21

Communicating with a Network Printer

21.1 Introduction

We now develop a program that can communicate with a network printer. These printers are connected to multiple computers via Ethernet and often support PostScript files as well as plain text files. Applications generally use the Internet Printing Protocol (IPP) to communicate with these printers, although some support alternative communication protocols.

We are about to describe two programs: a print spooler daemon that sends jobs to a printer and a command to submit print jobs to the spooler daemon. Since the print spooler has to do multiple things (e.g., communicate with clients submitting jobs, communicate with the printer, read files, scan directories), this gives us a chance to use many of the functions from earlier chapters. For example, we use threads (Chapters 11 and 12) to simplify the design of the print spooler and sockets (Chapter 16) to communicate between the program used to schedule a file to be printed and the print spooler, and also between the print spooler and the network printer.

21.2 The Internet Printing Protocol

IPP specifies the communication rules for building network-based printing systems. By embedding an IPP server inside a printer with an Ethernet card, the printer can service requests from many computer systems. These computer systems need not be located on the same physical network, however. IPP is built on top of standard Internet protocols, so any computer that can create a TCP/IP connection to the printer can submit a print job.

IPP is specified in a series of documents (Requests For Comments, or RFCs) available at `http://www.ietf.org/rfc.html`. Proposed draft standards are developed by the Printer Working Group, which is associated with the IEEE. These drafts are available at `http://www.pwg.org/ipp`. The main documents are listed in Figure 21.1, although many other documents are available to further specify administrative procedures, job attributes, and the like.

Document	Title
RFC 2567	Design Goals for an Internet Printing Protocol
RFC 2568	Rationale for the Structure of the Model and Protocol for the Internet Printing Protocol
RFC 2911	Internet Printing Protocol/1.1: Model and Semantics
RFC 2910	Internet Printing Protocol/1.1: Encoding and Transport
RFC 3196	Internet Printing Protocol/1.1: Implementor's Guide
Candidate Standard 5100.12-2011	Internet Printing Protocol Version 2.0, Second Edition

Figure 21.1 Primary IPP documents

Candidate Standard 5100.12-2011 specifies all features that implementations must support to conform to different versions of the IPP standard. There are many proposed extensions to the IPP protocol (specific features are defined in other IPP-related documents). These features are divided into groups to create different conformance levels; each level is a different protocol version. For compatibility, each higher level of conformance requires that implementations meet most of the requirements defined by lower versions of the standard. In this chapter, we will use IPP version 1.1 in our simple example.

IPP is built on top of HTTP, the Hypertext Transfer Protocol (Section 21.3). HTTP, in turn, is built on top of TCP/IP. Figure 21.2 shows the structure of an IPP message.

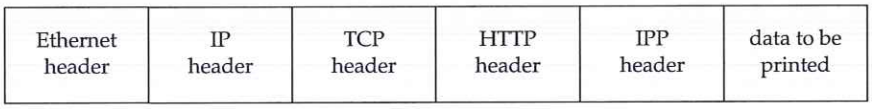

Figure 21.2 Structure of an IPP message

IPP is a request–response protocol. A client sends a request message to a server, and the server answers with a response message. The IPP header contains a field that indicates the requested operation. Operations are defined to submit print jobs, cancel print jobs, get job attributes, get printer attributes, pause and restart the printer, place a job on hold, and release a held job.

Figure 21.3 shows the structure of an IPP message header. The first 2 bytes are the IPP version number. For protocol version 1.1, each byte has a value of 1. For a protocol request, the next 2 bytes contain a value identifying the type of operation requested. For a protocol response, these 2 bytes contain a status code instead.

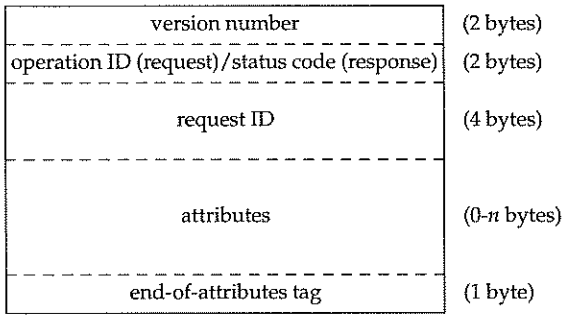

Figure 21.3 Structure of an IPP header

The next 4 bytes contain an integer identifying the request, which allows requests to be matched up with responses. Optional attributes follow this, terminated by an end-of-attributes tag. Any data that might be associated with the request follows immediately after the end-of-attributes tag.

In the header, integers are stored as signed, two's-complement, binary values in big-endian byte order (i.e., network byte order). Attributes are stored in groups. Each group starts with a single byte identifying the group. Within each group, an attribute is generally represented as a 1-byte tag, followed by a 2-byte name length, followed by the name of the attribute, followed by a 2-byte value length, and finally by the value itself. The value can be encoded as a string, a binary integer, or a more complex structure, such as a date/timestamp.

Figure 21.4 shows how the `attributes-charset` attribute would be encoded with a value of `utf-8`.

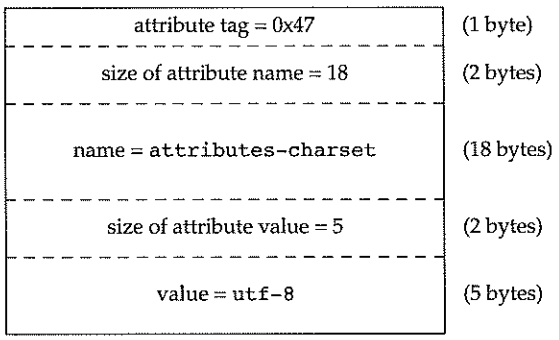

Figure 21.4 Sample IPP attribute encoding

Depending on the operation requested, some attributes are required to be provided in the request message, whereas others are optional. For example, Figure 21.5 shows the attributes defined for a print-job request.

Attribute	Status	Description
attributes-charset	required	the character set used by attributes of type text or name
attributes-natural-language	required	the natural language used by attributes of type text or name
printer-uri	required	the printer's Universal Resource Identifier
requesting-user-name	optional	name of user submitting job (used for authentication, if enabled)
job-name	optional	name of job used to distinguish between multiple jobs
ipp-attribute-fidelity	optional	if true, tells printer to reject job if all attributes can't be met; otherwise, printer does its best to print the job
document-name	optional	the name of the document (suitable for printing in a banner, for example)
document-format	optional	the format of the document (e.g., plaintext, PostScript)
document-natural-language	optional	the natural language of the document
compression	optional	the algorithm used to compress the document data
job-k-octets	optional	size of the document in 1,024-octet units
job-impressions	optional	number of impressions (images imposed on a page) submitted in this job
job-media-sheets	optional	number of sheets printed by this job

Figure 21.5 Attributes of print-job request

The IPP header contains a mixture of text and binary data. Attribute names are stored as text, but sizes are stored as binary integers. This complicates the process of building and parsing the header, since we need to worry about such things as network byte order and our host processor's ability to address an integer on an arbitrary byte boundary. A better alternative would have been to design the header to contain text only. This simplifies processing at the cost of slightly larger protocol messages.

21.3 The Hypertext Transfer Protocol

Version 1.1 of HTTP is specified in RFC 2616. HTTP is also a request–response protocol. A request message contains a start line, followed by header lines, a blank line, and an optional entity body. The entity body contains the IPP header and data in this case.

HTTP headers are ASCII, with each line terminated by a carriage return (`\r`) and a line feed (`\n`). The start line consists of a *method* that indicates which operation the client is requesting, a Uniform Resource Locator (URL) that describes the server and protocol, and a string indicating the HTTP version. The only method used by IPP is `POST`, which is used to send data to a server.

The header lines specify attributes, such as the format and length of the entity body. A header line consists of an attribute name followed by a colon, optional white space, and the attribute value, and is terminated by a carriage return and a line feed. For example, to specify that the entity body contains an IPP message, we include the header line

```
Content-Type: application/ipp
```

The following is a sample HTTP header for a print request submitted to the author's Xerox Phaser 8560 printer:

```
POST /ipp HTTP/1.1^M
Content-Length: 21931^M
Content-Type: application/ipp^M
Host: phaser8560:631^M
^M
```

The `Content-Length` line specifies the size in bytes of the amount of data in the HTTP message. This excludes the size of the HTTP header, but includes the size of the IPP header. The `Host` line specifies the host name and port number of the server to which the message is being sent.

The `^M` at the end of the each line is the carriage return that precedes the line feed. The line feed doesn't show up as a printable character. Note that the last line of the header is empty, except for the carriage return and line feed.

The start line in an HTTP response message contains a version string followed by a numeric status code and a status message, terminated by a carriage return and a line feed. The remainder of the HTTP response message has the same format as the request message: headers followed by a blank line and an optional entity body.

In response to a print request, the printer might send us the following message:

```
HTTP/1.1 200 OK^M
Content-Type: application/ipp^M
Cache-Control: no-cache, no-store, must-revalidate^M
Expires: THU, 26 OCT 1995 00:00:00 GMT^M
Content-Length: 215^M
Server: Allegro-Software-RomPager/4.34^M
^M
```

As far as our print spooler is concerned, all we care about in this message is the first line: it tells us whether the request succeeded or failed using a numeric error code and a short string. The remainder of the message contains additional information to control caching by nodes that might sit in between the client and the server and to indicate the software version running on the server.

