h>                                          p. 271
                    Returns: process group ID if OK, -1 on error

int       setsockopt(int sockfd, int level, int option, const void *val,
                     socklen_t len);
                    <sys/socket.h>                                      p. 579
                    Returns: 0 if OK, -1 on error

void      setspent(void);
                    <shadow.h>                                          p. 166
                    Platforms: Linux 2.4.22, Solaris 9

int       setuid(uid_t uid);
                    <unistd.h>                                          p. 237
                    Returns: 0 if OK, -1 on error

int       setvbuf(FILE *restrict fp, char *restrict buf, int mode, size_t size);
                    <stdio.h>                                           p. 136
                    mode: _IOFBF, _IOLBF, _IONBF
                    Returns: 0 if OK, nonzero on error

void     *shmat(int shmid, const void *addr, int flag);
                    <sys/shm.h>                                         p. 536
                    flag: 0, SHM_RND, SHM_RDONLY
                    Returns: pointer to shared memory segment if OK, -1 on error

int       shmctl(int shmid, int cmd, struct shmid_ds *buf);
                    <sys/shm.h>                                         p. 535
                    cmd: IPC_STAT, IPC_SET, IPC_RMID,
                         SHM_LOCK, SHM_UNLOCK
                    Returns: 0 if OK, -1 on error

int       shmdt(void *addr);
                    <sys/shm.h>                                         p. 536
                    Returns: 0 if OK, -1 on error

int       shmget(key_t key, int size, int flag);
                    <sys/shm.h>                                         p. 534
                    flag: 0, IPC_CREAT, IPC_EXCL
                    Returns: shared memory ID if OK, -1 on error
```

```
int     shutdown(int sockfd, int how);
                <sys/socket.h>                                          p. 548
                how: SHUT_RD, SHUT_WR, SHUT_RDWR
                Returns: 0 if OK, -1 on error

int     sig2str(int signo, char *str);
                <signal.h>                                              p. 353
                Returns: 0 if OK, -1 on error
                Platforms: Solaris 9

int     sigaction(int signo, const struct sigaction *restrict act,
                  struct sigaction *restrict oact);
                <signal.h>                                              p. 324
                Returns: 0 if OK, -1 on error

int     sigaddset(sigset_t *set, int signo);
                <signal.h>                                              p. 319
                Returns: 0 if OK, -1 on error

int     sigdelset(sigset_t *set, int signo);
                <signal.h>                                              p. 319
                Returns: 0 if OK, -1 on error

int     sigemptyset(sigset_t *set);
                <signal.h>                                              p. 319
                Returns: 0 if OK, -1 on error

int     sigfillset(sigset_t *set);
                <signal.h>                                              p. 319
                Returns: 0 if OK, -1 on error

int     sigismember(const sigset_t *set, int signo);
                <signal.h>                                              p. 319
                Returns: 1 if true, 0 if false, -1 on error

void    siglongjmp(sigjmp_buf env, int val);
                <setjmp.h>                                              p. 330
                This function never returns

void    (*signal(int signo, void (*func)(int)))(int);
                <signal.h>                                              p. 298
                Returns: previous disposition of signal if OK, SIG_ERR on error

int     sigpending(sigset_t *set);
                <signal.h>                                              p. 322
                Returns: 0 if OK, -1 on error
```

```
int       sigprocmask(int how, const sigset_t *restrict set,
                      sigset_t *restrict oset);
```
<signal.h> p. 320
how: SIG_BLOCK, SIG_UNBLOCK, SIG_SETMASK
Returns: 0 if OK, –1 on error

```
int       sigsetjmp(sigjmp_buf env, int savemask);
```
<setjmp.h> p. 330
Returns: 0 if called directly, nonzero if returning from a call to siglongjmp

```
int       sigsuspend(const sigset_t *sigmask);
```
<signal.h> p. 334
Returns: –1 with errno set to EINTR

```
int       sigwait(const sigset_t *restrict set, int *restrict signop);
```
<signal.h> p. 413
Returns: 0 if OK, error number on failure

```
unsigned
int       sleep(unsigned int seconds);
```
<unistd.h> p. 347
Returns: 0 or number of unslept seconds

```
int       snprintf(char *restrict buf, size_t n, const char *restrict format, ...);
```
<stdio.h> p. 149
Returns: number of characters stored in array if OK, negative value if encoding error

```
int       sockatmark(int sockfd);
```
<sys/socket.h> p. 582
Returns: 1 if at mark, 0 if not at mark, –1 on error

```
int       socket(int domain, int type, int protocol);
```
<sys/socket.h> p. 546
type: SOCK_STREAM, SOCK_DGRAM, SOCK_SEQPACKET,
Returns: file (socket) descriptor if OK, –1 on error

```
int       socketpair(int domain, int type, int protocol, int sockfd[2]);
```
<sys/socket.h> p. 594
type: SOCK_STREAM, SOCK_DGRAM, SOCK_SEQPACKET,
Returns: 0 if OK, –1 on error

```
int       sprintf(char *restrict buf, const char *restrict format, ...);
```
<stdio.h> p. 149
Returns: number of characters stored in array if OK, negative value if encoding error

```
int       sscanf(const char *restrict buf, const char *restrict format, ...);
```
<stdio.h> p. 151
Returns: number of input items assigned, EOF if input error or end of file before any conversion

```
int        stat(const char *restrict pathname, struct stat *restrict buf);
                    <sys/stat.h>                                          p. 87
                    Returns: 0 if OK, -1 on error

int        str2sig(const char *str, int *signop);
                    <signal.h>                                            p. 353
                    Returns: 0 if OK, -1 on error
                    Platforms: Solaris 9

char      *strerror(int errnum);
                    <string.h>                                            p. 15
                    Returns: pointer to message string

size_t     strftime(char *restrict buf, size_t maxsize, const char *restrict format,
                    const struct tm *restrict tmptr);
                    <time.h>                                              p. 176
                    Returns: number of characters stored in array if room, 0 otherwise

char       *strsignal(int signo);
                    <string.h>                                            p. 352
                    Returns: a pointer to a string describing the signal

int        symlink(const char *actualpath, const char *sympath);
                    <unistd.h>                                            p. 115
                    Returns: 0 if OK, -1 on error

void       sync(void);
                    <unistd.h>                                            p. 77

long       sysconf(int name);
                    <unistd.h>                                            p. 41
                    name: _SC_ARG_MAX, _SC_ATEXIT_MAX, _SC_CHILD_MAX,
                          _SC_CLK_TCK, _SC_COLL_WEIGHTS_MAX,
                          _SC_HOST_NAME_MAX, _SC_IOV_MAX, _SC_JOB_CONTROL,
                          _SC_LINE_MAX, _SC_LOGIN_NAME_MAX, _SC_NGROUPS_MAX,
                          _SC_OPEN_MAX, _SC_PAGESIZE, _SC_PAGE_SIZE,
                          _SC_READER_WRITER_LOCKS, _SC_RE_DUP_MAX,
                          _SC_SAVED_IDS, _SC_SHELL, _SC_STREAM_MAX,
                          _SC_SYMLOOP_MAX, _SC_TTY_NAME_MAX, _SC_TZNAME_MAX,
                          _SC_VERSION, _SC_XOPEN_CRYPT, _SC_XOPEN_LEGACY,
                          _SC_XOPEN_REALTIME, _SC_XOPEN_REALTIME_THREADS,
                          _SC_XOPEN_VERSION
                    Returns: corresponding value if OK, -1 on error

void       syslog(int priority, char *format, ...);
                    <syslog.h>                                            p. 430

int        system(const char *cmdstring);
                    <stdlib.h>                                            p. 246
                    Returns: termination status of shell
```

```
int        tcdrain(int filedes);
                   <termios.h>                                          p. 653
                   Returns: 0 if OK, -1 on error

int        tcflow(int filedes, int action);
                   <termios.h>                                          p. 653
                   action: TCOOFF, TCOON, TCIOFF, TCION
                   Returns: 0 if OK, -1 on error

int        tcflush(int filedes, int queue);
                   <termios.h>                                          p. 653
                   queue: TCIFLUSH, TCOFLUSH, TCIOFLUSH
                   Returns: 0 if OK, -1 on error

int        tcgetattr(int filedes, struct termios *termptr);
                   <termios.h>                                          p. 643
                   Returns: 0 if OK, -1 on error

pid_t      tcgetpgrp(int filedes);
                   <unistd.h>                                           p. 273
                   Returns: process group ID of foreground process group if OK, -1 on error

pid_t      tcgetsid(int filedes);
                   <termios.h>                                          p. 274
                   Returns: session leader's process group ID if OK, -1 on error

int        tcsendbreak(int filedes, int duration);
                   <termios.h>                                          p. 653
                   Returns: 0 if OK, -1 on error

int        tcsetattr(int filedes, int opt, const struct termios *termptr);
                   <termios.h>                                          p. 643
                   opt: TCSANOW, TCSADRAIN, TCSAFLUSH
                   Returns: 0 if OK, -1 on error

int        tcsetpgrp(int filedes, pid_t pgrpid);
                   <unistd.h>                                           p. 273
                   Returns: 0 if OK, -1 on error

long       telldir(DIR *dp);
                   <dirent.h>                                           p. 120
                   Returns: current location in directory associated with dp

char      *tempnam(const char *directory, const char *prefix);
                   <stdio.h>                                            p. 157
                   Returns: pointer to unique pathname

time_t     time(time_t *calptr);
                   <time.h>                                             p. 173
                   Returns: value of time if OK, -1 on error
```

