13

Daemon Processes

530128

13.1 Introduction

Daemons are processes that live for a long time. They are often started when the system is bootstrapped and terminate only when the system is shut down. Because they don't have a controlling terminal, we say that they run in the background. UNIX systems have numerous daemons that perform day-to-day activities.

In this chapter, we look at the process structure of daemons and explore how to write a daemon. Since a daemon does not have a controlling terminal, we need to see how a daemon can report error conditions when something goes wrong.

For a discussion of the historical background of the term *daemon* as it applies to computer systems, see Raymond [1996].

13.2 Daemon Characteristics

Let's look at some common system daemons and how they relate to the concepts of process groups, controlling terminals, and sessions that we described in Chapter 9. The `ps(1)` command prints the status of various processes in the system. There are a multitude of options—consult your system's manual for all the details. We'll execute

```
    ps -axj
```

under BSD-based systems to see the information we need for this discussion. The `-a` option shows the status of processes owned by others, and `-x` shows processes that don't have a controlling terminal. The `-j` option displays the job-related information: the session ID, process group ID, controlling terminal, and terminal process group ID.

Under System V–based systems, a similar command is `ps -efj`. (In an attempt to improve security, some UNIX systems don't allow us to use `ps` to look at any processes other than our own.) The output from `ps` looks like

```
UID       PID  PPID  PGID    SID  TTY  CMD
root        1     0     1      1  ?    /sbin/init
root        2     0     0      0  ?    [kthreadd]
root        3     2     0      0  ?    [ksoftirqd/0]
root        6     2     0      0  ?    [migration/0]
root        7     2     0      0  ?    [watchdog/0]
root       21     2     0      0  ?    [cpuset]
root       22     2     0      0  ?    [khelper]
root       26     2     0      0  ?    [sync_supers]
root       27     2     0      0  ?    [bdi-default]
root       29     2     0      0  ?    [kblockd]
root       35     2     0      0  ?    [kswapd0]
root       49     2     0      0  ?    [scsi_eh_0]
root      256     2     0      0  ?    [jbd2/sda5-8]
root      257     2     0      0  ?    [ext4-dio-unwrit]
syslog    847     1   843    843  ?    rsyslogd -c5
root      906     1   906    906  ?    /usr/sbin/cupsd -F
root     1037     1  1037   1037  ?    /usr/sbin/inetd
root     1067     1  1067   1067  ?    cron
daemon   1068     1  1068   1068  ?    atd
root     8196     1  8196   8196  ?    /usr/sbin/sshd -D
root    13047     2     0      0  ?    [kworker/1:0]
root    14596     2     0      0  ?    [flush-8:0]
root    26464     1 26464  26464  ?    rpcbind -w
statd   28490     1 28490  28490  ?    rpc.statd -L
root    28553     2     0      0  ?    [rpciod]
root    28554     2     0      0  ?    [nfsiod]
root    28561     1 28561  28561  ?    rpc.idmapd
root    28761     2     0      0  ?    [lockd]
root    28764     2     0      0  ?    [nfsd]
root    28775     1 28775  28775  ?    /usr/sbin/rpc.mountd --manage-gids
```

We have removed a few columns that don't interest us, such as the accumulated CPU time. The column headings, in order, are the user ID, process ID, parent process ID, process group ID, session ID, terminal name, and command string.

The system that this `ps` command was run on (Linux 3.2.0) supports the notion of a session ID, which we mentioned with the `setsid` function in Section 9.5. The session ID is simply the process ID of the session leader. Some BSD-based systems, such as Mac OS X 10.6.8, will print the address of the `session` structure corresponding to the process group that the process belongs to (Section 9.11) instead of the session ID.

The system processes you see will depend on the operating system implementation. Anything with a parent process ID of 0 is usually a kernel process started as part of the system bootstrap procedure. (An exception is `init`, which is a user-level command started by the kernel at boot time.) Kernel processes are special and generally exist for the entire lifetime of the system. They run with superuser privileges and have no controlling terminal and no command line.