21.4 Printer Spooling

The programs that we develop in this chapter form the basis of a simple printer spooler. A simple user command sends a file to the printer spooler; the spooler saves it to disk, queues the request, and ultimately sends the file to the printer.

All UNIX Systems provide at least one print spooling system. FreeBSD ships LPD, the BSD print spooling system (see `lpd`(8) and Chapter 13 of Stevens [1990]). Linux and Mac OS X include CUPS, the Common UNIX Printing System (see `cupsd`(8)). Solaris ships with the standard System V printer spooler (see `lp`(1) and `lpsched`(1M)). In this

chapter, our interest is not in these spooling systems per se, but rather in communicating with a network printer. We need to develop a spooling system to solve the problem of multiuser access to a single resource (the printer).

We use a simple command that reads a file and sends it to the printer spooler daemon. The command has one option to force the file to be treated as plaintext (the default assumes that the file is PostScript). We call this command `print`.

In our printer spooler daemon, `printd`, we use multiple threads to divide up the work that the daemon needs to accomplish.

- One thread listens on a socket for new print requests arriving from clients running the `print` command.
- A separate thread is spawned for each client to copy the file to be printed to a spooling area.
- One thread communicates with the printer, sending it queued jobs one at a time.
- One thread handles signals.

Figure 21.6 shows how these components fit together.

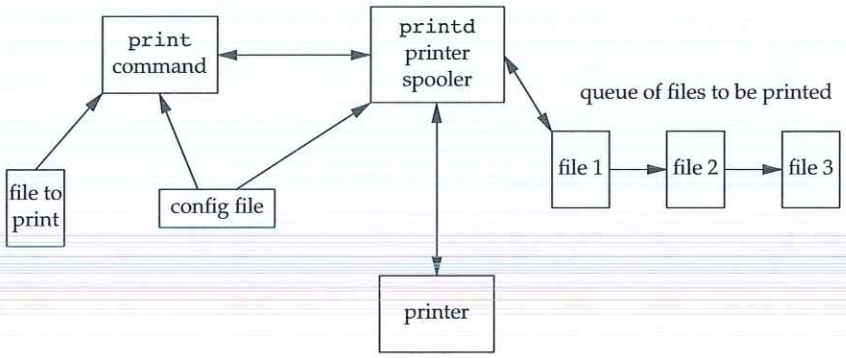

Figure 21.6 Printer spooling components

The print configuration file is `/etc/printer.conf`. It identifies the host name of the server running the printer spooling daemon and the host name of the network printer. The spooling daemon is identified by a line starting with the `printserver` keyword, followed by white space and the host name of the server. The printer is identified by a line starting with the `printer` keyword, followed by white space and the host name of the printer.

A sample printer configuration file might contain the following lines:

```
printserver   fujin
printer       phaser8560
```

where `fujin` is the host name of the computer system running the printer spooling daemon, and `phaser8560` is the host name of the network printer. We assume these

names are listed in `/etc/hosts` or registered with whatever name service we are using, so that we can translate the names to network addresses.

We can run the `print` command on the same machine where the printer spooling daemon is running, or we can run it from any machine on the same network. We only need to configure the `printserver` field in `/etc/printer.conf` in the latter case, because only the daemon needs to know the name of the printer.

Security

Programs that run with superuser privileges have the potential to open a computer system up to attack. Such programs usually aren't more vulnerable than any other program, but when compromised can lead to attackers obtaining full access to your system.

The printer spooling daemon in this chapter starts out with superuser privileges in this example to be able to bind a socket to a privileged TCP port number. To make the daemon less vulnerable to attack, we can

- Design the daemon to conform to the principles of least privilege (Section 8.11). After we obtain a socket bound to a privileged port address, we can change the user and group IDs of the daemon to something other than `root` (`lp`, for example). All the files and directories used to store queued print jobs should be owned by this nonprivileged user. This way, the daemon, if compromised, will provide the attacker with access only to the printing subsystem. This is still a concern, but it is far less serious than an attacker getting full access to your system.
- Audit the daemon's source code for all known potential vulnerabilities, such as buffer overruns.
- Log unexpected or suspicious behavior so that an administrator can take note and investigate further.

21.5 Source Code

The source code for this chapter comprises five files, not including some of the common library routines we've used in earlier chapters:

`ipp.h`	Header file containing IPP definitions
`print.h`	Header containing common constants, data structure definitions, and utility routine declarations
`util.c`	Utility routines used by the two programs
`print.c`	The C source file for the command used to print a file
`printd.c`	The C source file for the printer spooling daemon

We will study each file in the order listed.

We start with the `ipp.h` header file.

```
1  #ifndef _IPP_H
2  #define _IPP_H
3  /*
4   * Defines parts of the IPP protocol between the scheduler
5   * and the printer.  Based on RFC2911 and RFC2910.
6   */
7  /*
8   * Status code classes.
9   */
10 #define STATCLASS_OK(x)      ((x) >= 0x0000 && (x) <= 0x00ff)
11 #define STATCLASS_INFO(x)    ((x) >= 0x0100 && (x) <= 0x01ff)
12 #define STATCLASS_REDIR(x)   ((x) >= 0x0300 && (x) <= 0x03ff)
13 #define STATCLASS_CLIERR(x)  ((x) >= 0x0400 && (x) <= 0x04ff)
14 #define STATCLASS_SRVERR(x)  ((x) >= 0x0500 && (x) <= 0x05ff)
15 /*
16  * Status codes.
17  */
18 #define STAT_OK              0x0000  /* success */
19 #define STAT_OK_ATTRIGN      0x0001  /* OK; some attrs ignored */
20 #define STAT_OK_ATTRCON      0x0002  /* OK; some attrs conflicted */
21 #define STAT_CLI_BADREQ      0x0400  /* invalid client request */
22 #define STAT_CLI_FORBID      0x0401  /* request is forbidden */
23 #define STAT_CLI_NOAUTH      0x0402  /* authentication required */
24 #define STAT_CLI_NOPERM      0x0403  /* client not authorized */
25 #define STAT_CLI_NOTPOS      0x0404  /* request not possible */
26 #define STAT_CLI_TIMOUT      0x0405  /* client too slow */
27 #define STAT_CLI_NOTFND      0x0406  /* no object found for URI */
28 #define STAT_CLI_OBJGONE     0x0407  /* object no longer available */
29 #define STAT_CLI_TOOBIG      0x0408  /* requested entity too big */
30 #define STAT_CLI_TOOLNG      0x0409  /* attribute value too large */
31 #define STAT_CLI_BADFMT      0x040a  /* unsupported doc format */
32 #define STAT_CLI_NOTSUP      0x040b  /* attributes not supported */
33 #define STAT_CLI_NOSCHM      0x040c  /* URI scheme not supported */
34 #define STAT_CLI_NOCHAR      0x040d  /* charset not supported */
35 #define STAT_CLI_ATTRCON     0x040e  /* attributes conflicted */
36 #define STAT_CLI_NOCOMP      0x040f  /* compression not supported */
37 #define STAT_CLI_COMPERR     0x0410  /* data can't be decompressed */
38 #define STAT_CLI_FMTERR      0x0411  /* document format error */
39 #define STAT_CLI_ACCERR      0x0412  /* error accessing data */
```

[1–14] We start the `ipp.h` header with the standard `#ifdef` to prevent errors when it is included twice in the same file. Then we define the classes of IPP status codes (see Section 13 in RFC 2911).

[15–39] We define specific status codes based on RFC 2911. We don't use these codes in the program shown here; their use is left as an exercise (See Exercise 21.1).

```
40  #define STAT_SRV_INTERN    0x0500  /* unexpected internal error */
41  #define STAT_SRV_NOTSUP    0x0501  /* operation not supported */
42  #define STAT_SRV_UNAVAIL   0x0502  /* service unavailable */
43  #define STAT_SRV_BADVER    0x0503  /* version not supported */
44  #define STAT_SRV_DEVERR    0x0504  /* device error */
45  #define STAT_SRV_TMPERR    0x0505  /* temporary error */
46  #define STAT_SRV_REJECT    0x0506  /* server not accepting jobs */
47  #define STAT_SRV_TOOBUSY   0x0507  /* server too busy */
48  #define STAT_SRV_CANCEL    0x0508  /* job has been canceled */
49  #define STAT_SRV_NOMULTI   0x0509  /* multi-doc jobs unsupported */

50  /*
51   * Operation IDs
52   */
53  #define OP_PRINT_JOB          0x02
54  #define OP_PRINT_URI          0x03
55  #define OP_VALIDATE_JOB       0x04
56  #define OP_CREATE_JOB         0x05
57  #define OP_SEND_DOC           0x06
58  #define OP_SEND_URI           0x07
59  #define OP_CANCEL_JOB         0x08
60  #define OP_GET_JOB_ATTR       0x09
61  #define OP_GET_JOBS           0x0a
62  #define OP_GET_PRINTER_ATTR   0x0b
63  #define OP_HOLD_JOB           0x0c
64  #define OP_RELEASE_JOB        0x0d
65  #define OP_RESTART_JOB        0x0e
66  #define OP_PAUSE_PRINTER      0x10
67  #define OP_RESUME_PRINTER     0x11
68  #define OP_PURGE_JOBS         0x12

69  /*
70   * Attribute Tags.
71   */
72  #define TAG_OPERATION_ATTR    0x01  /* operation attributes tag */
73  #define TAG_JOB_ATTR          0x02  /* job attributes tag */
74  #define TAG_END_OF_ATTR       0x03  /* end of attributes tag */
75  #define TAG_PRINTER_ATTR      0x04  /* printer attributes tag */
76  #define TAG_UNSUPP_ATTR       0x05  /* unsupported attributes tag */
```

[40–49] We continue to define status codes. The ones in the range `0x500` to `0x5ff` are server error codes. All codes are described in Sections 13.1.1 through 13.1.5 in RFC 2911.