```
clock_t   times(struct tms *buf);
                    <sys/times.h>                                    p. 257
                    Returns: elapsed wall clock time in clock ticks if OK, -1 on error

FILE      *tmpfile(void);
                    <stdio.h>                                        p. 155
                    Returns: file pointer if OK, NULL on error

char      *tmpnam(char *ptr);
                    <stdio.h>                                        p. 155
                    Returns: pointer to unique pathname

int       truncate(const char *pathname, off_t length);
                    <unistd.h>                                       p. 105
                    Returns: 0 if OK, -1 on error

char      *ttyname(int filedes);
                    <unistd.h>                                       p. 655
                    Returns: pointer to pathname of terminal, NULL on error

mode_t    umask(mode_t cmask);
                    <sys/stat.h>                                     p. 97
                    Returns: previous file mode creation mask

int       uname(struct utsname *name);
                    <sys/utsname.h>                                  p. 171
                    Returns: non-negative value if OK, -1 on error

int       ungetc(int c, FILE *fp);
                    <stdio.h>                                        p. 141
                    Returns: c if OK, EOF on error

int       unlink(const char *pathname);
                    <unistd.h>                                       p. 109
                    Returns: 0 if OK, -1 on error

int       unlockpt(int filedes);
                    <stdlib.h>                                       p. 682
                    Returns: 0 on success, -1 on error
                    Platforms: FreeBSD 5.2.1, Linux 2.4.22, Solaris 9

void      unsetenv(const char *name);
                    <stdlib.h>                                       p. 194

int       utime(const char *pathname, const struct utimbuf *times);
                    <utime.h>                                        p. 116
                    Returns: 0 if OK, -1 on error
```

```
int     vfprintf(FILE *restrict fp, const char *restrict format, va_list arg);
                <stdarg.h>                                              p. 151
                <stdio.h>
                Returns: number of characters output if OK, negative
                    value if output error

int     vfscanf(FILE *restrict fp, const char *restrict format, va_list arg);
                <stdarg.h>                                              p. 151
                <stdio.h>
                Returns: number of input items assigned, EOF if input error or end of file
                    before any conversion

int     vprintf(const char *restrict format, va_list arg);
                <stdarg.h>                                              p. 151
                <stdio.h>
                Returns: number of characters output if OK, negative
                    value if output error

int     vscanf(const char *restrict format, va_list arg);
                <stdarg.h>                                              p. 151
                <stdio.h>
                Returns: number of input items assigned, EOF if input error or end of file
                    before any conversion

int     vsnprintf(char *restrict buf, size_t n, const char *restrict format,
            va_list arg);
                <stdarg.h>                                              p. 151
                <stdio.h>
                Returns: number of characters stored in array if OK, negative
                    value if encoding error

int     vsprintf(char *restrict buf, const char *restrict format, va_list arg);
                <stdarg.h>                                              p. 151
                <stdio.h>
                Returns: number of characters stored in array if OK, negative
                    value if encoding error

int     vsscanf(const char *restrict buf, const char *restrict format,
            va_list arg);
                <stdarg.h>                                              p. 151
                <stdio.h>
                Returns: number of input items assigned, EOF if input error or end of file
                    before any conversion

void    vsyslog(int priority, const char *format, va_list arg);
                <syslog.h>                                              p. 432
                <stdarg.h>

pid_t   wait(int *statloc);
                <sys/wait.h>                                            p. 220
                Returns: process ID if OK, 0, or -1 on error
```

```
int       waitid(idtype_t idtype, id_t id, siginfo_t *infop, int options);
                  <sys/wait.h>                                          p. 220
                  idtype: P_PID, P_PGID, P_ALL
                  options: WCONTINUED, WEXITED, WNOHANG, WNOWAIT, WSTOPPED
                  Returns: 0 if OK, -1 on error
                  Platforms: Solaris 9

pid_t     waitpid(pid_t pid, int *statloc, int options);
                  <sys/wait.h>                                          p. 220
                  options: 0, WCONTINUED, WNOHANG, WUNTRACED
                  Returns: process ID if OK, 0, or -1 on error

pid_t     wait3(int *statloc, int options, struct rusage *rusage);
                  <sys/types.h>                                         p. 227
                  <sys/wait.h>
                  <sys/time.h>
                  <sys/resource.h>
                  options: 0, WNOHANG, WUNTRACED
                  Returns: process ID if OK, 0, or -1 on error

pid_t     wait4(pid_t pid, int *statloc, int options, struct rusage *rusage);
                  <sys/types.h>                                         p. 227
                  <sys/wait.h>
                  <sys/time.h>
                  <sys/resource.h>
                  options: 0, WNOHANG, WUNTRACED
                  Returns: process ID if OK, 0, or -1 on error

ssize_t   write(int filedes, const void *buf, size_t nbytes);
                  <unistd.h>                                            p. 68
                  Returns: number of bytes written if OK, -1 on error

ssize_t   writev(int filedes, const struct iovec *iov, int iovcnt);
                  <sys/uio.h>                                           p. 483
                  Returns: number of bytes written if OK, -1 on error
```

Appendix B

Miscellaneous Source Code

B.1 Our Header File

Most programs in the text include the header `apue.h`, shown in Figure B.1. It defines constants (such as `MAXLINE`) and prototypes for our own functions.

Most programs need to include the following headers: `<stdio.h>`, `<stdlib.h>` (for the `exit` function prototype), and `<unistd.h>` (for all the standard UNIX function prototypes). So our header automatically includes these system headers, along with `<string.h>`. This also reduces the size of all the program listings in the text.

```
/* Our own header, to be included before all standard system headers */

#ifndef _APUE_H
#define _APUE_H

#define _XOPEN_SOURCE   600 /* Single UNIX Specification, Version 3 */

#include <sys/types.h>      /* some systems still require this */
#include <sys/stat.h>
#include <sys/termios.h>    /* for winsize */
#ifndef TIOCGWINSZ
#include <sys/ioctl.h>
#endif
#include <stdio.h>      /* for convenience */
#include <stdlib.h>     /* for convenience */
#include <stddef.h>     /* for offsetof */
#include <string.h>     /* for convenience */
#include <unistd.h>     /* for convenience */
#include <signal.h>     /* for SIG_ERR */
```

```
#define MAXLINE 4096            /* max line length */

/*
 * Default file access permissions for new files.
 */
#define FILE_MODE   (S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH)

/*
 * Default permissions for new directories.
 */
#define DIR_MODE    (FILE_MODE | S_IXUSR | S_IXGRP | S_IXOTH)

typedef void    Sigfunc(int);   /* for signal handlers */

#if defined(SIG_IGN) && !defined(SIG_ERR)
#define SIG_ERR ((Sigfunc *)-1)
#endif

#define min(a,b)    ((a) < (b) ? (a) : (b))
#define max(a,b)    ((a) > (b) ? (a) : (b))

/*
 * Prototypes for our own functions.
 */
char    *path_alloc(int *);             /* Figure 2.15 */
long     open_max(void);                /* Figure 2.16 */
void     clr_fl(int, int);              /* Figure 3.11 */
void     set_fl(int, int);              /* Figure 3.11 */
void     pr_exit(int);                  /* Figure 8.5 */
void     pr_mask(const char *);         /* Figure 10.14 */
Sigfunc *signal_intr(int, Sigfunc *);   /* Figure 10.19 */

int      tty_cbreak(int);               /* Figure 18.20 */
int      tty_raw(int);                  /* Figure 18.20 */
int      tty_reset(int);                /* Figure 18.20 */
void     tty_atexit(void);              /* Figure 18.20 */
#ifdef  ECHO    /* only if <termios.h> has been included */
struct termios  *tty_termios(void);     /* Figure 18.20 */
#endif

void     sleep_us(unsigned int);            /* Exercise 14.6 */
ssize_t  readn(int, void *, size_t);        /* Figure 14.29 */
ssize_t  writen(int, const void *, size_t); /* Figure 14.29 */
void     daemonize(const char *);           /* Figure 13.1 */

int      s_pipe(int *);                 /* Figures 17.6 and 17.13 */
int      recv_fd(int, ssize_t (*func)(int,
                const void *, size_t));/* Figures 17.21 and 17.23 */
int      send_fd(int, int);             /* Figures 17.20 and 17.22 */
int      send_err(int, int,
                  const char *);        /* Figure 17.19 */
int      serv_listen(const char *);     /* Figures 17.10 and 17.15 */
int      serv_accept(int, uid_t *);     /* Figures 17.11 and 17.16 */
```

