

# How To Write a UNIX Daemon

*Dave Lennert*

Hewlett-Packard Company

## ABSTRACT

On UNIX† systems users can easily write daemon programs that perform repetitive tasks in an unnoticed way. However, because daemon programs typically run outside a login session context and because most programmers are unfamiliar with designing a program to run outside this context, there are many subtle pitfalls that can prevent a daemon from being coded correctly. Further, the incompatibilities between various major UNIX variants compound these pitfalls. This paper discusses these pitfalls and how to avoid them.

Daemon: runs around in the shadows (background) doing devilish deeds.

*found in some daemon source code*

## Introduction

A daemon is a program which performs periodic tasks in such a manner that it is normally unnoticed by users.

Some daemons run constantly, waiting for a significant event. Examples include *init* which respawns login sessions (*gettys*) as they end, *cron* which launches programs at specified times, and *sendmail* which listens on a socket for incoming mail messages.

Other daemons are launched periodically and terminate after completing one execution of their task. Such daemons include the uucp file transfer utility, *uucico*, which can be launched as a login shell when a remote machine logs in, *calendar* which is launched nightly by *cron* to examine users' calendars and mail them notification of upcoming events, and various *mail* sending utilities which allow the user's shell to continue while the collected mail message is delivered asynchronously.

Daemon programs are very easy to write in the UNIX environment. They can be written by casual users and launched periodically via the *at* command or, on System V, by a user's personal *crontab* file, or perhaps at each login via *ssh's* *.login* command file. System administrators write daemons whenever they recognize a particular administrative task is becoming routine enough to be handled automatically.

However, daemon programs appear easier to write correctly than they really are. This is because there are many quirks and side effects of UNIX which are automatically taken care of in a login session context but not in a detached, daemon program. The *init*, *getty*, *login*, and shell programs oversee such functions as setting up user ID's, establishing process groups, allocating controlling terminals, and managing job control.

---

† UNIX is, of course, a registered trademark of AT&T in the U.S. and other countries. It is also fast becoming a household word, like Kleenex.‡

‡ Kleenex is a registered trademark of the Kimberly-Clark Corporation and should probably only be used as an adjective as in "Kleenex tissues".

If a daemon process is launched outside a login session (e.g., via `/etc/rc` or its ilk during system startup) then it needs to manage these functions itself explicitly. If a daemon process is launched from within a login session (e.g., as a background command from a login shell) then it needs to undo much of what the login process sequence has done. In order to code a daemon robustly, both concerns must be addressed.

This paper discusses these concerns and the methods for addressing them. Note that all the example coding fragments lack necessary error condition checking or handling; such handling should, of course, be added to any real daemon.

## Programming Rules

The following is a set of programming rules which avoid several subtle pitfalls. A discussion of each pitfall is also given along with the rule.

### *Make immune to background job control write checks.*

On systems which support 4.2BSD style job control, daemons which attempt I/O to their controlling terminal will stop if they were launched from `csh` in the background (with `"&"`). The real way around this is to disassociate yourself from your controlling terminal (see below). But in some cases, the daemon will want to perform some setup checks and output error messages before it loses its controlling terminal.

There is no way to allow a background process to read from its controlling tty. However, output can be reliably performed if the calling process ignores the SIGTTOU signal, as in:

```
#ifdef SIGTTOU
signal(SIGTTOU, SIG_IGN);
#endif
```

For safety, it is probably a good idea to ignore the other stop signals as well, as in:

```
#ifdef SIGTTIN
signal(SIGTTIN, SIG_IGN);
#endif

#ifdef SIGTSTP
signal(SIGTSTP, SIG_IGN);
#endif
```

Ignoring SIGTTIN also has the side effect of causing all background attempts to read from the controlling terminal to fail immediately and return the EIO error.

### *Close all open file descriptors, especially stdin, stdout, stderr.*

Do not leave stray file descriptors open. More importantly, if any of the file descriptors are terminal devices then they must be closed to allow reset of the terminal state during logout (see below). The typical code sequence is:

```
for (fd = 0; fd < _NFILE; fd++)
    close(fd); /* close all file descriptors */
```

### *Disassociate from your process group and controlling terminal.*

Daemons launched during a login session inherit both the controlling terminal and the process group of that session (or, in the case of job control, of that job within the session).

As long as the daemon is still in the process group associated with a controlling terminal it is subject to terminal-generated signals (such as SIGINT or SIGHUP). As long as the daemon still has a controlling terminal it is subject to job control terminal I/O restrictions on systems which support job control.

Further, while the daemon remains in the original process group in which it started, it is subject to any signals sent to that process group as a whole by another program via *kill(2)*.