[50–68] We define the various operation IDs next. There is one ID for each task defined by IPP (see Section 4.4.15 in RFC 2911). In our example, we will use only the print-job operation.

[69–76] The attribute tags delimit the attribute groups in the IPP request and response messages. The tag values are defined in Section 3.5.1 of RFC 2910.

```
 77  /*
 78   * Value Tags.
 79   */
 80  #define TAG_UNSUPPORTED       0x10 /* unsupported value */
 81  #define TAG_UNKNOWN           0x12 /* unknown value */
 82  #define TAG_NONE              0x13 /* no value */
 83  #define TAG_INTEGER           0x21 /* integer */
 84  #define TAG_BOOLEAN           0x22 /* boolean */
 85  #define TAG_ENUM              0x23 /* enumeration */
 86  #define TAG_OCTSTR            0x30 /* octetString */
 87  #define TAG_DATETIME          0x31 /* dateTime */
 88  #define TAG_RESOLUTION        0x32 /* resolution */
 89  #define TAG_INTRANGE          0x33 /* rangeOfInteger */
 90  #define TAG_TEXTWLANG         0x35 /* textWithLanguage */
 91  #define TAG_NAMEWLANG         0x36 /* nameWithLanguage */
 92  #define TAG_TEXTWOLANG        0x41 /* textWithoutLanguage */
 93  #define TAG_NAMEWOLANG        0x42 /* nameWithoutLanguage */
 94  #define TAG_KEYWORD           0x44 /* keyword */
 95  #define TAG_URI               0x45 /* URI */
 96  #define TAG_URISCHEME         0x46 /* uriScheme */
 97  #define TAG_CHARSET           0x47 /* charset */
 98  #define TAG_NATULANG          0x48 /* naturalLanguage */
 99  #define TAG_MIMETYPE          0x49 /* mimeMediaType */

100  struct ipp_hdr {
101    int8_t  major_version;  /* always 1 */
102    int8_t  minor_version;  /* always 1 */
103    union {
104        int16_t op; /* operation ID */
105        int16_t st; /* status */
106    } u;
107    int32_t request_id;     /* request ID */
108    char    attr_group[1];  /* start of optional attributes group */
109    /* optional data follows */
110  };

111  #define operation u.op
112  #define status u.st

113  #endif /* _IPP_H */
```

[77–99] The value tags indicate the format of individual attributes and parameters. They are defined in Section 3.5.2 of RFC 2910.

[100–113] We define the structure of an IPP header. Request messages start with the same header as response messages, except that the operation ID in the request is replaced by a status code in the response.

We end the header file with a `#endif` to match the `#ifdef` at the start of the file.

The next file is the `print.h` header.

```
 1  #ifndef _PRINT_H
 2  #define _PRINT_H

 3  /*
 4   * Print server header file.
 5   */
 6  #include <sys/socket.h>
 7  #include <arpa/inet.h>
 8  #include <netdb.h>
 9  #include <errno.h>

10  #define CONFIG_FILE     "/etc/printer.conf"
11  #define SPOOLDIR        "/var/spool/printer"
12  #define JOBFILE         "jobno"
13  #define DATADIR         "data"
14  #define REQDIR          "reqs"

15  #if defined(BSD)
16  #define LPNAME          "daemon"
17  #elif defined(MACOS)
18  #define LPNAME          "_lp"
19  #else
20  #define LPNAME          "lp"
21  #endif
```

[1–9] We include all header files that an application might need if it included this header. This makes it easy for applications to include `print.h` without having to track down all the header dependencies.

[10–14] We define the files and directories for the implementation. The configuration file containing the host names of the printer spooling daemon and the network-attached printer is `/etc/printer.conf`. Copies of the files to be printed will be stored in the directory `/var/spool/printer/data` and control information for each request will be stored in the directory `/var/spool/printer/reqs`. The file containing the next job number is `/var/spool/printer/jobno`.

The directories must be created by an administrator and be owned by the same user account under which the printer spooling daemon runs. The daemon won't try to create these directories if they don't exist, because the daemon would need root privileges to create directories in `/var/spool`. We design the daemon to do as little as possible while running as root to minimize the chance of creating a security hole.

[15–21] Next, we define the account name under which the printer spooling daemon will run. On Linux and Solaris, this name is `lp`. On Mac OS X, the name is `_lp`. FreeBSD, however, doesn't define a separate account for the printer spooling daemon, so we use the account reserved for system daemons.

```
22  #define FILENMSZ        64
23  #define FILEPERM        (S_IRUSR|S_IWUSR)

24  #define USERNM_MAX      64
25  #define JOBNM_MAX       256
26  #define MSGLEN_MAX      512

27  #ifndef HOST_NAME_MAX
28  #define HOST_NAME_MAX   256
29  #endif

30  #define IPP_PORT        631
31  #define QLEN            10

32  #define IBUFSZ          512     /* IPP header buffer size */
33  #define HBUFSZ          512     /* HTTP header buffer size */
34  #define IOBUFSZ         8192    /* data buffer size */

35  #ifndef ETIME
36  #define ETIME ETIMEDOUT
37  #endif

38  extern int getaddrlist(const char *, const char *,
39    struct addrinfo **);
40  extern char *get_printserver(void);
41  extern struct addrinfo *get_printaddr(void);
42  extern ssize_t tread(int, void *, size_t, unsigned int);
43  extern ssize_t treadn(int, void *, size_t, unsigned int);
44  extern int connect_retry(int, int, int, const struct sockaddr *,
45    socklen_t);
46  extern int initserver(int, const struct sockaddr *, socklen_t,
47    int);
```

[22–34] Next, we define limits and constants. `FILEPERM` is the permissions used when creating copies of files submitted to be printed. The permissions are restrictive because we don't want ordinary users to be able to read each other's files while they are waiting to be printed. We define `HOST_NAME_MAX` as the largest host name we will support if we are unable to determine the system's limit with `sysconf`.

IPP is defined to use port 631. The `QLEN` is the backlog parameter we pass to `listen` (see Section 16.4 for details).

[35–37] Some platforms don't define the error `ETIME`, so we define it to an alternate error code that makes sense for these systems. This is the error code we return when a read times out (we don't want the server to block indefinitely reading from a socket).

[38–47] Next, we declare all the public routines contained in `util.c` (we'll look at these shortly). Note that the `connect_retry` function, from Figure 16.11, and the `initserver` function, from Figure 16.22, are not included in `util.c`.

```
48  /*
49   * Structure describing a print request.
50   */
51  struct printreq {
52      uint32_t size;              /* size in bytes */
53      uint32_t flags;             /* see below */
54      char usernm[USERNM_MAX];    /* user's name */
55      char jobnm[JOBNM_MAX];      /* job's name */
56  };

57  /*
58   * Request flags.
59   */
60  #define PR_TEXT         0x01    /* treat file as plain text */

61  /*
62   * The response from the spooling daemon to the print command.
63   */
64  struct printresp {
65      uint32_t retcode;           /* 0=success, !0=error code */
66      uint32_t jobid;             /* job ID */
67      char msg[MSGLEN_MAX];       /* error message */
68  };

69  #endif /* _PRINT_H */
```

[48–69] The `printreq` and `printresp` structures define the protocol between the print command and the printer spooling daemon. The `print` command sends a `printreq` structure specifying the size of the job in bytes, job characteristics, the user name, and the job name to the printer spooling daemon. The daemon responds with a `printresp` structure containing a return code, the job ID, and an error message if the request failed.

The `PR_TEXT` job characteristic indicates that the file being printed should be treated as plaintext (instead of PostScript). We define a bitmask of flags instead of defining a separate field for each flag. Although only one flag value is currently defined, we could extend the protocol in the future to add more characteristics. For example, we could add a flag to request double-sided printing. We have room for 31 additional flags without requiring that we change the size of the structure. Changing the size of the structure means that we might introduce a compatibility problem between the client and the server unless we upgrade both at the same time. An alternative approach is to add a version number to the messages to allow the structures to change with each version.

Note that we define all integers in the protocol structures with an explicit size. This helps avoid misaligned structure elements when a client has a different long integer size than the server.

