Daemon Processes

Introduction

Daemons are processes that 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, they run in the background. UNIX systems have numerous daemons that perform day-to-day activities.

This chapter details the process structure of daemons and explores 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.

Daemon Characteristics

This section describes some common system daemons with the concepts of process groups, controlling terminals, and sessions as described in Chapter 9.

```
ps -axj
```

- The `-a` option shows the status of processes owned by others.
- The `-x` option shows processes that don't have a controlling terminal.
- The `-j` option displays the job-related information:
    - Session ID
    - Process group ID
    - Controlling terminal
    - Terminal process group ID

The output from `ps` on Linux 3.2.0 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
```

The column headings, in order, are:

- User ID
- Process ID
- Parent process ID
- Process group ID
- Session ID
- Terminal name
- Command string

The system processes in this output 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), except `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 (above) sample `ps` output, kernel daemons has their names in square brackets.

- `kthreadd` is a special kernel process on Linux that creates other kernel process, and thus appears as the parent of other kernel daemons. A kernel component, which need to run in a process context but isn't invoked from the context of a user-level process, will usually have its own kernel daemon. For example:
    - `kswapd`: pageout daemon. It supports the virtual memory subsystem by writing dirty pages to disk slowly over time, so the pages can be reclaimed.
    - `flush`.
        - This 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 with one for each backing device. The sample output `flush-8:0` means 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
- `init` (`launchd` on Mac OS X), usually Process 1, is a system daemon responsible for, among other things, starting system services specific to various run levels.
- `rpcbind` provides the service of mapping RPC (Remote Procedure Call) program numbers to network port numbers.
- 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.
- `rsyslogd` can log system messages of any program. The messages may be printed on a console device and/or written to a file.
- `cron` executes commands at regularly scheduled dates and times. Numerous system administration tasks are handled by cron running programs at regularly intervals.
- `atd`, similar to `cron`, allows users to execute jobs at specified times, only once.
- `cupsd` is a print spooler that handles print requests on the system.
- `sshd` provides secure remote login and execution facilities.

Some notes:

- 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`.)
- The parent of the user-level daemons is the `init` process.

Coding Rules

This section states basic rules to coding a daemon prevent unwanted interactions from happening, followed by a function `daemonize`, that implements these rules.

1. **Call `umask` to set the file mode creation mask** to a known value, usually 0.
    - If the daemon process creates files, it may want to set specific permissions.
    - 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:
    - If the daemon was started as a simple shell command, having the parent terminate makes the shell think that the command is done
    - 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. (See Ensuring the successful call of setsid in Chapter 9)
3. **Call `setsid` to create a new session**. The three steps listed in Section 9.5 occur. The process:

    - becomes the leader of a new session,
    - becomes the leader of a new process group,
    - and 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 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. [p466-467]

The code below shows the `daemonize` function that can be called from a program that wants to initialize itself as a daemon.

daemons/init.c

```
#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);
    }
}
```

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 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.

Error Logging

One problem a daemon has is how to handle error messages. It cannot (simply) write to:

- Standard error: it shouldn't have a controlling terminal.
- Console device: on many workstations the console device runs a windowing system.
- Separate files: it's a headache to keep up 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 has been widely used since 4.2BSD. Most daemons use this facility. The following figure illustrates its structure:

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.
2. Most user processes (daemons) call the `syslog(3)` function to generate log messages. This causes the message to be sent to the UNIX domain datagram socket `/dev/log`.
3. A user process on this host or 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.

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.

```c
#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 closes the descriptor being used to communicate with the syslogd daemon.
- The `openlog` function:
    - *indent* argument is the name of the program.
    - *option* argument is a bitmask specifying various options. (see the table below)

- *facility* argument lets 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. (see the table below)

option	**XSI**	**Description**
`LOG_CONS`	x	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`	x	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`	x	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`	x	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`	x	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`).

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

facility	XSI	Description
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 ~ LOG_LOCAL7	x	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		

facility	XSI	Description
LOG_UUCP		the UUCP system

- The `syslog` function:
    - The *priority* argument is a combination of the *facility* and a *level* (shown in the table below). These *level*s are ordered by priority, from highest to lowest.
    - The *format* argument and any remaining arguments are passed to the `vsprintf` function for formatting. Any occurrences of the characters `%m` are first replaced with the error message string (`strerror`) corresponding to the value of `errno`.

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)

- The `setlogmask` function sets the log priority mask ("logmask") for the process and 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.

The `logger(1)` program is also provided by many systems as a way to send log messages to the `syslog` facility. The `logger` command is intended for a shell script running noninteractively that needs to generate log messages.

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);
```

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

Example of `syslog`

In a 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 to `openlog` 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*.

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. This kind of daemon might need exclusive access to a device. For example 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.

The file- and record-locking mechanism (Section 14.3) is a way to ensure that only one copy of a daemon is running. 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.

[p473]

Example of using file locking to ensure single copy of daemon

daemons/single.c

```
#include <unistd.h>
#include <stdlib.h>
#include <fcntl.h>
#include <syslog.h>
```

```c
#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);
}
```

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`.

[p474]

Daemon Conventions

Common conventions are followed by daemons in the UNIX System.

- The lock file is usually stored in `/var/run`.
  - Creating a file in this directory requires superuser permissions;
  - The name of the file is usually *name*`.pid`, where name is the name of the daemon or the service. For example, the name of the Linux `cron` daemon's lock file is `/var/run/crond.pid`.
- The configuration options are usually stored in `/etc`.
  - The configuration file is named *name*`.conf`, where name is the name of the daemon or 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 and 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.

Examples:

- daemons/reread.c
- daemons/reread2.c

Client–Server Model

A common use for a daemon process is as a server process. syslogd process (Figure 13.2) is a server that has messages sent to it by user processes (clients) using a UNIX domain datagram socket.

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

Servers usually `fork` and `exec` another program to provide service to a client. These servers often manage multiple file descriptors (e.g. communication endpoints, configuration files, log files). 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. The following function does that:

lib/setfd.c

```
#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));
}
```