In the sample ps output, kernel daemons appear with their names in square brackets. This version of Linux uses a special kernel process, kthreadd, to create other kernel processes, so kthreadd appears as the parent of the other kernel daemons. Each kernel component that needs to perform work in a process context, but that isn't invoked from the context of a user-level process, will usually have its own kernel daemon. For example, on Linux

- The kswapd daemon is also known as the pageout daemon. It supports the virtual memory subsystem by writing dirty pages to disk slowly over time, so the pages can be reclaimed.
- The flush daemon flushes dirty pages to disk when available memory reaches a configured minimum threshold. It also flushes dirty pages back to disk at regular intervals to decrease data loss in the event of a system failure. Several flush daemons can exist—one for each backing device. The sample output shows one flush daemon named flush-8:0. In the name, the backing device is identified by its major device number (8) and its minor device number (0).
- The sync_supers daemon periodically flushes file system metadata to disk.
- The jbd daemon helps implement the journal in the ext4 file system.

Process 1 is usually init (launchd on Mac OS X), as we described in Section 8.2. It is a system daemon responsible for, among other things, starting system services specific to various run levels. These services are usually implemented with the help of their own daemons.

The rpcbind daemon provides the service of mapping RPC (Remote Procedure Call) program numbers to network port numbers. The rsyslogd daemon is available to any program to log system messages for an administrator. The messages may be printed on a console device and also written to a file. (We describe the syslog facility in Section 13.4.)

We talked about the inetd daemon in Section 9.3. It listens on the system's network interfaces for incoming requests for various network servers. The nfsd, nfsiod, lockd, rpciod, rpc.idmapd, rpc.statd, and rpc.mountd daemons provide support for the Network File System (NFS). Note that the first four are kernel daemons, while the last three are user-level daemons.

The cron daemon executes commands at regularly scheduled dates and times. Numerous system administration tasks are handled by cron running programs at regularly intervals. The atd daemon is similar to cron; it allows users to execute jobs at specified times, but it executes each job once only, instead of repeatedly at regularly scheduled times. The cupsd daemon is a print spooler; it handles print requests on the system. The sshd daemon provides secure remote login and execution facilities.

Note that most of the daemons run with superuser (root) privileges. None of the daemons has a controlling terminal: the terminal name is set to a question mark. The kernel daemons are started without a controlling terminal. The lack of a controlling terminal in the user-level daemons is probably the result of the daemons having called setsid. Most of the user-level daemons are process group leaders and session leaders, and are the only processes in their process group and session. (The one exception is rsyslogd.) Finally, note that the parent of the user-level daemons is the init process.

13.3 Coding Rules

Some basic rules to coding a daemon prevent unwanted interactions from happening. We state these rules here and then show a function, daemonize, that implements them.

1. Call umask to set the file mode creation mask to a known value, usually 0. The inherited file mode creation mask could be set to deny certain permissions. If the daemon process creates files, it may want to set specific permissions. For example, if it creates files with group-read and group-write enabled, a file mode creation mask that turns off either of these permissions would undo its efforts. On the other hand, if the daemon calls library functions that result in files being created, then it might make sense to set the file mode create mask to a more restrictive value (such as 007), since the library functions might not allow the caller to specify the permissions through an explicit argument.
2. Call fork and have the parent exit. This does several things. First, if the daemon was started as a simple shell command, having the parent terminate makes the shell think that the command is done. Second, the child inherits the process group ID of the parent but gets a new process ID, so we're guaranteed that the child is not a process group leader. This is a prerequisite for the call to setsid that is done next.
3. Call setsid to create a new session. The three steps listed in Section 9.5 occur. The process (a) becomes the leader of a new session, (b) becomes the leader of a new process group, and (c) is disassociated from its controlling terminal.

   Under System V–based systems, some people recommend calling fork again at this point, terminating the parent, and continuing the daemon in the child. This guarantees that the daemon is not a session leader, which prevents it from acquiring a controlling terminal under the System V rules (Section 9.6). Alternatively, to avoid acquiring a controlling terminal, be sure to specify O_NOCTTY whenever opening a terminal device.

4. Change the current working directory to the root directory. The current working directory inherited from the parent could be on a mounted file system. Since daemons normally exist until the system is rebooted, if the daemon stays on a mounted file system, that file system cannot be unmounted.