The next file we will look at is `util.c`, the file containing utility routines.

```
1   #include "apue.h"
2   #include "print.h"
3   #include <ctype.h>
4   #include <sys/select.h>

5   #define MAXCFGLINE 512
6   #define MAXKWLEN   16
7   #define MAXFMTLEN  16

8   /*
9    * Get the address list for the given host and service and
10   * return through ailistpp.  Returns 0 on success or an error
11   * code on failure.  Note that we do not set errno if we
12   * encounter an error.
13   *
14   * LOCKING: none.
15   */
16  int
17  getaddrlist(const char *host, const char *service,
18    struct addrinfo **ailistpp)
19  {
20     int             err;
21     struct addrinfo hint;

22     hint.ai_flags = AI_CANONNAME;
23     hint.ai_family = AF_INET;
24     hint.ai_socktype = SOCK_STREAM;
25     hint.ai_protocol = 0;
26     hint.ai_addrlen = 0;
27     hint.ai_canonname = NULL;
28     hint.ai_addr = NULL;
29     hint.ai_next = NULL;
30     err = getaddrinfo(host, service, &hint, ailistpp);
31     return(err);
32  }
```

[1–7] We first define the limits needed by the functions in this file. `MAXCFGLINE` is the maximum size of a line in the printer configuration file, `MAXKWLEN` is the maximum size of a keyword in the configuration file, and `MAXFMTLEN` is the maximum size of the format string we pass to `sscanf`.

[8–32] The first function is `getaddrlist`. It is a wrapper for `getaddrinfo` (Section 16.3.3), since we always call `getaddrinfo` with the same hint structure. Note that we do not need mutex locking in this function. The `LOCKING` comment at the beginning of each function is intended only for documenting multithreaded locking. This comment lists the assumptions, if any, that are made regarding the locking, tells which locks the function might acquire or release, and tells which locks must be held to call the function.

```
33  /*
34   * Given a keyword, scan the configuration file for a match
35   * and return the string value corresponding to the keyword.
36   *
37   * LOCKING: none.
38   */
39  static char *
40  scan_configfile(char *keyword)
41  {
42      int             n, match;
43      FILE            *fp;
44      char            keybuf[MAXKWLEN], pattern[MAXFMTLEN];
45      char            line[MAXCFGLINE];
46      static char     valbuf[MAXCFGLINE];

47      if ((fp = fopen(CONFIG_FILE, "r")) == NULL)
48          log_sys("can't open %s", CONFIG_FILE);
49      sprintf(pattern, "%%%ds %%%ds", MAXKWLEN-1, MAXCFGLINE-1);
50      match = 0;
51      while (fgets(line, MAXCFGLINE, fp) != NULL) {
52          n = sscanf(line, pattern, keybuf, valbuf);
53          if (n == 2 && strcmp(keyword, keybuf) == 0) {
54              match = 1;
55              break;
56          }
57      }
58      fclose(fp);
59      if (match != 0)
60          return(valbuf);
61      else
62          return(NULL);
63  }
```

[33–46] The `scan_configfile` function searches through the printer configuration file for the specified keyword.

[47–63] We open the configuration file for reading and build the format string corresponding to the search pattern. The notation `%%%ds` builds a format specifier that limits the string size so we don't overrun the buffers used to store the strings on the stack. We read the file one line at a time and scan for two strings separated by white space; if we find them, we compare the first string with the keyword. If we find a match or we reach the end of the file, the loop ends and we close the file. If the keyword matches, we return a pointer to the buffer containing the string after the keyword; otherwise, we return `NULL`.

The string returned is stored in a static buffer (`valbuf`), which can be overwritten on successive calls. Thus, `scan_configfile` can't be called by a multithreaded application unless we take care to avoid calling it from multiple threads at the same time.

```
64  /*
65   * Return the host name running the print server or NULL on error.
66   *
67   * LOCKING: none.
68   */
69  char *
70  get_printserver(void)
71  {
72      return(scan_configfile("printserver"));
73  }
74  /*
75   * Return the address of the network printer or NULL on error.
76   *
77   * LOCKING: none.
78   */
79  struct addrinfo *
80  get_printaddr(void)
81  {
82      int             err;
83      char            *p;
84      struct addrinfo *ailist;
85      if ((p = scan_configfile("printer")) != NULL) {
86          if ((err = getaddrlist(p, "ipp", &ailist)) != 0) {
87              log_msg("no address information for %s", p);
88              return(NULL);
89          }
90          return(ailist);
91      }
92      log_msg("no printer address specified");
93      return(NULL);
94  }
```

[64–73] The `get_printserver` function is simply a wrapper function that calls `scan_configfile` to find the name of the computer system where the printer spooling daemon is running.

[74–94] We use the `get_printaddr` function to get the address of the network printer. It is similar to the previous function except that when we find the name of the printer in the configuration file, we use the name to find the corresponding network address.

Both `get_printserver` and `get_printaddr` call `scan_configfile`. If it can't open the printer configuration file, `scan_configfile` calls `log_sys` to print an error message and exit. Although `get_printserver` is meant to be called from a client command and `get_printaddr` is meant to be called from the daemon, having both call `log_sys` is OK, because we can arrange for the log functions to print to the standard error instead of to the log file by setting a global variable.

```
 95  /*
 96   * "Timed" read - timout specifies the # of seconds to wait before
 97   * giving up (5th argument to select controls how long to wait for
 98   * data to be readable).  Returns # of bytes read or -1 on error.
 99   *
100   * LOCKING: none.
101   */
102  ssize_t
103  tread(int fd, void *buf, size_t nbytes, unsigned int timout)
104  {
105    int             nfds;
106    fd_set          readfds;
107    struct timeval  tv;

108    tv.tv_sec = timout;
109    tv.tv_usec = 0;
110    FD_ZERO(&readfds);
111    FD_SET(fd, &readfds);
112    nfds = select(fd+1, &readfds, NULL, NULL, &tv);
113    if (nfds <= 0) {
114        if (nfds == 0)
115            errno = ETIME;
116        return(-1);
117    }
118    return(read(fd, buf, nbytes));
119  }
```

[95–107] We provide a function called `tread` to read a specified number of bytes, but block for at most *timout* seconds before giving up. This function is useful when reading from a socket or a pipe. If we don't receive data before the specified time limit, we return -1 with `errno` set to `ETIME`. If data is available within the time limit, we return at most *nbytes* bytes of data, but we can return less than requested if all the data doesn't arrive in time.

We use `tread` to prevent denial-of-service attacks on the printer spooling daemon. A malicious user might repeatedly try to connect to the daemon without sending it data, just to prevent other users from being able to submit print jobs. By giving up after a reasonable amount of time, we prevent this from happening. The tricky part is selecting a suitable timeout value that is large enough to prevent premature failures when the system is under load and tasks are taking longer to complete. If we choose a value that is too large, however, we might enable denial-of-service attacks by allowing the daemon to consume too many resources to process the pending requests.

[108–119] We use `select` to wait for the specified file descriptor to be readable. If the time limit expires before data is available to be read, `select` returns 0, so we set `errno` to `ETIME` in this case. If `select` fails or times out, we return –1. Otherwise, we return whatever data is available.

```
120  /*
121   * "Timed" read - timout specifies the number of seconds to wait
122   * per read call before giving up, but read exactly nbytes bytes.
123   * Returns number of bytes read or -1 on error.
124   *
125   * LOCKING: none.
126   */
127  ssize_t
128  treadn(int fd, void *buf, size_t nbytes, unsigned int timout)
129  {
130    size_t  nleft;
131    ssize_t nread;

132    nleft = nbytes;
133    while (nleft > 0) {
134        if ((nread = tread(fd, buf, nleft, timout)) < 0) {
135            if (nleft == nbytes)
136                return(-1); /* error, return -1 */
137            else
138                break;      /* error, return amount read so far */
139        } else if (nread == 0) {
140            break;          /* EOF */
141        }
142        nleft -= nread;
143        buf += nread;
144    }
145    return(nbytes - nleft);      /* return >= 0 */
146  }
```

[120–146] We also provide a variation of `tread`, called `treadn`, that reads exactly the number of bytes requested. This is similar to the `readn` function described in Section 14.7, but with the addition of the timeout parameter.

To read exactly *nbytes* bytes, we have to be prepared to make multiple calls to `read`. The difficult part is trying to apply a single timeout value to multiple calls to `read`. We don't want to use an alarm, because signals can be messy to deal with in multithreaded applications. We can't rely on the system updating the `timeval` structure on return from `select` to indicate the amount of time left, because many platforms do not support this behavior (Section 14.4.1). Thus, we compromise and define the timeout value in this case to apply to an individual `read` call. Instead of limiting the total amount of time we wait, it limits the amount of time we'll wait in every iteration of the loop. The maximum time we can wait is bounded by (*nbytes* × *timout*) seconds (worst case, we'll receive only 1 byte at a time).

We use `nleft` to record the number of bytes remaining to be read. If `tread` fails and we have received data in a previous iteration, we break out of the `while` loop and return the number of bytes read; otherwise, we return –1.