```
int     cli_conn(const char *);         /* Figures 17.12 and 17.17 */
int     buf_args(char *, int (*func)(int,
                char **));              /* Figure 17.32 */

int     ptym_open(char *, int);     /* Figures 19.8, 19.9, and 19.10 */
int     ptys_open(char *);          /* Figures 19.8, 19.9, and 19.10 */
#ifdef  TIOCGWINSZ
pid_t   pty_fork(int *, char *, int, const struct termios *,
                const struct winsize *);        /* Figure 19.11 */
#endif

int     lock_reg(int, int, int, off_t, int, off_t); /* Figure 14.5 */
#define read_lock(fd, offset, whence, len) \
            lock_reg((fd), F_SETLK, F_RDLCK, (offset), (whence), (len))
#define readw_lock(fd, offset, whence, len) \
            lock_reg((fd), F_SETLKW, F_RDLCK, (offset), (whence), (len))
#define write_lock(fd, offset, whence, len) \
            lock_reg((fd), F_SETLK, F_WRLCK, (offset), (whence), (len))
#define writew_lock(fd, offset, whence, len) \
            lock_reg((fd), F_SETLKW, F_WRLCK, (offset), (whence), (len))
#define un_lock(fd, offset, whence, len) \
            lock_reg((fd), F_SETLK, F_UNLCK, (offset), (whence), (len))

pid_t   lock_test(int, int, off_t, int, off_t);     /* Figure 14.6 */

#define is_read_lockable(fd, offset, whence, len) \
            (lock_test((fd), F_RDLCK, (offset), (whence), (len)) == 0)
#define is_write_lockable(fd, offset, whence, len) \
            (lock_test((fd), F_WRLCK, (offset), (whence), (len)) == 0)

void    err_dump(const char *, ...);        /* Appendix B */
void    err_msg(const char *, ...);
void    err_quit(const char *, ...);
void    err_exit(int, const char *, ...);
void    err_ret(const char *, ...);
void    err_sys(const char *, ...);

void    log_msg(const char *, ...);         /* Appendix B */
void    log_open(const char *, int, int);
void    log_quit(const char *, ...);
void    log_ret(const char *, ...);
void    log_sys(const char *, ...);

void    TELL_WAIT(void);        /* parent/child from Section 8.9 */
void    TELL_PARENT(pid_t);
void    TELL_CHILD(pid_t);
void    WAIT_PARENT(void);
void    WAIT_CHILD(void);

#endif  /* _APUE_H */
```

Figure B.1 Our header: apue.h

The reasons we include our header before all the normal system headers are to allow us to define anything that might be required by headers before they are included, control the order in which header files are included, and allow us to redefine anything that needs to be fixed up to hide the differences between systems.

B.2 Standard Error Routines

Two sets of error functions are used in most of the examples throughout the text to handle error conditions. One set begins with `err_` and outputs an error message to standard error. The other set begins with `log_` and is for daemon processes (Chapter 13) that probably have no controlling terminal.

The reason for our own error functions is to let us write our error handling with a single line of C code, as in

```
if (error condition)
        err_dump(printf format with any number of arguments);
```

instead of

```
if (error condition) {
        char   buf[200];
        sprintf(buf, printf format with any number of arguments);
        perror(buf);
        abort();
}
```

Our error functions use the variable-length argument list facility from ISO C. See Section 7.3 of Kernighan and Ritchie [1988] for additional details. Be aware that this ISO C facility differs from the `varargs` facility provided by earlier systems (such as SVR3 and 4.3BSD). The names of the macros are the same, but the arguments to some of the macros have changed.

Figure B.2 summarizes the differences between the various error functions.

Function	Adds string from `strerror`?	Parameter to `strerror`	Terminate?
`err_dump`	yes	`errno`	`abort();`
`err_exit`	yes	explicit parameter	`exit(1);`
`err_msg`	no		`return;`
`err_quit`	no		`exit(1);`
`err_ret`	yes	`errno`	`return;`
`err_sys`	yes	`errno`	`exit(1);`
`log_msg`	no		`return;`
`log_quit`	no		`exit(2);`
`log_ret`	yes	`errno`	`return;`
`log_sys`	yes	`errno`	`exit(2);`

Figure B.2 Our standard error functions

Figure B.3 shows the error functions that output to standard error.

```
#include "apue.h"
#include <errno.h>      /* for definition of errno */
#include <stdarg.h>     /* ISO C variable aruments */

static void err_doit(int, int, const char *, va_list);

/*
 * Nonfatal error related to a system call.
 * Print a message and return.
 */
void
err_ret(const char *fmt, ...)
{
    va_list     ap;

    va_start(ap, fmt);
    err_doit(1, errno, fmt, ap);
    va_end(ap);
}

/*
 * Fatal error related to a system call.
 * Print a message and terminate.
 */
void
err_sys(const char *fmt, ...)
{
    va_list     ap;

    va_start(ap, fmt);
    err_doit(1, errno, fmt, ap);
    va_end(ap);
    exit(1);
}

/*
 * Fatal error unrelated to a system call.
 * Error code passed as explict parameter.
 * Print a message and terminate.
 */
void
err_exit(int error, const char *fmt, ...)
{
    va_list     ap;

    va_start(ap, fmt);
    err_doit(1, error, fmt, ap);
    va_end(ap);
    exit(1);
}

/*
```

```
 * Fatal error related to a system call.
 * Print a message, dump core, and terminate.
 */
void
err_dump(const char *fmt, ...)
{
    va_list     ap;

    va_start(ap, fmt);
    err_doit(1, errno, fmt, ap);
    va_end(ap);
    abort();        /* dump core and terminate */
    exit(1);        /* shouldn't get here */
}

/*
 * Nonfatal error unrelated to a system call.
 * Print a message and return.
 */
void
err_msg(const char *fmt, ...)
{
    va_list     ap;

    va_start(ap, fmt);
    err_doit(0, 0, fmt, ap);
    va_end(ap);
}

/*
 * Fatal error unrelated to a system call.
 * Print a message and terminate.
 */
void
err_quit(const char *fmt, ...)
{
    va_list     ap;

    va_start(ap, fmt);
    err_doit(0, 0, fmt, ap);
    va_end(ap);
    exit(1);
}

/*
 * Print a message and return to caller.
 * Caller specifies "errnoflag".
 */
static void
err_doit(int errnoflag, int error, const char *fmt, va_list ap)
{
    char    buf[MAXLINE];
```

```
    vsnprintf(buf, MAXLINE, fmt, ap);
    if (errnoflag)
        snprintf(buf+strlen(buf), MAXLINE-strlen(buf), ": %s",
          strerror(error));
    strcat(buf, "\n");
    fflush(stdout);     /* in case stdout and stderr are the same */
    fputs(buf, stderr);
    fflush(NULL);       /* flushes all stdio output streams */
}
```

Figure B.3 Error functions that output to standard error

Figure B.4 shows the `log_XXX` error functions. These require the caller to define the variable `log_to_stderr` and set it nonzero if the process is not running as a daemon. In this case, the error messages are sent to standard error. If the `log_to_stderr` flag is 0, the `syslog` facility (Section 13.4) is used.

```
/*
 * Error routines for programs that can run as a daemon.
 */

#include "apue.h"
#include <errno.h>      /* for definition of errno */
#include <stdarg.h>     /* ISO C variable arguments */
#include <syslog.h>

static void log_doit(int, int, const char *, va_list ap);

/*
 * Caller must define and set this: nonzero if
 * interactive, zero if daemon
 */
extern int  log_to_stderr;

/*
 * Initialize syslog(), if running as daemon.
 */
void
log_open(const char *ident, int option, int facility)
{
    if (log_to_stderr == 0)
        openlog(ident, option, facility);
}

/*
 * Nonfatal error related to a system call.
 * Print a message with the system's errno value and return.
 */
void
log_ret(const char *fmt, ...)
{
    va_list     ap;
```

```
    va_start(ap, fmt);
    log_doit(1, LOG_ERR, fmt, ap);
    va_end(ap);
}

/*
 * Fatal error related to a system call.
 * Print a message and terminate.
 */
void
log_sys(const char *fmt, ...)
{
    va_list     ap;

    va_start(ap, fmt);
    log_doit(1, LOG_ERR, fmt, ap);
    va_end(ap);
    exit(2);
}