   Alternatively, some daemons might change the current working directory to a specific location where they will do all their work. For example, a line printer spooling daemon might change its working directory to its spool directory.
5. Unneeded file descriptors should be closed. This prevents the daemon from holding open any descriptors that it may have inherited from its parent (which could be a shell or some other process). We can use our open_max function (Figure 2.17) or the getrlimit function (Section 7.11) to determine the highest descriptor and close all descriptors up to that value.
6. Some daemons open file descriptors 0, 1, and 2 to /dev/null so that any library routines that try to read from standard input or write to standard output or standard error will have no effect. Since the daemon is not associated with a

terminal device, there is nowhere for output to be displayed, nor is there anywhere to receive input from an interactive user. Even if the daemon was started from an interactive session, the daemon runs in the background, and the login session can terminate without affecting the daemon. If other users log in on the same terminal device, we wouldn't want output from the daemon showing up on the terminal, and the users wouldn't expect their input to be read by the daemon.

Example

Figure 13.1 shows a function that can be called from a program that wants to initialize itself as a daemon.

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

void
daemonize(const char *cmd)
{
    int                 i, fd0, fd1, fd2;
    pid_t               pid;
    struct rlimit       rl;
    struct sigaction    sa;

    /*
     * Clear file creation mask.
     */
    umask(0);

    /*
     * Get maximum number of file descriptors.
     */
    if (getrlimit(RLIMIT_NOFILE, &rl) < 0)
        err_quit("%s: can't get file limit", cmd);

    /*
     * Become a session leader to lose controlling TTY.
     */
    if ((pid = fork()) < 0)
        err_quit("%s: can't fork", cmd);
    else if (pid != 0) /* parent */
        exit(0);
    setsid();

    /*
     * Ensure future opens won't allocate controlling TTYs.
     */
    sa.sa_handler = SIG_IGN;
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = 0;
    if (sigaction(SIGHUP, &sa, NULL) < 0)
        err_quit("%s: can't ignore SIGHUP", cmd);
    if ((pid = fork()) < 0)
        err_quit("%s: can't fork", cmd);
    else if (pid != 0) /* parent */
        exit(0);

    /*
     * Change the current working directory to the root so
     * we won't prevent file systems from being unmounted.
     */
    if (chdir("/") < 0)
        err_quit("%s: can't change directory to /", cmd);

    /*
     * Close all open file descriptors.
     */
    if (rl.rlim_max == RLIM_INFINITY)
        rl.rlim_max = 1024;
    for (i = 0; i < rl.rlim_max; i++)
        close(i);

    /*
     * Attach file descriptors 0, 1, and 2 to /dev/null.
     */
    fd0 = open("/dev/null", O_RDWR);
    fd1 = dup(0);
    fd2 = dup(0);

    /*
     * Initialize the log file.
     */
    openlog(cmd, LOG_CONS, LOG_DAEMON);
    if (fd0 != 0 || fd1 != 1 || fd2 != 2) {
        syslog(LOG_ERR, "unexpected file descriptors %d %d %d",
          fd0, fd1, fd2);
        exit(1);
    }
}
```

Figure 13.1 Initialize a daemon process

If the daemonize function is called from a main program that then goes to sleep, we can check the status of the daemon with the ps command:

```
$ ./a.out
$ ps -efj
UID      PID  PPID  PGID   SID  TTY  CMD
sar    13800     1 13799 13799  ?    ./a.out
$ ps -efj | grep 13799
sar    13800     1 13799 13799  ?    ./a.out
```

We can also use ps to verify that no active process exists with ID 13799. This means that our daemon is in an orphaned process group (Section 9.10) and is not a session leader and, therefore, has no chance of allocating a controlling terminal. This is a result of performing the second fork in the daemonize function. We can see that our daemon has been initialized correctly. □

13.4 Error Logging

One problem a daemon has is how to handle error messages. It can't simply write to standard error, since it shouldn't have a controlling terminal. We don't want all the daemons writing to the console device, because on many workstations the console device runs a windowing system. We also don't want each daemon writing its own error messages into a separate file. It would be a headache for anyone administering the system to keep up with which daemon writes to which log file and to check these files on a regular basis. A central daemon error-logging facility is required.

> The BSD syslog facility was developed at Berkeley and used widely in 4.2BSD. Most systems derived from BSD support syslog. Until SVR4, System V never had a central daemon logging facility. The syslog function is included in the XSI option in the Single UNIX Specification.

The BSD syslog facility has been widely used since 4.2BSD. Most daemons use this facility. Figure 13.2 illustrates its structure.

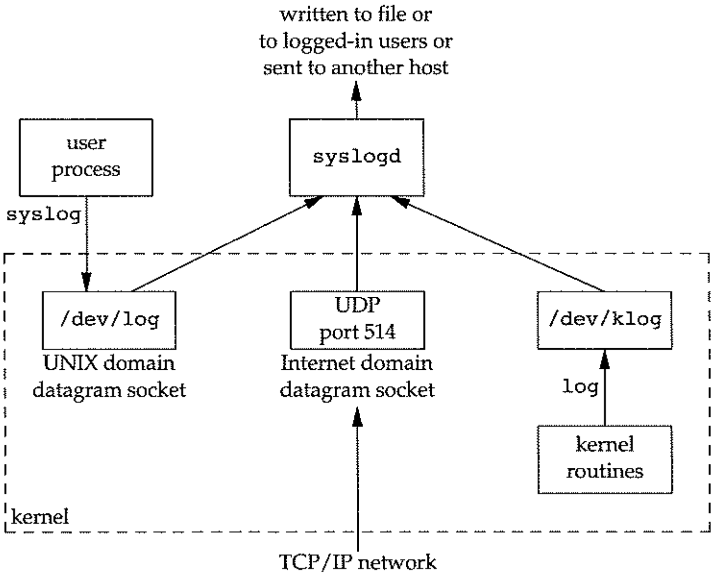

Figure 13.2 The BSD syslog facility

There are three ways to generate log messages:

1. Kernel routines can call the `log` function. These messages can be read by any user process that `open`s and `read`s the `/dev/klog` device. We won't describe this function any further, since we're not interested in writing kernel routines.
2. Most user processes (daemons) call the `syslog`(3) function to generate log messages. We describe its calling sequence later. This causes the message to be sent to the UNIX domain datagram socket `/dev/log`.
3. A user process on this host, or on some other host that is connected to this host by a TCP/IP network, can send log messages to UDP port 514. Note that the `syslog` function never generates these UDP datagrams: they require explicit network programming by the process generating the log message.

Refer to Stevens, Fenner, and Rudoff [2004] for details on UNIX domain sockets and UDP sockets.

Normally, the `syslogd` daemon reads all three forms of log messages. On start-up, this daemon reads a configuration file, usually `/etc/syslog.conf`, which determines where different classes of messages are to be sent. For example, urgent messages can be sent to the system administrator (if logged in) and printed on the console, whereas warnings may be logged to a file.

Our interface to this facility is through the `syslog` function.

```
#include <syslog.h>

void openlog(const char *ident, int option, int facility);

void syslog(int priority, const char *format, ...);

void closelog(void);