The command used to submit a print job is shown next. The C source file is `print.c`.

```
1   /*
2    * The client command for printing documents.  Opens the file
3    * and sends it to the printer spooling daemon.  Usage:
4    *     print [-t] filename
5    */
6   #include "apue.h"
7   #include "print.h"
8   #include <fcntl.h>
9   #include <pwd.h>

10  /*
11   * Needed for logging funtions.
12   */
13  int log_to_stderr = 1;

14  void submit_file(int, int, const char *, size_t, int);

15  int
16  main(int argc, char *argv[])
17  {
18      int             fd, sfd, err, text, c;
19      struct stat     sbuf;
20      char            *host;
21      struct addrinfo *ailist, *aip;

22      err = 0;
23      text = 0;
24      while ((c = getopt(argc, argv, "t")) != -1) {
25          switch (c) {
26          case 't':
27              text = 1;
28              break;

29          case '?':
30              err = 1;
31              break;
32          }
33      }
```

[1–14] We need to define an integer called `log_to_stderr` to be able to use the log functions in our library. If this integer is set to a nonzero value, error messages will be sent to the standard error stream instead of to a log file. Although we don't use any logging functions in `print.c`, we do link `util.o` with `print.o` to build the executable `print` command, and `util.c` contains functions for both user commands and daemons.

[15–33] We support one option, `-t`, to force the file to be printed as text (instead of as a PostScript program, for example). We use the `getopt` function (introduced in Section 17.6) to process the command options.

```
34      if (err || (optind != argc - 1))
35          err_quit("usage: print [-t] filename");
36      if ((fd = open(argv[optind], O_RDONLY)) < 0)
37          err_sys("print: can't open %s", argv[optind]);
38      if (fstat(fd, &sbuf) < 0)
39          err_sys("print: can't stat %s", argv[optind]);
40      if (!S_ISREG(sbuf.st_mode))
41          err_quit("print: %s must be a regular file", argv[optind]);

42      /*
43       * Get the hostname of the host acting as the print server.
44       */
45      if ((host = get_printserver()) == NULL)
46          err_quit("print: no print server defined");
47      if ((err = getaddrlist(host, "print", &ailist)) != 0)
48          err_quit("print: getaddrinfo error: %s", gai_strerror(err));

49      for (aip = ailist; aip != NULL; aip = aip->ai_next) {
50          if ((sfd = connect_retry(AF_INET, SOCK_STREAM, 0,
51            aip->ai_addr, aip->ai_addrlen)) < 0) {
52              err = errno;
```

[34–41] When `getopt` completes processing the command options, it leaves the variable `optind` set to the index of the first nonoptional argument. If this is any value other than the index of the last argument, then the wrong number of arguments was specified (we support only one nonoptional argument). Our error processing includes checks to ensure that we can open the file to be printed and that it is a regular file (as opposed to a directory or other type of file).

[42–48] We get the name of the host where the printer spooling daemon is running by calling the `get_printserver` function from `util.c`. Then we translate the host name into a network address by calling `getaddrlist` (also from `util.c`).

Note that we specify the service as "print." As part of installing the printer spooling daemon on a system, we need to make sure that `/etc/services` (or the equivalent database) has an entry for the printer service. When we select a port number for the daemon, it would be a good idea to select one that is privileged, to prevent malicious users from writing applications that pretend to be a printer spooling daemon but instead steal copies of the files we try to print. This means that the port number should be less than 1,024 (recall Section 16.3.4) and that our daemon will have to run with superuser privileges to allow it to bind to a reserved port.

[49–52] We try to connect to the daemon using one address at a time from the list returned by `getaddrinfo`. We will try to send the file to the daemon using the first address to which we can connect.

```
53          } else {
54              submit_file(fd, sfd, argv[optind], sbuf.st_size, text);
55              exit(0);
56          }
57      }
58      err_exit(err, "print: can't contact %s", host);
59  }

60  /*
61   * Send a file to the printer daemon.
62   */
63  void
64  submit_file(int fd, int sockfd, const char *fname, size_t nbytes,
65              int text)
66  {
67      int                 nr, nw, len;
68      struct passwd       *pwd;
69      struct printreq     req;
70      struct printresp    res;
71      char                buf[IOBUFSZ];

72      /*
73       * First build the header.
74       */
75      if ((pwd = getpwuid(geteuid())) == NULL) {
76          strcpy(req.usernm, "unknown");
77      } else {
78          strncpy(req.usernm, pwd->pw_name, USERNM_MAX-1);
79          req.usernm[USERNM_MAX-1] = '\0';
80      }
```

[53–59] If we are able to connect to the printer spooling daemon, we call `submit_file` to transmit the file we want to print to the daemon. Then we exit with a value of 0 to indicate success. If we can't connect to any of the addresses, we call `err_exit` to print an error message and exit with a value of 1 to indicate failure. (Appendix B contains the source code for `err_exit` and the other error routines.)

[60–80] The `submit_file` function sends a print request to the daemon and reads the response. First, we build the `printreq` request header. We use `geteuid` to get the caller's effective user ID and pass this to `getpwuid` to look for the user in the system's password file. We copy the user's name to the request header or use the string unknown if we can't identify the user. We use `strncpy` when copying the name from the password file to avoid writing past the end of the user name buffer in the request header. If the name is longer than the size of the buffer, `strncpy` won't store a terminating null byte in the buffer, so we need to do it ourselves.

```
 81      req.size = htonl(nbytes);

 82      if (text)
 83          req.flags = htonl(PR_TEXT);
 84      else
 85          req.flags = 0;

 86      if ((len = strlen(fname)) >= JOBNM_MAX) {
 87          /*
 88           * Truncate the filename (+-5 accounts for the leading
 89           * four characters and the terminating null).
 90           */
 91          strcpy(req.jobnm, "... ");
 92          strncat(req.jobnm, &fname[len-JOBNM_MAX+5], JOBNM_MAX-5);
 93      } else {
 94          strcpy(req.jobnm, fname);
 95      }

 96      /*
 97       * Send the header to the server.
 98       */
 99      nw = writen(sockfd, &req, sizeof(struct printreq));
100      if (nw != sizeof(struct printreq)) {
101          if (nw < 0)
102              err_sys("can't write to print server");
103          else
104              err_quit("short write (%d/%d) to print server",
105                nw, sizeof(struct printreq));
106      }
```

[81–95] We store the size of the file to be printed in the header after converting it to network byte order. Then we do the same with the `PR_TEXT` flag if the file is to be printed as plaintext. By translating these integers to network byte order, we can run the print command on a client system while the printer spooling daemon is running on another computer system. If these systems use processors with different byte ordering, then the commands will still work. (We discussed byte ordering in Section 16.3.1.)

We set the job name to the name of the file being printed. If the name is longer than will fit in the job name field in the message, we copy only the last portion of the name that will fit. This effectively truncates the beginning portion of the name. In this case, we prepend an ellipsis to indicate that there were more characters than would fit in the field.

[96–106] We send the request header to the daemon using `writen`. (Recall that we introduced the `writen` function in Figure 14.24.) The `writen` function uses multiple calls to `write`, if necessary, to transmit the specified amount. If the `writen` function returns an error or transmits less than we requested, we print an error message and exit.

```
107     /*
108      * Now send the file.
109      */
110     while ((nr = read(fd, buf, IOBUFSZ)) != 0) {
111         nw = writen(sockfd, buf, nr);
112         if (nw != nr) {
113             if (nw < 0)
114                 err_sys("can't write to print server");
115             else
116                 err_quit("short write (%d/%d) to print server",
117                   nw, nr);
118         }
119     }

120     /*
121      * Read the response.
122      */
123     if ((nr = readn(sockfd, &res, sizeof(struct printresp))) !=
124       sizeof(struct printresp))
125         err_sys("can't read response from server");
126     if (res.retcode != 0) {
127         printf("rejected: %s\n", res.msg);
128         exit(1);
129     } else {
130         printf("job ID %ld\n", (long)ntohl(res.jobid));
131     }
132 }
```

[107–119] After sending the header to the daemon, we send the file to be printed. We read the file `IOBUFSZ` bytes at a time and use `writen` to send the data to the daemon. As with the header, if the write fails or we write less than we expect, we print an error message and exit.

[120–132] Once we have sent the file to be printed to the print spooling daemon, we read the daemon's response. If the print request failed, the return code (`retcode`) will be nonzero, so we print the textual error message included in the response. If the request succeeded, we print the job ID so that the user knows how to refer to the request in the future. (Writing a command to cancel a pending print request is left as an exercise; the job ID can be used in the cancellation request to identify the job to be removed from the print queue. See Exercise 21.5.) When `submit_file` returns to the `main` function, we exit, indicating success.

Note that a successful response from the daemon does not mean that the printer was able to print the file; it merely means that the daemon successfully added the print job to the queue.

This completes our look at the print command. The last file we will look at is the C source file for the printer spooling daemon.

```
1   /*
2    * Print server daemon.
3    */
4   #include "apue.h"
5   #include <fcntl.h>
6   #include <dirent.h>
7   #include <ctype.h>
8   #include <pwd.h>
9   #include <pthread.h>
10  #include <strings.h>
11  #include <sys/select.h>
12  #include <sys/uio.h>

13  #include "print.h"
14  #include "ipp.h"

15  /*
16   * These are for the HTTP response from the printer.
17   */
18  #define HTTP_INFO(x)    ((x) >= 100 && (x) <= 199)
19  #define HTTP_SUCCESS(x) ((x) >= 200 && (x) <= 299)

20  /*
21   * Describes a print job.
22   */
23  struct job {
24      struct job      *next;      /* next in list */
25      struct job      *prev;      /* previous in list */
26      int32_t          jobid;     /* job ID */
27      struct printreq  req;       /* copy of print request */
28  };

29  /*
30   * Describes a thread processing a client request.
31   */
32  struct worker_thread {
33      struct worker_thread  *next;    /* next in list */
34      struct worker_thread  *prev;    /* previous in list */
35      pthread_t              tid;     /* thread ID */
36      int                    sockfd;  /* socket */
37  };
```

[1–19] The printer spooling daemon includes the IPP header file that we saw earlier, because the daemon needs to communicate with the printer using this protocol. The `HTTP_INFO` and `HTTP_SUCCESS` macros define the status of the HTTP request (recall that IPP is built on top of HTTP). Section 10 in RFC 2616 defines the HTTP status codes.