One way to prevent the daemon from receiving these "unintended" signals is simply to ignore all signals. However, this means that the signals cannot be used by the daemon for other purposes (such as rudimentary interprocess communication). Also, this approach is insufficient because there are some signals which a process cannot ignore (for example, SIGKILL or SIGSTOP).

A better approach is for the daemon to disassociate itself from both the controlling terminal and from the process group which it inherited. On 4.2BSD systems, the former can be performed via the TIOCNOTTY *ioctl(2)* and the latter via *setpgrp(2)*. Under AT&T UNIX, *setpgrp(2)* performs both functions.

However, (under AT&T UNIX) in order for *setpgrp(2)* to have its desired effect, this must be the first time the process has called *setpgrp(2)*; that is, the process must not already be a process group leader. (A process group leader is a process whose process group ID is equal to its process ID.) Since a program has no control over the process which *exec(2)*'d it, it must *fork(2)* to ensure that it is not already a process group leader before calling *setpgrp(2)*. (This is especially important if the daemon is launched from a *cs*h which supports job control since *cs*h automatically makes its children process group leaders. But this also happens, for example, when an imprudent user launches a daemon from a login shell via the *exec* command.)

In order to prevent locking up a user's terminal when a daemon is started (i.e., without "&"), the daemon usually *fork(2)*'s anyway and runs in the child while the parent immediately *exit(2)*'s without waiting for the child. This causes the shell to believe that the daemon has terminated.

A typical code sequence would be:

```
if (fork() != 0)
    exit(0);    /* parent */

/* child */

#ifdef BSD
setpgrp(0, getpid());    /* change process group */

if ((fd = open("/dev/tty", O_RDWR)) >= 0) {
    ioctl(fd, TIOCNOTTY, 0);    /* lose controlling terminal */
    close(fd);
}

#else /* AT&T */

setpgrp();    /* lose controlling terminal & change process group */

#endif
```

*Do not reacquire a controlling terminal.*

Once the daemon is a process group leader without a controlling terminal (having called *setpgrp(2)* as described above) it is now potentially capable of reacquiring a controlling terminal. If it does, other processes (for example, logins) will not be able to acquire the terminal correctly as their controlling

terminal.

(Interestingly, this problem does not exist under 4.2BSD. Unlike AT&T UNIX, where a terminal can only be acquired as a controlling terminal if it is not already a controlling terminal, 4.2BSD allows a process to join an already allocated controlling terminal and its process group. Basically, the process merges with the already established process group.)

The symptoms of this problem are somewhat subtle. Since *getty* and *login* are not able to acquire a controlling terminal, the special file, `/dev/tty`, cannot be successfully opened. Because of this, the *getpass(3)* routine, used by *login* to obtain the user's password, fails without ever printing the `Password:` prompt. All login attempts for accounts with passwords silently fail without ever prompting for a password. Login attempts for accounts without passwords succeed (because *getpass(3)* is never called), however the login shell does not have a controlling terminal. Terminal input and output still succeeds (via *stdin*, *stdout*, and *stderr*), but any keyboard signals are not sent to the processes spawned during this login session. Instead the signals are sent to the process which acquired this terminal as its controlling terminal (the daemon) and its descendants.

For this reason the daemon program must ensure that it does not reacquire a controlling terminal.

On 4.2BSD systems, a new controlling terminal can only be acquired by a process with a process group ID of zero. After calling *setpgrp(2)* to set its process group ID equal to its process ID, the daemon cannot reacquire a controlling terminal.

Under AT&T UNIX, a new controlling terminal is acquired whenever a process group leader without a controlling terminal opens a terminal which is not already the controlling terminal for another process group. On such systems the daemon can reacquire a controlling terminal when opening, say, `/dev/console`, to perform logging or error reporting. Even if the daemon subsequently closes the terminal it still possesses it as a controlling terminal. There is no way to relinquish it since subsequent *setpgrp(2)* calls are ineffective. (*Setpgrp(2)* has no effect if the caller is already a process group leader.) Therefore the acquisition must be prevented.

A simple(?) way to prevent the acquisition of a new controlling terminal is to *fork(2)* yet another time *after* calling *setpgrp(2)*. The daemon actually runs in this second child and the parent (the first child) immediately *exit(2)*'s. However, (on AT&T UNIX) when the parent (first child) terminates, the `SIGHUP` signal is sent to the child since the parent is a process group leader. Thus, the parent must ignore `SIGHUP` before *fork(2)*'ing the second child otherwise the child will be killed. (The ignored setting is inherited by the child.) The final side effect of the terminating (process group leader) parent is to set the process group of the child to zero. The daemon (second child) now has no controlling terminal, it is in a new (zero) process group which is immune to signals from the tty driver, and it cannot acquire a new controlling terminal since it is not a process group leader.