int setlogmask(int maskpri);
```

Returns: previous log priority mask value

Calling `openlog` is optional. If it's not called, the first time `syslog` is called, `openlog` is called automatically. Calling `closelog` is also optional—it just closes the descriptor that was being used to communicate with the `syslogd` daemon.

Calling `openlog` lets us specify an *ident* that is added to each log message. This is normally the name of the program (e.g. `cron`, `inetd`). The *option* argument is a bitmask specifying various options. Figure 13.3 describes the available options, including a bullet in the XSI column if the option is included in the `openlog` definition in the Single UNIX Specification.

The *facility* argument for `openlog` is taken from Figure 13.4. Note that the Single UNIX Specification defines only a subset of the facility codes typically available on a given platform. The reason for the *facility* argument is to let the configuration file specify that messages from different facilities are to be handled differently. If we don't call `openlog`, or if we call it with a *facility* of 0, we can still specify the facility as part of the *priority* argument to `syslog`.

We call `syslog` to generate a log message. The *priority* argument is a combination of the *facility*, shown in Figure 13.4, and a *level*, shown in Figure 13.5. These *level*s are ordered by priority, from highest to lowest.

option	XSI	Description
LOG_CONS	•	If the log message can't be sent to `syslogd` via the UNIX domain datagram, the message is written to the console instead.
LOG_NDELAY	•	Open the UNIX domain datagram socket to the `syslogd` daemon immediately; don't wait until the first message is logged. Normally, the socket is not opened until the first message is logged.
LOG_NOWAIT	•	Do not wait for child processes that might have been created in the process of logging the message. This prevents conflicts with applications that catch `SIGCHLD`, since the application might have retrieved the child's status by the time that `syslog` calls `wait`.
LOG_ODELAY	•	Delay the opening of the connection to the `syslogd` daemon until the first message is logged.
LOG_PERROR		Write the log message to standard error in addition to sending it to `syslogd`. (Unavailable on Solaris.)
LOG_PID	•	Log the process ID with each message. This is intended for daemons that `fork` a child process to handle different requests (as compared to daemons, such as `syslogd`, that never call `fork`).

Figure 13.3 The *option* argument for `openlog`

The *format* argument and any remaining arguments are passed to the `vsprintf` function for formatting. Any occurrences of the characters `%m` in *format* are first replaced with the error message string (`strerror`) corresponding to the value of `errno`.

The `setlogmask` function can be used to set the log priority mask for the process. This function returns the previous mask. When the log priority mask is set, messages are not logged unless their priority is set in the log priority mask. Note that attempts to set the log priority mask to 0 will have no effect.

The `logger`(1) program is also provided by many systems as a way to send log messages to the `syslog` facility. Some implementations allow optional arguments to this program, specifying the *facility*, *level*, and *ident*, although the Single UNIX Specification doesn't define any options. The `logger` command is intended for a shell script running noninteractively that needs to generate log messages.

Example

In a (hypothetical) line printer spooler daemon, you might encounter the sequence

```
openlog("lpd", LOG_PID, LOG_LPR);
syslog(LOG_ERR, "open error for %s: %m", filename);
```

The first call sets the *ident* string to the program name, specifies that the process ID should always be printed, and sets the default *facility* to the line printer system. The call to `syslog` specifies an error condition and a message string. If we had not called `openlog`, the second call could have been

```
syslog(LOG_ERR | LOG_LPR, "open error for %s: %m", filename);
```

Here, we specify the *priority* argument as a combination of a *level* and a *facility*. □

facility	XSI	Description
LOG_AUDIT		the audit facility
LOG_AUTH		authorization programs: `login`, `su`, `getty`, ...
LOG_AUTHPRIV		same as LOG_AUTH, but logged to file with restricted permissions
LOG_CONSOLE		messages written to `/dev/console`
LOG_CRON		`cron` and `at`
LOG_DAEMON		system daemons: `inetd`, `routed`, ...
LOG_FTP		the FTP daemon (`ftpd`)
LOG_KERN		messages generated by the kernel
LOG_LOCAL0	●	reserved for local use
LOG_LOCAL1	●	reserved for local use
LOG_LOCAL2	●	reserved for local use
LOG_LOCAL3	●	reserved for local use
LOG_LOCAL4	●	reserved for local use
LOG_LOCAL5	●	reserved for local use
LOG_LOCAL6	●	reserved for local use
LOG_LOCAL7	●	reserved for local use
LOG_LPR		line printer system: `lpd`, `lpc`, ...
LOG_MAIL		the mail system
LOG_NEWS		the Usenet network news system
LOG_NTP		the network time protocol system
LOG_SECURITY		the security subsystem
LOG_SYSLOG		the `syslogd` daemon itself
LOG_USER	●	messages from other user processes (default)
LOG_UUCP		the UUCP system

Figure 13.4 The *facility* argument for `openlog`

level	Description
LOG_EMERG	emergency (system is unusable) (highest priority)
LOG_ALERT	condition that must be fixed immediately
LOG_CRIT	critical condition (e.g., hard device error)
LOG_ERR	error condition
LOG_WARNING	warning condition
LOG_NOTICE	normal, but significant condition
LOG_INFO	informational message
LOG_DEBUG	debug message (lowest priority)

Figure 13.5 The `syslog` *levels* (ordered)

In addition to `syslog`, many platforms provide a variant that handles variable argument lists.