[20–37] The `job` and `worker_thread` structures are used by the spooling daemon to keep track of print jobs and threads accepting print requests, respectively.

```
38  /*
39   * Needed for logging.
40   */
41  int                     log_to_stderr = 0;

42  /*
43   * Printer-related stuff.
44   */
45  struct addrinfo         *printer;
46  char                    *printer_name;
47  pthread_mutex_t         configlock = PTHREAD_MUTEX_INITIALIZER;
48  int                     reread;

49  /*
50   * Thread-related stuff.
51   */
52  struct worker_thread    *workers;
53  pthread_mutex_t         workerlock = PTHREAD_MUTEX_INITIALIZER;
54  sigset_t                mask;

55  /*
56   * Job-related stuff.
57   */
58  struct job              *jobhead, *jobtail;
59  int                     jobfd;
```

[38–41] Our logging functions require that we define the `log_to_stderr` variable and set it to 0 to force log messages to be sent to the system log instead of to the standard error. In `print.c`, we defined `log_to_stderr` and set it to 1, even though we don't use the log functions in the user command. We could have avoided this by splitting the utility functions into two separate files: one for the server and one for the client commands.

[42–48] We use the global variable `printer` to hold the network address of the printer. We store the host name of the printer in `printer_name`. The `configlock` mutex protects access to the `reread` variable, which is used to indicate that the daemon needs to reread the configuration file, presumably because an administrator changed the printer or its network address.

[49–54] Next, we define the thread-related variables. We use `workers` as the head of a doubly linked list of threads that are receiving files from clients. This list is protected by the mutex `workerlock`. The signal mask used by the threads is held in the variable `mask`.

[55–59] For the list of pending jobs, we define `jobhead` to be the start of the list and `jobtail` to be the tail of the list. This list is also doubly linked, but we need to add jobs to the end of the list, so we must remember a pointer to the list tail. With the list of worker threads, the order doesn't matter, so we can add them to the head of the list and don't need to remember the tail pointer. `jobfd` is the file descriptor for the job file.

```
60  int32_t                 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  int32_t     get_newjobno(void);
70  void        add_job(struct printreq *, int32_t);
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      n = sysconf(_SC_HOST_NAME_MAX);
112      if (n < 0)  /* best guess */
113          n = HOST_NAME_MAX;
114      if ((host = malloc(n)) == NULL)
115          log_sys("malloc error");
116      if (gethostname(host, n) < 0)
117          log_sys("gethostname error");

118      if ((err = getaddrlist(host, "print", &ailist)) != 0) {
119          log_quit("getaddrinfo error: %s", gai_strerror(err));
120          exit(1);
121      }
```

[98–100] The daemon doesn't have any options (the only argument is the command name itself), 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.

[101–110] 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 send the `SIGHUP` signal to the daemon to tell it to reread its configuration file. We'll send the `SIGTERM` signal to the daemon to tell it to clean up and exit gracefully.

[111–117] 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 constant 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.

[118–121] Next, we try to find the network address that the daemon is supposed to use to provide the printer spooling service.

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

[122–131] 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 16.22) 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.

[132–133] 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.

[134–140] 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 and group IDs to the ones associated with the `LPNAME` account. We 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 for the daemon. If no such user account exists, or if it exists with the same user ID as the superuser, we log an error message and exit. Otherwise, we change both the real and effective IDs by calling `setgid` and `setuid`. To avoid exposing our system, we choose to provide no service at all if we can't reduce our privileges.

[141–142] 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).

```
143     err = pthread_create(&tid, NULL, printer_thread, NULL);
144     if (err == 0)
145         err = pthread_create(&tid, NULL, signal_thread, NULL);
146     if (err != 0)
147         log_exit(err, "can't create thread");
148     build_qonstart();
149     log_msg("daemon initialized");
150     for (;;) {
151         rset = rendezvous;
152         if (select(maxfd+1, &rset, NULL, NULL, NULL) < 0)
153             log_sys("select failed");
154         for (i = 0; i <= maxfd; i++) {
155             if (FD_ISSET(i, &rset)) {
156                 /*
157                  * Accept the connection and handle the request.
158                  */
159                 if ((sockfd = accept(i, NULL, NULL)) < 0)
160                     log_ret("accept failed");
161                 pthread_create(&tid, NULL, client_thread,
162                   (void *)((long)sockfd));
163             }
164         }
165     }
166     exit(1);
167 }
```

[143–149] We 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.

[150–167] 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 an error 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.

```
168  /*
169   * Initialize the job ID file.  Use a record lock to prevent
170   * more than one printer daemon from running at a time.
171   *
172   * LOCKING: none, except for record-lock on job ID file.
173   */
174  void
175  init_request(void)
176  {
177      int     n;
178      char    name[FILENMSZ];

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

183      /*
184       * Reuse the name buffer for the job counter.
185       */
186      if ((n = read(jobfd, name, FILENMSZ)) < 0)
187          log_sys("can't read job file");
188      if (n == 0)
189          nextjob = 1;
190      else
191          nextjob = atol(name);
192  }
```

[168–182] 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.)

[183–192] 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 value 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 can safely reuse the filename buffer, because `FILENMSZ` is defined to be 64 in `print.h`.

```
193  /*
194   * Initialize printer information from configuration file.
195   *
196   * LOCKING: none.
197   */
198  void
199  init_printer(void)
200  {
201    printer = get_printaddr();
202    if (printer == NULL)
203        exit(1);    /* message already logged */
204    printer_name = printer->ai_canonname;
205    if (printer_name == NULL)
206        printer_name = "printer";
207    log_msg("printer is %s", printer_name);
208  }

209  /*
210   * Update the job ID file with the next job number.
211   * Doesn't handle wrap-around of job number.
212   *
213   * LOCKING: none.
214   */
215  void
216  update_jobno(void)
217  {
218    char    buf[32];

219    if (lseek(jobfd, 0, SEEK_SET) == -1)
220        log_sys("can't seek in job file");
221    sprintf(buf, "%d", nextjob);
222    if (write(jobfd, buf, strlen(buf)) < 0)
223        log_sys("can't update job file");
224  }
```

[193–208] 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 exit. The `get_printaddr` function logs its own message when it is unable to find the printer's address. If a printer address is found, however, 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.

[209–224] The `update_jobno` function is used to write the next job number to the job file, `/var/spool/printer/jobno`. We seek to the beginning of the file, convert the integer job number into a string, and write it to the file. On error, we log a message and exit. The job number increases monotonically; handling wrap-around is left as an exercise (see Exercise 21.9).

```
225  /*
226   * Get the next job number.
227   *
228   * LOCKING: acquires and releases joblock.
229   */
230  int32_t
231  get_newjobno(void)
232  {
233    int32_t jobid;

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

241  /*
242   * Add a new job to the list of pending jobs.  Then signal
243   * the printer thread that a job is pending.
244   *
245   * LOCKING: acquires and releases joblock.
246   */
247  void
248  add_job(struct printreq *reqp, int32_t jobid)
249  {
250    struct job  *jp;

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

[225–240] 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.)

[241–253] 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 an error 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 lines 23–28 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.

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

266   /*
267    * Replace a job back on the head of the list.
268    *
269    * LOCKING: acquires and releases joblock.
270    */
271   void
272   replace_job(struct job *jp)
273   {
274     pthread_mutex_lock(&joblock);
275     jp->prev = NULL;
276     jp->next = jobhead;
277     if (jobhead == NULL)
278         jobtail = jp;
279     else
280         jobhead->prev = jp;
281     jobhead = jp;
282     pthread_mutex_unlock(&joblock);
283   }
```

[254–265] 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.

[266–283] 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.

```
284   /*
285    * Remove a job from the list of pending jobs.
286    *
287    * LOCKING: caller must hold joblock.
288    */
289   void
290   remove_job(struct job *target)
291   {
292     if (target->next != NULL)
293         target->next->prev = target->prev;
294     else
295         jobtail = target->prev;
296     if (target->prev != NULL)
297         target->prev->next = target->next;
298     else
299         jobhead = target->next;
300   }

301   /*
302    * Check the spool directory for pending jobs on start-up.
303    *
304    * LOCKING: none.
305    */
306   void
307   build_qonstart(void)
308   {
309     int             fd, err, nr;
310     int32_t         jobid;
311     DIR             *dirp;
312     struct dirent   *entp;
313     struct printreq req;
314     char            dname[FILENMSZ], fname[FILENMSZ];

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

[284–300] `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.

[301–317] 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.

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

[318–324] We read each entry in the directory, one at a time. We skip the entries for dot and dot-dot.

[325–345] 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 an error message, and unlink the file. Then we create the full pathname of the corresponding data file and unlink it, too.

[346–351] 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.