Thus the typical code sequence becomes:

```
if (fork() != 0)
    exit(0);          /* first parent */

/* first child */
setpgrp();          /* lose controlling terminal & change process group */

signal(SIGHUP, SIG_IGN); /* immune from pgrp leader death */
if (fork() != 0)    /* become non-pgrp-leader */
    exit(0);        /* first child */

/* second child */
```

*Do not "hold" open tty files.*

Even after ensuring that the daemon will not reacquire a controlling terminal when a terminal device is opened, there is a further concern:

Terminal state settings, such as BAUD rate and signal character definitions, are only reset to the default state when the last process having the terminal open finally closes it. Thus, if the daemon has a terminal open continuously, then the last close never happens and the terminal settings are not reset at logout.

Typical examples of terminal files held open by a daemon are `stdin`, `stdout`, `stderr`, and `/dev/console`.

It's probably best to log errors and status messages to a disk file rather than a terminal. However, when terminal logging is desired, the "correct" method is to hold the terminal open only long enough to perform a single logging transaction. Note that this logging transaction still represents a window of time during which a logout would not reset the terminal state.

4.2BSD systems have a further problem which makes this suggestion mandatory. Whenever a new login session is initiated via *getty* or its ilk, the *vhangup(2)* system call is invoked to prevent any existing process from continuing to access the login terminal. This results in read and write permissions being removed from any currently open file descriptor which references the login terminal; this affects all processes regardless of user ID. Therefore, daemons which access a terminal that is also used for regular login sessions, must reopen it whenever access is desired. If a file descriptor for such a terminal is continuously held open, it is very likely that *vhangup(2)* will quickly destroy its usefulness.

To determine if an unknown file descriptor is a terminal device use *isatty(3)*.

*Change current directory to "/".*

Each process has a current working directory. The kernel holds this directory file open during the life of the process. If a process has a current directory on a mounted file system, the file system is "in use" and cannot be dismounted by the administrator without finding and killing this process. (The hard part is finding the process!) Unless a process explicitly alters this via *chdir(2)*, it inherits the current directory of its parent. When launched from an interactive shell, the current directory will be whatever the user has most recently selected via the *cd* command.

Because of this, daemons should adopt a current directory which is not located on a mounted file system (assuming that the daemon's purpose allows this). The root file system, "/", is the most reliable choice. The simple call is:

```
chdir("/");
```

*Reset the file mode creation mask.*

A file mode creation mask, or *umask*, is associated with each process. It specifies how file permissions are to be restricted for each file created by the process. Like the current directory, it is inherited from the parent process and remains in effect until altered via *umask(2)*. When launched from an interactive shell, the *umask* will be whatever the user has most recently selected via the *umask(1)* command.

A daemon should reset its *umask* to an appropriate value. The typical call would be:

```
umask(0);
```

*Other attributes to worry about.*

The environment attributes discussed above are the primary ones to worry about, but the list is not exhaustive. Any attribute inherited across an *exec(2)* system call is of concern. Some other fun ones which could bite you are the nice priority value (see *nice(2)*), the time left until an alarm clock signal (see *alarm(2)*), and, on 4.2BSD systems, the signal mask and set of pending signals (see *sigvec(2)*). However, these are less likely to be *accidentally* set "wrong".

### **Interactions with *init***

The system initialization process, *init*, is responsible for directly or indirectly starting all processes on the system (with the exception of kernel processes such as the swapper or pageout process). On many versions of UNIX, *init* keeps track of all processes which it directly spawned and it can optionally respawn them if they die or it can kill them when changing to a new system *run state* (or *level*). Under AT&T UNIX, the */etc/inittab* file specifies the programs *init* should spawn in which run levels and whether or not they should be automatically respawned when they die. (Note that this file differs in both format and capabilities between System III and System V.)

Historically, system daemon programs are launched by the */etc/rc* shell script which *init* launches when moving the system from the single user run state to multi-user mode.

Some system administrators now prefer to launch daemons directly from *init* by placing the appropriate commands in */etc/inittab*. They rely on *init* respawning the daemon should it inadvertently die and on *init* killing the daemon during system state changes.