```
#include <syslog.h>
#include <stdarg.h>

void vsyslog(int priority, const char *format, va_list arg);
```

All four platforms described in this book provide `vsyslog`, but this function is not included in the Single UNIX Specification. Note that to make its declaration visible to your application, you might need to define an additional symbol, such as __BSD_VISIBLE on FreeBSD or __USE_BSD on Linux.

Most syslogd implementations will queue messages for a short time. If a duplicate message arrives during this period, the syslogd daemon will not write it to the log. Instead, the daemon prints a message similar to "last message repeated *N* times."

13.5 Single-Instance Daemons

Some daemons are implemented so that only a single copy of the daemon should be running at a time for proper operation. Such a daemon might need exclusive access to a device, for example. In the case of the cron daemon, if multiple instances were running, each copy might try to start a single scheduled operation, resulting in duplicate operations and probably an error.

If the daemon needs to access a device, the device driver will sometimes prevent multiple attempts to open the corresponding device node in /dev. This restricts us to one copy of the daemon running at a time. If no such device is available, however, we need to do the work ourselves.

The file- and record-locking mechanism provides the basis for one way to ensure that only one copy of a daemon is running. (We discuss file and record locking in Section 14.3.) If each daemon creates a file with a fixed name and places a write lock on the entire file, only one such write lock will be allowed to be created. Successive attempts to create write locks will fail, serving as an indication to successive copies of the daemon that another instance is already running.

File and record locking provides a convenient mutual-exclusion mechanism. If the daemon obtains a write-lock on an entire file, the lock will be removed automatically if the daemon exits. This simplifies recovery, eliminating the need for us to clean up from the previous instance of the daemon.

Example

The function shown in Figure 13.6 illustrates the use of file and record locking to ensure that only one copy of a daemon is running.

```
#include <unistd.h>
#include <stdlib.h>
#include <fcntl.h>
#include <syslog.h>
#include <string.h>
#include <errno.h>
#include <stdio.h>
#include <sys/stat.h>

#define LOCKFILE "/var/run/daemon.pid"
#define LOCKMODE (S_IRUSR|S_IWUSR|S_IRGRP|S_IROTH)

extern int lockfile(int);
```

```
int
already_running(void)
{
    int     fd;
    char    buf[16];