```
352  /*
353   * Accept a print job from a client.
354   *
355   * LOCKING: none.
356   */
357  void *
358  client_thread(void *arg)
359  {
360    int                 n, fd, sockfd, nr, nw, first;
361    int32_t             jobid;
362    pthread_t           tid;
363    struct printreq     req;
364    struct printresp    res;
365    char                name[FILENMSZ];
366    char                buf[IOBUFSZ];
367    tid = pthread_self();
368    pthread_cleanup_push(client_cleanup, (void *)((long)tid));
369    sockfd = (long)arg;
370    add_worker(tid, sockfd);
371    /*
372     * Read the request header.
373     */
374    if ((n = treadn(sockfd, &req, sizeof(struct printreq), 10)) !=
375      sizeof(struct printreq)) {
376        res.jobid = 0;
377        if (n < 0)
378            res.retcode = htonl(errno);
379        else
380            res.retcode = htonl(EIO);
381        strncpy(res.msg, strerror(res.retcode), MSGLEN_MAX);
382        writen(sockfd, &res, sizeof(struct printresp));
383        pthread_exit((void *)1);
384    }
```

[352–370] 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.

[371–384] 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 data 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.

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

[385–401] 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`.

[402–413] We read the file contents from the client, with the intention 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.

```
414             nw = write(fd, buf, nr);
415             if (nw != nr) {
416                 res.jobid = 0;
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.  Then write the
433          * print request information to the control
434          * file.
435          */
436         sprintf(name, "%s/%s/%d", SPOOLDIR, REQDIR, jobid);
437         fd = creat(name, FILEPERM);
438         if (fd < 0) {
439             res.jobid = 0;
440             res.retcode = htonl(errno);
441             log_msg("client_thread: can't create %s: %s", name,
442               strerror(res.retcode));
443             strncpy(res.msg, strerror(res.retcode), MSGLEN_MAX);
444             writen(sockfd, &res, sizeof(struct printresp));
445             sprintf(name, "%s/%s/%d", SPOOLDIR, DATADIR, jobid);
446             unlink(name);
447             pthread_exit((void *)1);
448         }
```

[414–430] 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.

[431–448] 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.

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

[449–465] 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.

[466–473] 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.

[474–481] 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 with 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.

```
482  /*
483   * Add a worker to the list of worker threads.
484   *
485   * LOCKING: acquires and releases workerlock.
486   */
487  void
488  add_worker(pthread_t tid, int sockfd)
489  {
490    struct worker_thread    *wtp;

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

505  /*
506   * Cancel (kill) all outstanding workers.  They take themselves
507   * off the workers list when they see the cancellation.
508   *
509   * LOCKING: acquires and releases workerlock.
510   */
511  void
512  kill_workers(void)
513  {
514    struct worker_thread    *wtp;

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

[482–504] `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.

[505–519] 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.

```
520   /*
521    * Cancellation routine for the worker thread.
522    *
523    * LOCKING: acquires and releases workerlock.
524    */
525   void
526   client_cleanup(void *arg)
527   {
528     struct worker_thread    *wtp;
529     pthread_t               tid;

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

[520–542] 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.

[543–548] 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.

```
549  /*
550   * Deal with signals.
551   *
552   * LOCKING: acquires and releases configlock.
553   */
554  void *
555  signal_thread(void *arg)
556  {
557      int     err, signo;

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

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

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

[549–562] 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.

[563–570] 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.

[571–574] If we receive `SIGTERM`, we call `kill_workers` to kill all the worker threads, log a message, and call `exit` to terminate the process.

[575–580] If we receive a signal we are not expecting, we kill the worker threads and call `log_quit` to log an error message and exit.

```
581  /*
582   * Add an option to the IPP header.
583   *
584   * LOCKING: none.
585   */
586  char *
587  add_option(char *cp, int tag, char *optname, char *optval)
588  {
589    int     n;
590    union {
591        int16_t s;
592        char c[2];
593    }       u;

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

[581–593] 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 21.4 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 `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.

[594–607] 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.

```
608  /*
609   * Single thread to communicate with the printer.
610   *
611   * LOCKING: acquires and releases joblock and configlock.
612   */
613  void *
614  printer_thread(void *arg)
615  {
616    struct job      *jp;
617    int             hlen, ilen, sockfd, fd, nr, nw, extra;
618    char            *icp, *hcp, *p;
619    struct ipp_hdr  *hp;
620    struct stat     sbuf;
621    struct iovec    iov[2];
622    char            name[FILENMSZ];
623    char            hbuf[HBUFSZ];
624    char            ibuf[IBUFSZ];
625    char            buf[IOBUFSZ];
626    char            str[64];
627    struct timespec ts = { 60, 0 };     /* 1 minute */

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 %d", jp->jobid);
639        pthread_mutex_unlock(&joblock);
640        update_jobno();
```

[608–627] 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/%d", SPOOLDIR, DATADIR, jp->jobid);
659         if ((fd = open(name, O_RDONLY)) < 0) {
660             log_msg("job %d 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 %d 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 11.6.2).

[655–671] If we can't open the data file, we log an error 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 an error message, clean up, and continue.

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

[672–677] We open a stream socket connected to the printer. If the `connect_retry` call fails, we jump down to `defer`, where we will clean up, delay, and try again later.

[678–698] 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 21.5 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 Uniform Resource Identifier (URI); we set it to `http://`*printer_name*`/ipp`.

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.

```
699         if (jp->req.flags & PR_TEXT) {
700             p = "text/plain";
701             extra = 1;
702         } else {
703             p = "application/postscript";
704             extra = 0;
705         }
706         icp = add_option(icp, TAG_MIMETYPE, "document-format", p);
707         *icp++ = TAG_END_OF_ATTR;
708         ilen = icp - ibuf;

709         /*
710          * Set up the HTTP header.
711          */
712         hcp = hbuf;
713         sprintf(hcp, "POST /ipp HTTP/1.1\r\n");
714         hcp += strlen(hcp);
715         sprintf(hcp, "Content-Length: %ld\r\n",
716           (long)sbuf.st_size + ilen + extra);
717         hcp += strlen(hcp);
718         strcpy(hcp, "Content-Type: application/ipp\r\n");
719         hcp += strlen(hcp);
720         sprintf(hcp, "Host: %s:%d\r\n", printer_name, IPP_PORT);
721         hcp += strlen(hcp);
722         *hcp++ = '\r';
723         *hcp++ = '\n';
724         hlen = hcp - hbuf;
```

[699–708] The last attribute we supply is the `document-format`. If we omit it, the printer will interpret the file using some default format. For a PostScript printer, this is probably PostScript, but some printers can autosense the format and choose between PostScript, plaintext, or PCL (Hewlett-Packard's Printer Command Language). If the `PR_TEXT` flag is set, we set the format to `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.

The `extra` integer counts any extra characters we might need to transmit to the printer. As we shall see shortly, we need to send an extra character to be able to print plain text reliably. We need to account for this extra character when we calculate the content length.

[709–724] Now that we know the IPP header size, we can set up the HTTP header. We set the `Content-Length` to the size in bytes of the IPP header plus the size of the file to be printed plus any extra characters we might need to send. The `Content-Type` is `application/ipp`. We mark the end of the HTTP header with a carriage return and a line feed. Finally, we calculate the size of the HTTP header.

```
725         /*
726          * Write the headers first.  Then send the file.
727          */
728         iov[0].iov_base = hbuf;
729         iov[0].iov_len = hlen;
730         iov[1].iov_base = ibuf;
731         iov[1].iov_len = ilen;
732         if (writev(sockfd, iov, 2) != hlen + ilen) {
733             log_ret("can't write to printer");
734             goto defer;
735         }
736         if (jp->req.flags & PR_TEXT) {
737             /*
738              * Hack: allow PostScript to be printed as plain text.
739              */
740             if (write(sockfd, "\b", 1) != 1) {
741                 log_ret("can't write to printer");
742                 goto defer;
743             }
744         }
745         while ((nr = read(fd, buf, IOBUFSZ)) > 0) {
746             if ((nw = writen(sockfd, buf, nr)) != nr) {
747                 if (nw < 0)
748                   log_ret("can't write to printer");
749                 else
750                   log_msg("short write (%d/%d) to printer", nw, nr);
751                 goto defer;
752             }
753         }
```

[725–735] 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 or we write less than we requested, we log a message and jump to `defer`, where we will clean up and delay before trying again.

[736–744] Even if we specify plaintext, the Phaser 8560 will try to autosense the document format. To prevent it from recognizing the beginning of a file we want to print as plaintext, the first character we send is a backspace. This character doesn't show up in the printout and defeats the printer's ability to autosense the file format. This allows us to print the source to a PostScript file instead of printing the image resulting from the PostScript file.

[745–753] We send the data file to the printer in `IOBUFSZ` chunks. `write` can send less than we requested when the socket buffers are full, so we use `writen` to handle this case. We don't worry about this condition when we write the headers, because they are small. However, the file to print could be large.