Note that the respawning and terminating capabilities of *init* depend on the spawned program not terminating prematurely. The above programming rules, however, suggest that daemons should immediately *fork(2)* and have the original process *exit(2)*. If launched from */etc/inittab*, this procedure would cause *init* to believe that the daemon was no longer running and hence it would not terminate the daemon during state changes and would instead immediately relaunch the daemon (if automatic respawn were requested). This procedure thus defeats both the respawning and terminating capabilities provided by */etc/inittab*.

What can be done to correct this? The only solution is to prevent the daemon from following the above procedure if it is launched from */etc/inittab*.

One tempting approach is for the daemon to retrieve the process ID of its parent immediately using *getppid(2)* and, if it is *init*'s process ID ("1"), skip the problematic code. However this is not perfectly reliable since any process whose original parent has terminated assumes *init* as its new parent. If a daemon is launched interactively from a user's shell, the shell might subsequently terminate before the daemon has executed the *getppid(2)* call. In short, there is a race condition. However, for practical purposes, this is a quick and easy way to solve the problem.

Another approach is to pass a command line flag to the daemon indicating whether the daemon is being launched from */etc/inittab* or not. But this requires the user to set the flag correctly during both automatic and interactive invocations. A common error would be for a user to examine the launching command in */etc/inittab* and then use it verbatim interactively.

Regardless of what approach is used, all the above mentioned pitfalls must still be recognized and avoided.

In the final analysis it seems that launching daemons from */etc/inittab*, as opposed to */etc/rc*, is unnecessary for the following reasons: (1) Relying on *init* to respawn a daemon is really masking a bug in the daemon; the daemon should never terminate by itself.† (2) Changing run states is an

---

† One interesting counterexample is that some systems (e.g., ACSnet) allow system administrators to reset things by killing the appropriate daemons. It's nice to have the daemon start correctly (i.e., right arguments) by itself through the auspices of */etc/inittab*. However, it's arguably better to have the daemon catch the termination signal and perform the reset without actually terminating; this may even be essential in the case of orderly shutdown of operations such as line printer spooling.

unusual occurrence on most systems; usually a system will move to multi-user mode and stay there.

### **Conclusions**

Without following the above rules, strange symptoms which are hard to track down often result. Many times the errant daemon program is the last thing suspected (e.g., when terminal settings are not reset after logout). Other times it is the daemon that silently and mysteriously dies (e.g., when it attempts background I/O on a job control system). Frequently these symptoms only begin occurring well after the "debug period" for the daemon.

The example below collects the above coding fragments into a single routine which a daemon calls to detach itself from the context of a login session.

```

/* Detach a daemon process from login session context.
 *
 * (This is a skeleton; add error condition checking and handling.)
 */

#include <signal.h>
#include <stdio.h>

#ifdef BSD
#include <sys/file.h>
#include <sys/ioctl.h>
#endif

sessdetach()
{
    int fd;          /* file descriptor */

    /* If launched by init (process 1), there's no need to detach.
     *
     * Note: this test is unreliable due to an unavoidable race
     * condition if the process is orphaned.
     */

    if (getppid() == 1)
        goto out;

    /* Ignore terminal stop signals */

#ifdef SIGTTOU
    signal(SIGTTOU, SIG_IGN);
#endif

#ifdef SIGTTIN
    signal(SIGTTIN, SIG_IGN);
#endif

#ifdef SIGTSTP
    signal(SIGTSTP, SIG_IGN);
#endif

    /* Allow parent shell to continue.
     * Ensure the process is not a process group leader.
     */

    if (fork() != 0)
        exit(0);    /* parent */

    /* child */

    /* Disassociate from controlling terminal and process group.
     *
     * Ensure the process can't reacquire a new controlling terminal.
     * This is done differently on BSD vs. AT&T:

```

```

*
*   BSD won't assign a new controlling terminal
*   because process group is non-zero.
*
*   AT&T won't assign a new controlling terminal
*   because process is not a process group leader.
*   (Must not do a subsequent setpgrp(!))
*/

#ifdef BSD

    setpgrp(0, getpid()); /* change process group */

    if ((fd = open("/dev/tty", O_RDWR)) >= 0) {
        ioctl(fd, TIOCNOTTY, 0); /* lose controlling terminal */
        close(fd);
    }

#else /* AT&T */

    setpgrp(); /* lose controlling terminal & change process group */

    signal(SIGHUP, SIG_IGN); /* immune from pgrp leader death */
    if (fork() != 0) /* become non-pgrp-leader */
        exit(0); /* first child */

    /* second child */

#endif

out:
    for (fd = 0; fd < _NFILE; fd++)
        close(fd); /* close all file descriptors */

    chdir("/"); /* move current directory off mounted filesystem */

    umask(0); /* clear any inherited file mode creation mask */

    return;
}

```