    fd = open(LOCKFILE, O_RDWR|O_CREAT, LOCKMODE);
    if (fd < 0) {
        syslog(LOG_ERR, "can't open %s: %s", LOCKFILE, strerror(errno));
        exit(1);
    }
    if (lockfile(fd) < 0) {
        if (errno == EACCES || errno == EAGAIN) {
            close(fd);
            return(1);
        }
        syslog(LOG_ERR, "can't lock %s: %s", LOCKFILE, strerror(errno));
        exit(1);
    }
    ftruncate(fd, 0);
    sprintf(buf, "%ld", (long)getpid());
    write(fd, buf, strlen(buf)+1);
    return(0);
}
```

Figure 13.6 Ensure that only one copy of a daemon is running

Each copy of the daemon will try to create a file and write its process ID in the file. This will allow administrators to identify the process easily. If the file is already locked, the `lockfile` function will fail with `errno` set to `EACCES` or `EAGAIN`, so we return 1, indicating that the daemon is already running. Otherwise, we truncate the file, write our process ID to it, and return 0.

We need to truncate the file, because the previous instance of the daemon might have had a process ID larger than ours, with a larger string length. For example, if the previous instance of the daemon was process ID 12345, and the new instance is process ID 9999, when we write the process ID to the file, we will be left with 99995 in the file. Truncating the file prevents data from the previous daemon appearing as if it applies to the current daemon. □

13.6 Daemon Conventions

Several common conventions are followed by daemons in the UNIX System.

- If the daemon uses a lock file, the file is usually stored in `/var/run`. Note, however, that the daemon might need superuser permissions to create a file here. The name of the file is usually *name*`.pid`, where *name* is the name of the daemon or the service. For example, on Linux, the name of the `cron` daemon's lock file is `/var/run/crond.pid`.

- If the daemon supports configuration options, they are usually stored in /etc. The configuration file is named *name*.conf, where *name* is the name of the daemon or the name of the service. For example, the configuration for the syslogd daemon is usually /etc/syslog.conf.
- Daemons can be started from the command line, but they are usually started from one of the system initialization scripts (/etc/rc* or /etc/init.d/*). If the daemon should be restarted automatically when it exits, we can arrange for init to restart it if we include a respawn entry for it in /etc/inittab (assuming the system uses a System V style init command).
- If a daemon has a configuration file, the daemon reads the file when it starts, but usually won't look at it again. If an administrator changes the configuration, the daemon would need to be stopped and restarted to account for the configuration changes. To avoid this, some daemons will catch SIGHUP and reread their configuration files when they receive the signal. Since they aren't associated with terminals and are either session leaders without controlling terminals or members of orphaned process groups, daemons have no reason to expect to receive SIGHUP. Thus they can safely reuse it.

Example

The program shown in Figure 13.7 shows one way a daemon can reread its configuration file. The program uses sigwait and multiple threads, as discussed in Section 12.8.

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

sigset_t    mask;

extern int already_running(void);

void
reread(void)
{
    /* ... */
}

void *
thr_fn(void *arg)
{
    int err, signo;

    for (;;) {
        err = sigwait(&mask, &signo);
        if (err != 0) {
            syslog(LOG_ERR, "sigwait failed");
            exit(1);
        }

        switch (signo) {
        case SIGHUP:
            syslog(LOG_INFO, "Re-reading configuration file");
            reread();
            break;

        case SIGTERM:
            syslog(LOG_INFO, "got SIGTERM; exiting");
            exit(0);

        default:
            syslog(LOG_INFO, "unexpected signal %d\n", signo);
        }
    }
    return(0);
}

int
main(int argc, char *argv[])
{
    int                 err;
    pthread_t           tid;
    char                *cmd;
    struct sigaction    sa;

    if ((cmd = strrchr(argv[0], '/')) == NULL)
        cmd = argv[0];
    else
        cmd++;

    /*
     * Become a daemon.
     */
    daemonize(cmd);

    /*
     * Make sure only one copy of the daemon is running.
     */
    if (already_running()) {
        syslog(LOG_ERR, "daemon already running");
        exit(1);
    }

    /*
     * Restore SIGHUP default and block all signals.
     */
    sa.sa_handler = SIG_DFL;
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = 0;
    if (sigaction(SIGHUP, &sa, NULL) < 0)
        err_quit("%s: can't restore SIGHUP default");
    sigfillset(&mask);
    if ((err = pthread_sigmask(SIG_BLOCK, &mask, NULL)) != 0)
        err_exit(err, "SIG_BLOCK error");

    /*
     * Create a thread to handle SIGHUP and SIGTERM.
     */
    err = pthread_create(&tid, NULL, thr_fn, 0);
    if (err != 0)
        err_exit(err, "can't create thread");

    /*
     * Proceed with the rest of the daemon.
     */
    /* ... */
    exit(0);
}
```

Figure 13.7 Daemon rereading configuration files

We call daemonize from Figure 13.1 to initialize the daemon. When it returns, we call already_running from Figure 13.6 to ensure that only one copy of the daemon is running. At this point, SIGHUP is still ignored, so we need to reset the disposition to the default behavior; otherwise, the thread calling sigwait may never see the signal.

We block all signals, as is recommended for multithreaded programs, and create a thread to handle signals. The thread's only job is to wait for SIGHUP and SIGTERM. When it receives SIGHUP, the thread calls reread to reread its configuration file. When it receives SIGTERM, the thread logs a message and exits.

Recall from Figure 10.1 that the default action for SIGHUP and SIGTERM is to terminate the process. Because we block these signals, the daemon will not die when one of them is sent to the process. Instead, the thread calling sigwait will return with an indication that the signal has been received. □

Example

Not all daemons are multithreaded. The program in Figure 13.8 shows how a single-threaded daemon can catch SIGHUP and reread its configuration file.