```
754             if (nr < 0) {
755                 log_ret("can't read %s", name);
756                 goto defer;
757             }
758             /*
759              * Read the response from the printer.
760              */
761             if (printer_status(sockfd, jp)) {
762                 unlink(name);
763                 sprintf(name, "%s/%s/%d", SPOOLDIR, REQDIR, jp->jobid);
764                 unlink(name);
765                 free(jp);
766                 jp = NULL;
767             }
768     defer:
769             close(fd);
770             if (sockfd >= 0)
771                 close(sockfd);
772             if (jp != NULL) {
773                 replace_job(jp);
774                 nanosleep(&ts, NULL);
775             }
776         }
777     }
778     /*
779      * Read data from the printer, possibly increasing the buffer.
780      * Returns offset of end of data in buffer or -1 on failure.
781      *
782      * LOCKING: none.
783      */
784     ssize_t
785     readmore(int sockfd, char **bpp, int off, int *bszp)
```

[754–757] When we reach the end of the file, `read` will return 0. However, if `read` fails, we log an error message and jump to `defer`.

[758–767] After sending the file to the printer, we call `printer_status` to read the printer's response to our request. On success, `printer_status` returns a nonzero 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.

[768–777] 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, `jp` will point to the job structure for the job we are trying to print, so 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.

[778–785] The `readmore` function is used to read part of the response message from the printer.

```
786   {
787     ssize_t nr;
788     char    *bp = *bpp;
789     int     bsz = *bszp;

790     if (off >= bsz) {
791         bsz += IOBUFSZ;
792         if ((bp = realloc(*bpp, bsz)) == NULL)
793             log_sys("readmore: can't allocate bigger read buffer");
794         *bszp = bsz;
795         *bpp = bp;
796     }
797     if ((nr = tread(sockfd, &bp[off], bsz-off, 1)) > 0)
798         return(off+nr);
799     else
800         return(-1);
801   }

802   /*
803    * Read and parse the response from the printer.  Return 1
804    * if the request was successful, and 0 otherwise.
805    *
806    * LOCKING: none.
807    */
808   int
809   printer_status(int sfd, struct job *jp)
810   {
811     int             i, success, code, len, found, bufsz, datsz;
812     int32_t         jobid;
813     ssize_t         nr;
814     char            *bp, *cp, *statcode, *reason, *contentlen;
815     struct ipp_hdr  h;

816     /*
817      * Read the HTTP header followed by the IPP response header.
818      * They can be returned in multiple read attempts.  Use the
819      * Content-Length specifier to determine how much to read.
820      */
```

[786–801] If we're at the end of the buffer, we reallocate a bigger buffer and return the new starting address and size through the `bpp` and `bszp` parameters, respectively. We read as much as the buffer will hold, starting at the end of the data already in the buffer, and return the new end-of-data offset in the buffer. If the `read` fails or the timeout expires, we return –1.

[802–820] 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.

```
821     success = 0;
822     bufsz = IOBUFSZ;
823     if ((bp = malloc(IOBUFSZ)) == NULL)
824         log_sys("printer_status: can't allocate read buffer");

825     while ((nr = tread(sfd, bp, bufsz, 5)) > 0) {
826         /*
827          * Find the status.  Response starts with "HTTP/x.y"
828          * so we can skip the first 8 characters.
829          */
830         cp = bp + 8;
831         datsz = nr;
832         while (isspace((int)*cp))
833             cp++;
834         statcode = cp;
835         while (isdigit((int)*cp))
836             cp++;
837         if (cp == statcode) { /* Bad format; log it and move on */
838             log_msg(bp);
839         } else {
840             *cp++ = '\0';
841             reason = cp;
842             while (*cp != '\r' && *cp != '\n')
843                 cp++;
844             *cp = '\0';
845             code = atoi(statcode);
846             if (HTTP_INFO(code))
847                 continue;
848             if (!HTTP_SUCCESS(code)) { /* probable error: log it */
849                 bp[datsz] = '\0';
850                 log_msg("error: %s", reason);
851                 break;
852             }
```

[821–838] 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` string 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.

[839–844] If we have found a numeric status code in the response, we convert the first nondigit character following the status code to a null byte (this character should be some form of white space). 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.

[845–852] We call the `atoi` function to convert the status code string into an integer. If this is an informational message only, we ignore it and continue the loop to read 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.

```
853             /*
854              * HTTP request was okay, but still need to check
855              * IPP status.  Search for the Content-Length.
856              */
857             i = cp - bp;
858             for (;;) {
859                 while (*cp != 'C' && *cp != 'c' && i < datsz) {
860                     cp++;
861                     i++;
862                 }
863                 if (i >= datsz) {   /* get more header */
864                     if ((nr = readmore(sfd, &bp, i, &bufsz)) < 0) {
865                         goto out;
866                     } else {
867                         cp = &bp[i];
868                         datsz += nr;
869                     }
870                 }
871                 if (strncasecmp(cp, "Content-Length:", 15) == 0) {
872                     cp += 15;
873                     while (isspace((int)*cp))
874                         cp++;
875                     contentlen = cp;
876                     while (isdigit((int)*cp))
877                         cp++;
878                     *cp++ = '\0';
879                     i = cp - bp;
880                     len = atoi(contentlen);
881                     break;
882                 } else {
883                     cp++;
884                     i++;
885                 }
886             }
```

[853–870] If the HTTP request succeeds, we need to check the IPP status. We search through the message until we find the `Content-Length` attribute. HTTP header keywords are case insensitive, so we need to check both lowercase and uppercase characters. If we run out of buffer space, we call `readmore`, which uses `realloc` to increase the buffer size. Because the buffer address might change, we need to adjust `cp` to point to the correct place in the buffer.

[871–886] We use the `strncasecmp` function to do a case-insensitive comparison. If we find the `Content-Length` attribute string, we search for its value. We convert this numeric string into an integer and break out of the `for` loop. If the comparison fails, we continue searching the buffer byte by byte. If we reach the end of the buffer without finding the `Content-Length` attribute, we read more from the printer and continue the search.

```
887             if (i >= datsz) {   /* get more header */
888                 if ((nr = readmore(sfd, &bp, i, &bufsz)) < 0) {
889                     goto out;
890                 } else {
891                     cp = &bp[i];
892                     datsz += nr;
893                 }
894             }
895             found = 0;
896             while (!found) {  /* look for end of HTTP header */
897                 while (i < datsz - 2) {
898                     if (*cp == '\n' && *(cp + 1) == '\r' &&
899                       *(cp + 2) == '\n') {
900                         found = 1;
901                         cp += 3;
902                         i += 3;
903                         break;
904                     }
905                     cp++;
906                     i++;
907                 }
908                 if (i >= datsz) {   /* get more header */
909                     if ((nr = readmore(sfd, &bp, i, &bufsz)) < 0) {
910                         goto out;
911                     } else {
912                         cp = &bp[i];
913                         datsz += nr;
914                     }
915                 }
916             }
917             if (datsz - i < len) {  /* get more header */
918                 if ((nr = readmore(sfd, &bp, i, &bufsz)) < 0) {
919                     goto out;
920                 } else {
921                     cp = &bp[i];
922                     datsz += nr;
```

[887–916] We now know the length of the message (specified by the `Content-Length` attribute). If we've exhausted the contents of the buffer, we read more from the printer. Next we search for the end of the HTTP header (a blank line). If we find it, we set the `found` flag and skip the blank line. Whenever we call `readmore`, we set `cp` to point to the same offset in the buffer that it had previously just in case the buffer address changed when it was reallocated.

[917–922] 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.

```
923                 }
924             }

925             memcpy(&h, cp, sizeof(struct ipp_hdr));
926             i = ntohs(h.status);
927             jobid = ntohl(h.request_id);

928             if (jobid != jp->jobid) {
929                 /*
930                  * Different jobs.  Ignore it.
931                  */
932                 log_msg("jobid %d status code %d", jobid, i);
933                 break;
934             }

935             if (STATCLASS_OK(i))
936                 success = 1;
937             break;
938         }
939     }

940 out:
941     free(bp);
942     if (nr < 0) {
943         log_msg("jobid %d: error reading printer response: %s",
944           jobid, strerror(errno));
945     }
946     return(success);
947 }
```

[923–927] 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`, respectively.

[928–939] 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.

[940–947] Before we return, we free the buffer we used to hold the response message. 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 8560 network-attached PostScript printer. Unfortunately, this printer doesn't disable its autosense feature when we set the document format to `text/plain`. This led us to use a hack to trick the printer so that it wouldn't autosense the document format when we wanted to treat a document as plaintext. An alternative is to print the source to a PostScript program using a utility such as `a2ps`(1), which encapsulates the PostScript program before printing.

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 many features that we described in earlier chapters used in real programs: threads, I/O multiplexing, file I/O, reading directories, 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 which 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. Use the job ID as the argument to the command to specify which job to cancel. How can you prevent one user from canceling another user's print jobs?

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

21.7 In the printer daemon, explain why we don't need to prod the printer thread when the signal handling thread catches `SIGHUP` and sets `reread` to 1.

21.8 In the `printer_status` function, we search for the length of the IPP message by looking for the `Content-Length` HTTP attribute. This technique won't work with a printer that responds using a chunked transfer encoding. Check RFC 2616 to see how a chunked message is formatted and then modify `printer_status` to support this form of response as well.

21.9 In the `update_jobno` function, when the next job number wraps around from the maximum positive value to 1 (see `get_newjobno` to see how this can happen), we can write a smaller number over a larger one. This can result in the daemon reading an incorrect number when it restarts. What is a simple solution to this problem?

This page intentionally left blank