/*
 * Nonfatal error unrelated to a system call.
 * Print a message and return.
 */
void
log_msg(const char *fmt, ...)
{
    va_list     ap;

    va_start(ap, fmt);
    log_doit(0, LOG_ERR, fmt, ap);
    va_end(ap);
}

/*
 * Fatal error unrelated to a system call.
 * Print a message and terminate.
 */
void
log_quit(const char *fmt, ...)
{
    va_list     ap;

    va_start(ap, fmt);
    log_doit(0, LOG_ERR, fmt, ap);
    va_end(ap);
    exit(2);
}

/*
 * Print a message and return to caller.
 * Caller specifies "errnoflag" and "priority".
 */
```

```
static void
log_doit(int errnoflag, int priority, const char *fmt, va_list ap)
{
    int     errno_save;
    char    buf[MAXLINE];

    errno_save = errno;     /* value caller might want printed */
    vsnprintf(buf, MAXLINE, fmt, ap);
    if (errnoflag)
        snprintf(buf+strlen(buf), MAXLINE-strlen(buf), ": %s",
          strerror(errno_save));
    strcat(buf, "\n");
    if (log_to_stderr) {
        fflush(stdout);
        fputs(buf, stderr);
        fflush(stderr);
    } else {
        syslog(priority, buf);
    }
}
```

Figure B.4 Error functions for daemons

Appendix C

Solutions to Selected Exercises

Chapter 1

1.1 For this exercise, we use the following two arguments for the `ls`(1) command: `-i` prints the i-node number of the file or directory (we say more about i-nodes in Section 4.14), and `-d` prints information about a directory instead of information on all the files in the directory.

Execute the following:

```
$ ls -ldi /etc/. /etc/..          -i says print i-node number
 162561 drwxr-xr-x  66 root    4096 Feb  5 03:59 /etc/./
      2 drwxr-xr-x  19 root    4096 Jan 15 07:25 /etc/../
$ ls -ldi /. /..                  both . and .. have i-node number 2
      2 drwxr-xr-x  19 root    4096 Jan 15 07:25 /./
      2 drwxr-xr-x  19 root    4096 Jan 15 07:25 /../
```

1.2 The UNIX System is a multiprogramming, or multitasking, system. Other processes were running at the time this program was run.

1.3 Since the *ptr* argument to `perror` is a pointer, `perror` could modify the string that *ptr* points to. The qualifier `const`, however, says that `perror` does not modify what the pointer points to. On the other hand, the error number argument to `strerror` is an integer, and since C passes all arguments by value,

the `strerror` function couldn't modify this value even if it wanted to. (If the handling of function arguments in C is not clear, you should review Section 5.2 of Kernighan and Ritchie [1988].)

1.4 It is possible for the calls to `fflush`, `fprintf`, and `vprintf` to modify `errno`. If they did modify its value and we didn't save it, the error message finally printed would be incorrect.

1.5 During the year 2038. We can solve the problem by making the `time_t` data type a 64-bit integer. If it is currently a 32-bit integer, applications will have to be recompiled to work properly. But the problem is worse. Some file systems and backup media store times in 32-bit integers. These would need to be updated as well, but we still need to be able to read the old format.

1.6 Approximately 248 days.

Chapter 2

2.1 The following technique is used by FreeBSD. The primitive data types that can appear in multiple headers are defined in the header `<machine/_types.h>`. For example:

```
#ifndef _MACHINE__TYPES_H_
#define _MACHINE__TYPES_H_

typedef int             __int32_t;
typedef unsigned int    __uint32_t;
...

typedef __uint32_t      __size_t;
...

#endif  /* _MACHINE__TYPES_H_ */
```

In each of the headers that can define the `size_t` primitive system data type, we have the sequence

```
#ifndef  _SIZE_T_DECLARED
typedef __size_t        size_t;
#define _SIZE_T_DECLARED
#endif
```

This way, the `typedef` for `size_t` is executed only once.

2.3 If `OPEN_MAX` is indeterminite or ridiculously large (i.e., equal to `LONG_MAX`), we can use `getrlimit` to get the per process maximum for open file descriptors. Since the per process limit can be modified, we can't cache the value obtained from the previous call (it might have changed). See Figure C.1.

```
#include "apue.h"
#include <limits.h>
#include <sys/resource.h>

#define OPEN_MAX_GUESS  256

long
open_max(void)
{
    long openmax;
    struct rlimit rl;

    if ((openmax = sysconf(_SC_OPEN_MAX)) < 0 ||
      openmax == LONG_MAX) {
        if (getrlimit(RLIMIT_NOFILE, &rl) < 0)
            err_sys("can't get file limit");
        if (rl.rlim_max == RLIM_INFINITY)
            openmax = OPEN_MAX_GUESS;
        else
            openmax = rl.rlim_max;
    }
    return(openmax);
}
```

Figure C.1 Alternate method for identifying the largest possible file descriptor

Chapter 3

3.1 All disk I/O goes through the kernel's block buffers (also called the kernel's buffer cache). The exception to this is I/O on a raw disk device, which we aren't considering. Chapter 3 of Bach [1986] describes the operation of this buffer cache. Since the data that we `read` or `write` is buffered by the kernel, the term *unbuffered I/O* refers to the lack of automatic buffering in the user process with these two functions. Each `read` or `write` invokes a single system call.

3.3 Each call to open gives us a new file table entry. But since both opens reference the same file, both file table entries point to the same v-node table entry. The call to dup references the existing file table entry. We show this in Figure C.2. An `F_SETFD` on `fd1` affects only the file descriptor flags for `fd1`. But an `F_SETFL` on `fd1` affects the file table entry that both `fd1` and `fd2` point to.

3.4 If `fd` is 1, then the `dup2(fd, 1)` returns 1 without closing descriptor 1. (Remember our discussion of this in Section 3.12.) After the three calls to dup2, all three descriptors point to the same file table entry. Nothing needs to be closed.

If `fd` is 3, however, after the three calls to dup2, four descriptors are pointing to the same file table entry. In this case, we need to close descriptor 3.

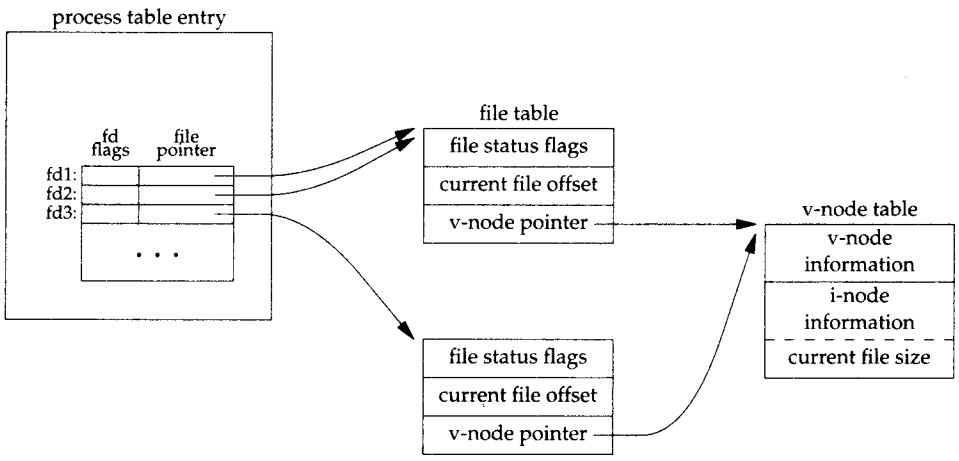

Figure C.2 Result of dup and open

3.5 Since the shells process their command line from left to right, the command

```
./a.out > outfile 2>&1
```

first sets standard output to `outfile` and then dups standard output onto descriptor 2 (standard error). The result is that standard output and standard error are set to the same file. Descriptors 1 and 2 both point to the same file table entry. With

```
./a.out 2>&1 > outfile
```

however, the dup is executed first, causing descriptor 2 to be the terminal (assuming that the command is run interactively). Then standard output is redirected to the file `outfile`. The result is that descriptor 1 points to the file table entry for `outfile`, and descriptor 2 points to the file table entry for the terminal.

3.6 You can still `lseek` and `read` anywhere in the file, but a `write` automatically resets the file offset to the end of file before the data is written. This makes it impossible to `write` anywhere other than at the end of file.

Chapter 4

4.1 If `stat` is called, it always tries to follow a symbolic link (Figure 4.17), so the program will never print a file type of "symbolic link." For the example shown in the text, where `/dev/cdrom` is a symbolic link to `cdroms/cdrom0` (which itself is a symbolic link to `../scsi/host0/bus0/target0/lun0/cd`), `stat` reports that `/dev/cdrom` is a block special file, not a symbolic link. If the symbolic link points to a nonexistent file, `stat` returns an error.

4.2 All permissions are turned off:

```
$ umask 777
$ date > temp.foo
$ ls -l temp.foo
----------  1 sar          0 Feb  5 14:06 temp.foo
```

4.3 The following shows what happens when user-read permission is turned off:

```
$ date > foo
$ chmod u-r foo                          turn off user-read permission
$ ls -l foo                              verify the file's permissions
--w-r--r--  1 sar         29 Feb  5 14:21 foo
$ cat foo                                and try to read it
cat: foo: Permission denied
```

4.4 If we try, using either open or `creat`, to create a file that already exists, the file's access permission bits are not changed. We can verify this by running the program from Figure 4.9:

```
$ rm foo bar                             delete the files in case they already exist
$ date > foo                             create them with some data
$ date > bar
$ chmod a-r foo bar                      turn off all read permissions
$ ls -l foo bar                          verify their permissions
--w-------  1 sar         29 Feb  5 14:25 bar
--w-------  1 sar         29 Feb  5 14:25 foo
$ ./a.out                                run program from Figure 4.9
$ ls -l foo bar                          check permissions and sizes
--w-------  1 sar          0 Feb  5 14:26 bar
--w-------  1 sar          0 Feb  5 14:26 foo
```

Note that the permissions didn't change but that the files were truncated.

4.5 The size of a directory should never be 0, since there should always be entries for dot and dot-dot. The size of a symbolic link is the number of characters in the pathname contained in the symbolic link, and this pathname must always contain at least one character.

4.7 The kernel has a default setting for the file access permission bits when it creates a new `core` file. In this example, it was `rw-r--r--`. This default value may or may not be modified by the `umask` value. The shell also has a default setting for the file access permission bits when it creates a new file for redirection. In this example, it was `rw-rw-rw-`, and this value is always modified by our current `umask`. In this example, our `umask` was 02.

4.8 We can't use du, because it requires either the name of the file, as in

```
du tempfile
```

or a directory name, as in

```
du
```

But when the `unlink` function returns, the directory entry for `tempfile` is gone.

The `du .` command just shown would not account for the space still taken by `tempfile`. We have to use the `df` command in this example to see the actual amount of free space on the file system.

4.9 If the link being removed is not the last link to the file, the file is not removed. In this case, the changed-status time of the file is updated. But if the link being removed is the last link to the file, it makes no sense to update this time, because all the information about the file (the i-node) is removed with the file.

4.10 We recursively call our function `dopath` after opening a directory with `opendir`. Assuming that `opendir` uses a single file descriptor, this means that each time we descend one level, we use another descriptor. (We assume that the descriptor isn't closed until we're finished with a directory and call `closedir`.) This limits the depth of the file system tree that we can traverse to the maximum number of open descriptors for the process. Note that the `ftw` function as specified in the XSI extensions of the Single UNIX Specification allows the caller to specify the number of descriptors to use, implying that it can close and reuse descriptors.

4.12 The `chroot` function is used by the Internet File Transfer Program (FTP) to aid in security. Users without accounts on a system (termed *anonymous FTP*) are placed in a separate directory, and a `chroot` is done to that directory. This prevents the user from accessing any file outside this new root directory.

In addition, `chroot` can be used to build a copy of a file system hierarchy at a new location and then modify this new copy without changing the original file system. This could be used, for example, to test the installation of new software packages.

Only the superuser can execute `chroot`, and once you change the root of a process, it (and all its descendants) can never get back to the original root.

4.13 First, call `stat` to fetch the three times for the file; then call `utime` to set the desired value. The value that we don't want to change in the call to `utime` should be the corresponding value from `stat`.

4.14 The `finger`(1) command calls `stat` on the mailbox. The last-modification time is the time that mail was last received, and the last-access time is when the mail was last read.

4.15 Both `cpio` and `tar` store only the modification time (`st_mtime`) on the archive. The access time isn't stored, because its value corresponds to the time the archive was created, since the file has to be read to be archived. The `-a` option to `cpio` has it reset the access time of each input file after the file has been read. This way, the creation of the archive doesn't change the access time. (Resetting the access time, however, does modify the changed-status time.) The changed-status time isn't stored on the archive, because we can't set this value on extraction even if it was archived. (The `utime` function can change only the access time and the modification time.)

When the archive is read back (extracted), `tar`, by default, restores the modification time to the value on the archive. The `m` option to `tar` tells it to not

restore the modification time from the archive; instead, the modification time is set to the time of extraction. In all cases with `tar`, the access time after extraction will be the time of extraction.

On the other hand, `cpio` sets the access time and the modification time to the time of extraction. By default, it doesn't try to set the modification time to the value on the archive. The `-m` option to `cpio` has it set both the access time and the modification time to the value that was archived.

4.16 The kernel has no inherent limit on the depth of a directory tree. But many commands will fail on pathnames that exceed `PATH_MAX`. The program shown in Figure C.3 creates a directory tree that is 100 levels deep, with each level being a 45-character name. We are able to create this structure on all platforms; however, we cannot obtain the absolute pathname of the directory at the 100th level using `getcwd` on all platforms. On Linux 2.4.22 and Solaris 9, we can never get `getcwd` to succeed while in the directory at the end of this long path. The program is able to retrieve the pathname on FreeBSD 5.2.1 and Mac OS X 10.3, but we have to call `realloc` numerous times to obtain a buffer that is large enough. Running this program on FreeBSD 5.2.1 gives us

```
$ ./a.out
getcwd failed, size = 1025: Result too large
getcwd failed, size = 1125: Result too large
...                          33 more lines
getcwd failed, size = 4525: Result too large
length = 4610
                             the 4,610-byte pathname is printed here
```

We are not able to archive this directory, however, using either `tar` or `cpio`. Both complain of a filename that is too long.

4.17 The `/dev` directory has all write permissions turned off to prevent a normal user from removing the filenames in the directory. This means that the `unlink` fails.

Chapter 5

5.2 The `fgets` function reads up through and including the next newline *or* until the buffer is full (leaving room, of course, for the terminating null). Also, `fputs` writes everything in the buffer until it hits a null byte; it doesn't care whether a newline is in the buffer. So, if `MAXLINE` is too small, both functions still work; they're just called more often than they would be if the buffer were larger.

If either of these functions removed or added the newline (as `gets` and `puts` do), we would have to ensure that our buffer was big enough for the largest line.

5.3 The function call

```
printf("");
```

returns 0, since no characters are output.

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

#define DEPTH   100         /* directory depth */
#define MYHOME  "/home/sar"
#define NAME    "alonglonglonglonglonglonglonglonglongname"
#define MAXSZ   8192

int
main(void)
{
    int     i, size;
    char    *path;

    if (chdir(MYHOME) < 0)
        err_sys("chdir error");

    for (i = 0; i < DEPTH; i++) {
        if (mkdir(NAME, DIR_MODE) < 0)
            err_sys("mkdir failed, i = %d", i);
        if (chdir(NAME) < 0)
            err_sys("chdir failed, i = %d", i);
    }
    if (creat("afile", FILE_MODE) < 0)
        err_sys("creat error");

    /*
     * The deep directory is created, with a file at the leaf.
     * Now let's try to obtain its pathname.
     */
    path = path_alloc(&size);
    for ( ; ; ) {
        if (getcwd(path, size) != NULL) {
            break;
        } else {
            err_ret("getcwd failed, size = %d", size);
            size += 100;
            if (size > MAXSZ)
                err_quit("giving up");
            if ((path = realloc(path, size)) == NULL)
                err_sys("realloc error");
        }
    }
    printf("length = %d\n%s\n", strlen(path), path);

    exit(0);
}
```

Figure C.3 Create a deep directory tree

5.4 This is a common error. The return value from `getc` and `getchar` is an `int`, not a `char`. `EOF` is often defined to be –1, so if the system uses signed characters, the code normally works. But if the system uses unsigned characters, after the `EOF` returned by `getchar` is stored as an unsigned character, the character's value no longer equals –1, so the loop never terminates. The four platforms described in this book all use signed characters, so the example code works on these platforms.

5.5 A 5-character prefix, a 4-character per process unique identifier, and a 5-character per system unique identifier (the process ID) equals 14 characters, the original UNIX System limit on the length of a filename.

5.6 Call `fsync` after each call to `fflush`. The argument to `fsync` is obtained with the `fileno` function. Calling `fsync` without calling `fflush` might do nothing if all the data were still in memory buffers.

5.7 Standard input and standard output are both line buffered when a program is run interactively. When `fgets` is called, standard output is flushed automatically.

Chapter 6

6.1 The functions to access the shadow password file on Linux and Solaris are discussed in Section 6.3. We can't use the value returned in the `pw_passwd` field by the functions described in Section 6.2 to compare an encrypted password, since that field is not the encrypted password. Instead, we need to find the user's entry in the shadow file and use its encrypted password field.

On FreeBSD and Mac OS X, the password file is shadowed automatically. In the `passwd` structure returned by `getpwnam` and `getpwuid`, the `pw_passwd` field contains the encrypted password (only if the caller's effective user ID is 0 on FreeBSD, however).

6.2 The program in Figure C.4 prints the encrypted password on Linux and Solaris. Unless this program is run with superuser permissions, the call to `getspnam` fails with an error of `EACCES`.