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

extern int lockfile(int);
extern int already_running(void);

void
reread(void)
{
    /* ... */
```

```
}

void
sigterm(int signo)
{
    syslog(LOG_INFO, "got SIGTERM; exiting");
    exit(0);
}

void
sighup(int signo)
{
    syslog(LOG_INFO, "Re-reading configuration file");
    reread();
}

int
main(int argc, char *argv[])
{
    char                *cmd;
    struct sigaction    sa;

    if ((cmd = strrchr(argv[0], '/')) == NULL)
        cmd = argv[0];
    else
        cmd++;

    /*
     * Become a daemon.
     */
    daemonize(cmd);

    /*
     * Make sure only one copy of the daemon is running.
     */
    if (already_running()) {
        syslog(LOG_ERR, "daemon already running");
        exit(1);
    }

    /*
     * Handle signals of interest.
     */
    sa.sa_handler = sigterm;
    sigemptyset(&sa.sa_mask);
    sigaddset(&sa.sa_mask, SIGHUP);
    sa.sa_flags = 0;
    if (sigaction(SIGTERM, &sa, NULL) < 0) {
        syslog(LOG_ERR, "can't catch SIGTERM: %s", strerror(errno));
        exit(1);
    }
    sa.sa_handler = sighup;
    sigemptyset(&sa.sa_mask);
    sigaddset(&sa.sa_mask, SIGTERM);
    sa.sa_flags = 0;
    if (sigaction(SIGHUP, &sa, NULL) < 0) {
        syslog(LOG_ERR, "can't catch SIGHUP: %s", strerror(errno));
        exit(1);
    }

    /*
     * Proceed with the rest of the daemon.
     */
    /* ... */
    exit(0);
}
```

Figure 13.8 Alternative implementation of daemon rereading configuration files

After initializing the daemon, we install signal handlers for `SIGHUP` and `SIGTERM`. We can either place the reread logic in the signal handler or just set a flag in the handler and have the main thread of the daemon do all the work instead. □

13.7 Client–Server Model

A common use for a daemon process is as a server process. Indeed, in Figure 13.2, we can call the `syslogd` process a server that has messages sent to it by user processes (clients) using a UNIX domain datagram socket.

In general, a *server* is a process that waits for a *client* to contact it, requesting some type of service. In Figure 13.2, the service being provided by the `syslogd` server is the logging of an error message.

In Figure 13.2, the communication between the client and the server is one way. The client sends its service request to the server; the server sends nothing back to the client. In the upcoming chapters, we'll see numerous examples of two-way communication between a client and a server—the client sends a request to the server, and the server sends a reply back to the client.

It is common to find servers that `fork` and `exec` another program to provide service to a client. These servers often manage multiple file descriptors: communication endpoints, configuration files, log files, and the like. At best, it would be careless to leave these file descriptors open in the child process, because they probably won't be used in the program executed by the child, especially if the program is unrelated to the server. At worst, leaving them open could pose a security problem—the program executed could do something malicious, such as change the server's configuration file or trick the client into thinking it is communicating with the server, thereby gaining access to unauthorized information.

An easy solution to this problem is to set the close-on-exec flag for all file descriptors that the executed program won't need. Figure 13.9 shows a function that we can use in a server process to do just this.

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

int
set_cloexec(int fd)
{
    int     val;

    if ((val = fcntl(fd, F_GETFD, 0)) < 0)
        return(-1);

    val |= FD_CLOEXEC;       /* enable close-on-exec */

    return(fcntl(fd, F_SETFD, val));
}
```

Figure 13.9 Set close-on-exec flag

13.8 Summary

Daemon processes are running all the time on most UNIX systems. Initializing our own process to run as a daemon takes some care and an understanding of the process relationships described in Chapter 9. In this chapter, we developed a function that can be called by a daemon process to initialize itself correctly.

We also discussed the ways a daemon can log error messages, since a daemon normally doesn't have a controlling terminal. We discussed several conventions that daemons follow on most UNIX systems and showed examples of how to implement some of these conventions.

Exercises

13.1 As we might guess from Figure 13.2, when the `syslog` facility is initialized, either by calling `openlog` directly or on the first call to `syslog`, the special device file for the UNIX domain datagram socket, `/dev/log`, has to be opened. What happens if the user process (the daemon) calls `chroot` before calling `openlog`?

13.2 Recall the sample `ps` output from Section 13.2. The only user-level daemon that isn't a session leader is the `rsyslogd` process. Explain why the `syslogd` daemon isn't a session leader.

13.3 List all the daemons active on your system, and identify the function of each one.

13.4 Write a program that calls the `daemonize` function in Figure 13.1. After calling this function, call `getlogin` (Section 8.15) to see whether the process has a login name now that it has become a daemon. Print the results to a file.