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

int
main(void)      /* Linux/Solaris version */
{
    struct spwd *ptr;

    if ((ptr = getspnam("sar")) == NULL)
        err_sys("getspnam error");
    printf("sp_pwdp = %s\n", ptr->sp_pwdp == NULL ||
      ptr->sp_pwdp[0] == 0 ?  "(null)" : ptr->sp_pwdp);
    exit(0);
}
```

Figure C.4 Print encrypted password under Linux and Solaris

Under FreeBSD, the program in Figure C.5 prints the encrypted password if the program is run with superuser permissions. Otherwise, the value returned in pw_passwd is an asterisk. On Mac OS X, the encrypted password is printed regardless of the permissions with which it is run.

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

int
main(void)      /* FreeBSD/Mac OS X version */
{
    struct passwd   *ptr;

    if ((ptr = getpwnam("sar")) == NULL)
        err_sys("getpwnam error");
    printf("pw_passwd = %s\n", ptr->pw_passwd == NULL ||
      ptr->pw_passwd[0] == 0 ?  "(null)" : ptr->pw_passwd);
    exit(0);
}
```

Figure C.5 Print encrypted password under FreeBSD and Mac OS X

6.5 The program shown in Figure C.6 prints the date in a format similar to `date`.

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

int
main(void)
{
    time_t      caltime;
    struct tm   *tm;
    char        line[MAXLINE];

    if ((caltime = time(NULL)) == -1)
        err_sys("time error");
    if ((tm = localtime(&caltime)) == NULL)
        err_sys("localtime error");
    if (strftime(line, MAXLINE, "%a %b %d %X %Z %Y\n", tm) == 0)
        err_sys("strftime error");
    fputs(line, stdout);
    exit(0);
}
```

Figure C.6 Print the time and date in a format similar to `date`(1)

Running this program gives us

```
$ ./a.out                          author's default is US/Eastern
Sun Feb 06 16:53:57 EST 2005
$ TZ=US/Mountain ./a.out           U.S. Mountain time zone
Sun Feb 06 14:53:57 MST 2005
$ TZ=Japan ./a.out                 Japan
Mon Feb 07 06:53:57 JST 2005
```

Chapter 7

7.1 It appears that the return value from `printf` (the number of characters output) becomes the return value of `main`. Not all systems exhibit this property.

7.2 When the program is run interactively, standard output is usually line buffered, so the actual output occurs when each newline is output. If standard output were directed to a file, however, it would probably be fully buffered, and the actual output wouldn't occur until the standard I/O cleanup is performed.

7.3 On most UNIX systems, there is no way to do this. Copies of `argc` and `argv` are not kept in global variables like `environ` is.

7.4 This provides a way to terminate the process when it tries to dereference a null pointer, a common C programming error.

7.5 The definitions are

```
typedef void    Exitfunc(void);

int atexit(Exitfunc *func);
```

7.6 `calloc` initializes the memory that it allocates to all zero bits. ISO C does not guarantee that this is the same as either a floating-point 0 or a null pointer.

7.7 The heap and the stack aren't allocated until a program is executed by one of the `exec` functions (described in Section 8.10).

7.8 The executable file (`a.out`) contains symbol table information that can be helpful in debugging a `core` file. To remove this information, the `strip`(1) command is used. Stripping the two `a.out` files reduces their size to 381,976 and 2,912 bytes.

7.9 When shared libraries are not used, a large portion of the executable file is occupied by the standard I/O library.

7.10 The code is incorrect, since it references the automatic integer `val` through a pointer after the automatic variable is no longer in existence. Automatic variables declared after the left brace that starts a compound statement disappear after the matching right brace.

Chapter 8

8.1 To simulate the behavior of the child closing the standard output when it exits, add the following line before calling `exit` in the child:

```
fclose(stdout);
```

To see the effects of doing this, replace the call to `printf` with the lines

```
i = printf("pid = %d, glob = %d, var = %d\n",
    getpid(), glob, var);
sprintf(buf, "%d\n", i);
write(STDOUT_FILENO, buf, strlen(buf));
```

You need to define the variables `i` and `buf` also.

This assumes that the standard I/O stream `stdout` is closed when the child calls `exit`, not the file descriptor `STDOUT_FILENO`. Some versions of the standard I/O library close the file descriptor associated with standard output, which would cause the `write` to standard output to also fail. In this case, dup standard output to another descriptor, and use this new descriptor for the `write`.

8.2 Consider Figure C.7.

```
#include "apue.h"

static void f1(void), f2(void);

int
main(void)
{
    f1();
    f2();
    _exit(0);
}

static void
f1(void)
{
    pid_t   pid;

    if ((pid = vfork()) < 0)
        err_sys("vfork error");
    /* child and parent both return */
}

static void
f2(void)
{
    char    buf[1000];      /* automatic variables */
    int     i;

    for (i = 0; i < sizeof(buf); i++)
        buf[i] = 0;
}
```

Figure C.7 Incorrect use of `vfork`

When `vfork` is called, the parent's stack pointer points to the stack frame for the `f1` function that calls `vfork`. Figure C.8 shows this.

`vfork` causes the child to execute first, and the child returns from `f1`. The child then calls `f2`, and its stack frame overwrites the previous stack frame for `f1`. The child then zeros out the automatic variable `buf`, setting 1,000 bytes of the stack frame to 0. The child returns from `f2` and then calls `_exit`, but the contents of

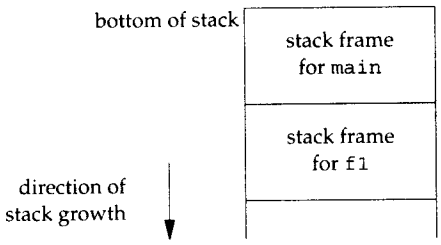

Figure C.8 Stack frames when `vfork` is called

the stack beneath the stack frame for `main` have been changed. The parent then resumes after the call to `vfork` and does a return from `f1`. The return information is often stored in the stack frame, and that information has probably been modified by the child. After the parent resumes, what happens with this example depends on many implementation features of your UNIX system (where in the stack frame the return information is stored, what information in the stack frame is wiped out when the automatic variables are modified, and so on). The normal result is a `core` file, but your results may differ.

8.3 In Figure 8.13, we have the parent output first. When the parent is done, the child writes its output, but we let the parent terminate. Whether the parent terminates or whether the child finishes its output first depends on the kernel's scheduling of the two processes (another race condition). When the parent terminates, the shell starts up the next program, and this next program can interfere with the output from the previous child.

We can prevent this from happening by not letting the parent terminate until the child has also finished its output. Replace the code following the `fork` with the following:

```
else if (pid == 0) {
    WAIT_PARENT();          /* parent goes first */
    charatatime("output from child\n");
    TELL_PARENT(getppid()); /* tell parent we're done */
} else {
    charatatime("output from parent\n");
    TELL_CHILD(pid);        /* tell child we're done */
    WAIT_CHILD();           /* wait for child to finish */
}
```

We won't see this happen if we let the child go first, since the shell doesn't start the next program until the parent terminates.

8.4 The same value (`/home/sar/bin/testinterp`) is printed for `argv[2]`. The reason is that `execlp` ends up calling `execve` with the same *pathname* as when we call `execl` directly. Recall Figure 8.15.

8.5 A function is not provided to return the saved set-user-ID. Instead, we must save the effective user ID when the process is started.

8.6 The program in Figure C.9 creates a zombie.

```
#include "apue.h"

#ifdef SOLARIS
#define PSCMD   "ps -a -o pid,ppid,s,tty,comm"
#else
#define PSCMD   "ps -o pid,ppid,state,tty,command"
#endif

int
main(void)
{
    pid_t   pid;

    if ((pid = fork()) < 0)
        err_sys("fork error");
    else if (pid == 0)      /* child */
        exit(0);

    /* parent */
    sleep(4);
    system(PSCMD);

    exit(0);
}
```

Figure C.9 Create a zombie and look at its status with ps

Zombies are usually designated by ps(1) with a status of Z:

```
$ ./a.out
  PID  PPID S TT       COMMAND
 3395  3264 S pts/3    bash
29520  3395 S pts/3    ./a.out
29521 29520 Z pts/3    [a.out] <defunct>
29522 29520 R pts/3    ps -o pid,ppid,state,tty,command
```

Chapter 9

9.1 The `init` process learns when a terminal user logs out, because `init` is the parent of the login shell and receives the `SIGCHLD` signal when the login shell terminates.

For a network login, however, `init` is not involved. Instead, the login entries in the `utmp` and `wtmp` files, and their corresponding logout entries, are usually written by the process that handles the login and detects the logout (`telnetd` in our example).

Chapter 10

10.1 The program terminates the first time we send it a signal. The reason is that the `pause` function returns whenever a signal is caught.

10.3 Figure C.10 shows the stack frames.

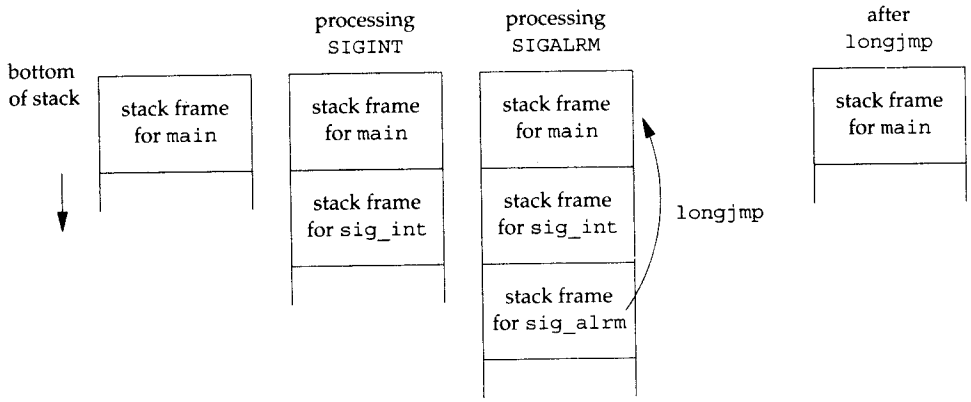

Figure C.10 Stack frames before and after `longjmp`

The `longjmp` from `sig_alrm` back to `main` effectively aborts the call to `sig_int`.

10.4 We again have a race condition, this time between the first call to `alarm` and the call to `setjmp`. If the process is blocked by the kernel between these two function calls, the alarm goes off, the signal handler is called, and `longjmp` is called. But since `setjmp` was never called, the buffer `env_alrm` is not set. The operation of `longjmp` is undefined if its jump buffer has not been initialized by `setjmp`.

10.5 See "Implementing Software Timers" by Don Libes (*C Users Journal*, vol. 8, no. 11, Nov. 1990) for an example.

10.7 If we simply called `_exit`, the termination status of the process would not show that it was terminated by the `SIGABRT` signal.

10.8 If the signal was sent by a process owned by some other user, the process has to be set-user-ID to either root or to the owner of the receiving process, or the `kill` won't work. Therefore, the real user ID provides more information to the receiver of the signal.

10.10 On one system used by the author, the value for the number of seconds increased by 1 about every 60–90 minutes. This skew occurs because each call to `sleep` schedules an event for a time in the future, but is not awakened exactly when that event occurs (because of CPU scheduling). In addition, a finite amount of time is required for our process to start running and call `sleep` again.

A program such as the `cron` daemon has to fetch the current time every minute, as well as to set its first sleep period so that it wakes up at the beginning of the next minute. (Convert the current time to the local time and look at the `tm_sec` value.) Every minute, it sets the next sleep period so that it'll wake up at the next minute. Most of the calls will probably be `sleep(60)`, with an occasional `sleep(59)` to resynchronize with the next minute. But if at some point the process takes a long time executing commands or if the system gets heavily loaded and scheduling delays hold up the process, the sleep value can be much less than 60.

10.11 Under Linux 2.4.22 and Solaris 9, the signal handler for `SIGXFSZ` is never called. But `write` returns a count of 24 as soon as the file's size reaches 1,024 bytes.

When the file's size has reached 1,000 bytes under FreeBSD 5.2.1 and Mac OS X 10.3, the signal handler is called on the next attempt to write 100 bytes, and the `write` call returns –1 with `errno` set to `EFBIG` ("File too big").

10.12 The results depend on the implementation of the standard I/O library: how the `fwrite` function handles an interrupted `write`.

Chapter 11

11.1 A version of the program that allocates the memory dynamically instead of using an automatic variable is shown in Figure C.11.

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

struct foo {
    int a, b, c, d;
};

void
printfoo(const char *s, const struct foo *fp)
{
    printf(s);
    printf("  structure at 0x%x\n", (unsigned)fp);
    printf("  foo.a = %d\n", fp->a);
    printf("  foo.b = %d\n", fp->b);
    printf("  foo.c = %d\n", fp->c);
    printf("  foo.d = %d\n", fp->d);
}

void *
thr_fn1(void *arg)
{
    struct foo *fp;

    if ((fp = malloc(sizeof(struct foo))) == NULL)
        err_sys("can't allocate memory");
    fp->a = 1;
```

```
    fp->b = 2;
    fp->c = 3;
    fp->d = 4;
    printfoo("thread:\n", fp);
    return((void *)fp);
}

int
main(void)
{
    int err;
    pthread_t tid1;
    struct foo *fp;

    err = pthread_create(&tid1, NULL, thr_fn1, NULL);
    if (err != 0)
        err_exit(err, "can't create thread 1");
    err = pthread_join(tid1, (void *)&fp);
    if (err != 0)
        err_exit(err, "can't join with thread 1");
    printfoo("parent:\n", fp);
    exit(0);
}
```

Figure C.11 Correct use of thread return value

11.2 To change the thread ID of a pending job, the reader–writer lock must be held in write mode to prevent anyone from searching the list while the ID is being changed. The problem with the way the interfaces are currently defined is that the ID of a job can change between the time that the job is found with `job_find` and the job is removed from the list by calling `job_remove`. This problem can be solved by embedding a reference count and a mutex inside the `job` structure and having `job_find` increment the reference count. The code that changes the ID can then avoid any job in the list that has a nonzero reference count.

11.3 First of all, the list is protected by a reader–writer lock, but the condition variable needs a mutex to protect the condition. Second, the condition each thread should wait to be satisfied is that there is a job for it to process, so we need to create a per thread data structure to represent this condition. Alternatively, we can embed the mutex and condition variable in the queue structure, but this means that all worker threads will wait on the same condition. If there are many worker threads, we can run into a *thundering herd* problem, whereby many threads are awakened without work to do, resulting in a waste of CPU resources and increased lock contention.

11.4 It depends on the circumstances. In general, both can be correct, but each alternative has drawbacks. In the first sequence, the waiting threads will be scheduled to run after we call `pthread_cond_broadcast`. If the program is running on a multiprocessor, some threads will run and immediately block because we are still holding the mutex (recall that `pthread_cond_wait` returns

with the mutex held). In the second sequence, a running thread can acquire the mutex between steps 3 and 4, invalidate the condition, and release the mutex. Then, when we call `pthread_cond_broadcast`, the condition will no longer be true, and the threads will run needlessly. This is why the awakened threads must recheck the condition and not assume that it is true merely because `pthread_cond_wait` returned.

Chapter 12

12.1 This is not a multithreading problem, as one might first guess. The standard I/O routines are indeed thread-safe. When we call `fork`, each process gets a copy of the standard I/O data structures. When we run the program with standard output attached to a terminal, the output is line buffered, so every time we print a line, the standard I/O library writes it to our terminal. However, if we redirect the standard output to a file, then the standard output is fully buffered. The output is written when the buffer fills or the process closes the stream. When we `fork` in this example, the buffer contains several printed lines not yet written, so when the parent and the child finally flush their copies of the buffer, the initial duplicate contents are written to the file.

12.3 Theoretically, if we arrange for all signals to be blocked when the signal handler runs, we should be able to make a function async-signal-safe. The problem is that we don't know whether any of the functions we call might unmask a signal that we've blocked, thereby making it possible for the function to be reentered through another signal handler.

12.4 On FreeBSD 5.2.1, we get a continuous spray of error messages, and after a while, the program drops core. With `gdb`, we are able to see that the program was stuck in an infinite loop during initialization. The program initialization calls thread initialization functions, which call `malloc`. The `malloc` function, in turn, calls `getenv` to find the value of the `MALLOC_OPTIONS` environment variable. Our implementation of `getenv` calls pthread functions, which then try to call the thread initialization functions. Eventually, we hit an error and get stuck in a similar infinite loop after `abort` is called. After half a million or so stack frames, the process exits, generating a core dump.

12.5 We still need `fork` if we want to run a program from within another program (i.e., before calling `exec`).

12.6 Figure C.12 shows a thread-safe `sleep` implementation that uses `select` to delay for the specified amount of time. It is thread-safe because it doesn't use any unprotected global or static data and calls only other thread-safe functions.

12.7 The implementation of a condition variable most likely uses a mutex to protect its internal structure. Because this is an implementation detail and therefore hidden, there is no portable way for us to acquire and release the lock in the fork handlers. Since we can't determine the state of the internal lock in a condition variable after calling `fork`, it is unsafe for us to use the condition variable in the child process.

```
#include <unistd.h>
#include <time.h>
#include <sys/select.h>

unsigned
sleep(unsigned nsec)
{
    int n;
    unsigned slept;
    time_t start, end;
    struct timeval tv;

    tv.tv_sec = nsec;
    tv.tv_usec = 0;
    time(&start);
    n = select(0, NULL, NULL, NULL, &tv);
    if (n == 0)
        return(0);
    time(&end);
    slept = end - start;
    if (slept >= nsec)
        return(0);
    return(nsec - slept);
}
```

Figure C.12 A thread-safe implementation of `sleep`

Chapter 13

13.1 If it calls `chroot`, the process will not be able to open `/dev/log`. The solution is for the daemon to call `openlog` with an *option* of `LOG_NDELAY`, before calling `chroot`. This opens the special device file (the UNIX domain datagram socket), yielding a descriptor that is still valid, even after a call to `chroot`. This scenario is encountered in daemons, such as `ftpd` (the File Transfer Protocol daemon), that specifically call `chroot` for security reasons but still need to call `syslog` to log error conditions.

13.3 Figure C.13 shows a solution. The results depend on the platform. Recall that `daemonize` closes all open file descriptors and then reopens the first three to `/dev/null`. This means that the process won't have a controlling terminal, so `getlogin` won't be able to look in the `utmp` file for the process's login entry. Thus, on Linux 2.4.22 and Solaris 9, we find that a daemon has no login name.

Under FreeBSD 5.2.1 and Mac OS X 10.3, however, the login name is maintained in the process table and copied across a `fork`. This means that the process can always get the login name, unless the parent didn't have one to start out (such as `init` when the system is bootstrapped).

```
#include "apue.h"

int
main(void)
{
    FILE *fp;
    char *p;

    daemonize("getlog");
    p = getlogin();
    fp = fopen("/tmp/getlog.out", "w");
    if (fp != NULL) {
        if (p == NULL)
            fprintf(fp, "no login name\n");
        else
            fprintf(fp, "login name: %s\n", p);
    }
    exit(0);
}
```

Figure C.13 Call daemonize and then obtain login name

Chapter 14

14.1 The test program is shown in Figure C.14

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

void
sigint(int signo)
{
}

int
main(void)
{
    pid_t pid1, pid2, pid3;
    int fd;

    setbuf(stdout, NULL);
    signal_intr(SIGINT, sigint);

    /*
     * Create a file.
     */
    if ((fd = open("lockfile", O_RDWR|O_CREAT, 0666)) < 0)
        err_sys("can't open/create lockfile");

    /*
     * Read-lock the file.
     */
```

```
if ((pid1 = fork()) < 0) {
    err_sys("fork failed");
} else if (pid1 == 0) { /* child */
    if (lock_reg(fd, F_SETLK, F_RDLCK, 0, SEEK_SET, 0) < 0)
        err_sys("child 1: can't read-lock file");
    printf("child 1: obtained read lock on file\n");
    pause();
    printf("child 1: exit after pause\n");
    exit(0);
} else {        /* parent */
    sleep(2);
}

/*
 * Parent continues ... read-lock the file again.
 */
if ((pid2 = fork()) < 0) {
    err_sys("fork failed");
} else if (pid2 == 0) { /* child */
    if (lock_reg(fd, F_SETLK, F_RDLCK, 0, SEEK_SET, 0) < 0)
        err_sys("child 2: can't read-lock file");
    printf("child 2: obtained read lock on file\n");
    pause();
    printf("child 2: exit after pause\n");
    exit(0);
} else {        /* parent */
    sleep(2);
}

/*
 * Parent continues ... block while trying to write-lock
 * the file.
 */
if ((pid3 = fork()) < 0) {
    err_sys("fork failed");
} else if (pid3 == 0) { /* child */
    if (lock_reg(fd, F_SETLK, F_WRLCK, 0, SEEK_SET, 0) < 0)
        printf("child 3: can't set write lock: %s\n",
          strerror(errno));
    printf("child 3 about to block in write-lock...\n");
    if (lock_reg(fd, F_SETLKW, F_WRLCK, 0, SEEK_SET, 0) < 0)
        err_sys("child 3: can't write-lock file");
    printf("child 3 returned and got write lock????\n");
    pause();
    printf("child 3: exit after pause\n");
    exit(0);
} else {        /* parent */
    sleep(2);
}

/*
 * See if a pending write lock will block the next
```

```
         * read-lock attempt.
         */
        if (lock_reg(fd, F_SETLK, F_RDLCK, 0, SEEK_SET, 0) < 0)
            printf("parent: can't set read lock: %s\n",
              strerror(errno));
        else
            printf("parent: obtained additional read lock while"
              " write lock is pending\n");
        printf("killing child 1...\n");
        kill(pid1, SIGINT);
        printf("killing child 2...\n");
        kill(pid2, SIGINT);
        printf("killing child 3...\n");
        kill(pid3, SIGINT);
        exit(0);
}
```

Figure C.14 Determine record-locking behavior

On all four systems described in this book, the behavior is the same: additional readers can starve pending writers. Running the program gives us

```
child 1: obtained read lock on file
child 2: obtained read lock on file
child 3: can't set write lock: Resource temporarily unavailable
child 3 about to block in write-lock...
parent: obtained additional read lock while write lock is pending
killing child 1...
child 1: exit after pause
killing child 2...
child 2: exit after pause
killing child 3...
child 3: can't write-lock file: Interrupted system call
```

14.2 Most systems define the `fd_set` data type to be a structure that contains a single member: an array of long integers. One bit in this array corresponds to each descriptor. The four `FD_` macros then manipulate this array of longs, turning specific bits on and off and testing specific bits.

One reason that the data type is defined to be a structure containing an array and not simply an array is to allow variables of type `fd_set` to be assigned to one another with the C assignment statement.

14.3 Most systems allow us to define the constant `FD_SETSIZE` before including the header `<sys/select.h>`. For example, we can write

```
#define FD_SETSIZE  2048
#include <sys/select.h>
```

to define the `fd_set` data type to accommodate 2,048 descriptors. This works on FreeBSD 5.2.1, Mac OS X 10.3, and Solaris 9. Linux 2.4.22 implements things differently.

14.4 The following table lists the functions that do similar things.

FD_ZERO	sigemptyset
FD_SET	sigaddset
FD_CLR	sigdelset
FD_ISSET	sigismember

There is not an FD_xxx function that corresponds to sigfillset. With signal sets, the pointer to the set is always the first argument, and the signal number is the second argument. With descriptor sets, the descriptor number is the first argument, and the pointer to the set is the next argument.

14.5 Up to five types of information are returned by getmsg: the data itself, the length of the data, the control information, the length of the control information, and the flags.

14.6 Figure C.15 shows an implementation using select.

```
#include "apue.h"
#include <sys/select.h>

void
sleep_us(unsigned int nusecs)
{
    struct timeval  tval;

    tval.tv_sec = nusecs / 1000000;
    tval.tv_usec = nusecs % 1000000;
    select(0, NULL, NULL, NULL, &tval);
}
```

Figure C.15 Implementation of sleep_us using select

Figure C.16 shows an implementation using poll.

```
#include <poll.h>

void
sleep_us(unsigned int nusecs)
{
    struct pollfd   dummy;
    int             timeout;

    if ((timeout = nusecs / 1000) <= 0)
        timeout = 1;
    poll(&dummy, 0, timeout);
}
```

Figure C.16 Implementation of sleep_us using poll

As the BSD usleep(3) manual page states, usleep uses the setitimer interval timer and performs eight system calls each time it's called. It correctly interacts with other timers set by the calling process, and it is not interrupted if a signal is caught.

14.7 No. What we would like to do is have `TELL_WAIT` create a temporary file and use 1 byte for the parent's lock and 1 byte for the child's lock. `WAIT_CHILD` would have the parent wait to obtain a lock on the child's byte, and `TELL_PARENT` would have the child release the lock on the child's byte. The problem, however, is that calling `fork` releases all the locks in the child, so the child can't start off with any locks of its own.

14.8 A solution is shown in Figure C.17.

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

int
main(void)
{
    int i, n;
    int fd[2];

    if (pipe(fd) < 0)
        err_sys("pipe error");
    set_fl(fd[1], O_NONBLOCK);

    /*
     * Write 1 byte at a time until pipe is full.
     */
    for (n = 0; ; n++) {
        if ((i = write(fd[1], "a", 1)) != 1) {
            printf("write ret %d, ", i);
            break;
        }
    }
    printf("pipe capacity = %d\n", n);
    exit(0);
}
```

Figure C.17 Calculation of pipe capacity using nonlocking writes

The following table shows the values calculated for our four platforms.

Platform	Pipe Capacity (bytes)
FreeBSD 5.2.1	16,384
Linux 2.4.22	4,096
Mac OS X 10.3	8,192
Solaris 9	9,216

These values can differ from the corresponding `PIPE_BUF` values, because `PIPE_BUF` is defined to be the maximum amount of data that can be written to a pipe *atomically*. Here, we calculate the amount of data that a pipe can hold independent of any atomicity constraints.

14.10 Whether the program in Figure 14.32 updates the last-access time for the input file depends on the operating system and the type of file system on which the file resides.