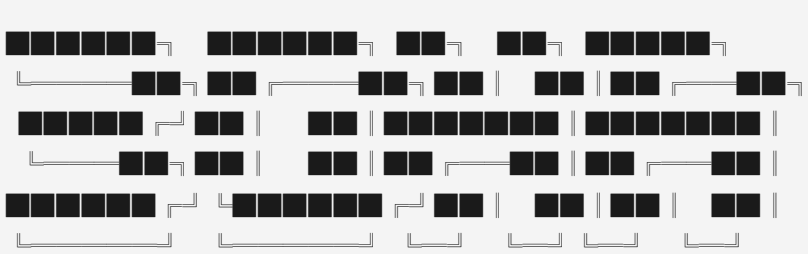
jet



Welcome to 3OHA, a place for random notes, thoughts, and factoids that I want to share or remember

---

3OHA

11 April 2022

UNIX daemonization and the double fork

Some UNIX implants are expected to effectively run as daemons. This is how Stevens teaches us to daemonize a process:

```
void
daemonize (const char *cmd)
{

    /*
     * Some missing stuff here
     */

    [...]

    if ((pid = fork()) < 0)
        err(EXIT_FAILURE, "fork fail");
    else if (pid != 0) /* parent */
        exit(0)

    /* child A */
    setsid()

    if ((pid = fork()) < 0)
        err(EXIT_FAILURE, "fork fail");
```

```
    else if (pid != 0) /* child A * /
        exit(0)

    /* child B (grandchild) */
    do_daemon_stuff();
}
```

You might wonder

1. why the double `fork()`, and
2. why the `setsid()` in the first child.

To answer both questions, you need to understand what the controlling terminal of a process is, why it is risky for a daemon to have one, and how to get rid of it.

The controlling terminal

I assume you are familiar with some fundamental concepts of job control in UNIX:

- *Process groups*. Each process in the system is a member of a process group. Process groups are identified by the process group ID (PGID). When a process is created with `fork()`, the child inherits the PGID of its parent. One key goal of process groups is that all members of a group can be signalled at once.
- *Process group leader*. Each process group has a group leader. The process group leader is the process whose PID is the same as the PGID.
- *Sessions*. Each process group is a member of a session. Sessions are identified by a session ID (SID).
- *Session leader*. Each session has a session leader. The session leader is the process whose PID is the same as its PGID and its SID. Note that only process leaders can be session leaders.
- *Session and process group changes*. Any process except the group leader can create a new process group and become its leader. Any process except the group leader can join another process group within the same session.

An important attribute of a process is its controlling terminal. Controlling terminals are, in fact, associated with sessions: each session can have at most one controlling terminal, and a controlling terminal can be associated with at most one session. When you create a process with `fork()`, the child process inherits the controlling terminal from its parent. Thus, all the processes in a session inherit the controlling terminal from the session leader.

Controlling terminals are important because they can receive signals from, and send signals to, the process group in the session which is associated with the controlling terminal.

A process can detach from its controlling terminal by creating a new session with the `setsid()` function. The remaining processes in the old session continue having the old controlling terminal. A critical situation happens when a process which has no controlling terminal opens a terminal device file, such as `/dev/tty` or `/dev/console`. Section 11.1.3 of the POSIX.1-2008 standard gives the answer:

> *If a session leader has no controlling terminal, and opens a terminal device file that is not already associated with a session without using the `O_NOCTTY` option (see `open()`), it is implementation-*

> *defined whether the terminal becomes the controlling terminal of the session leader. If a process which is not a session leader opens a terminal file, or the* `O_NOCTTY` *option is used on* `open()`*, then that terminal shall not become the controlling terminal of the calling process.*

The last sentence is the key to the double-fork technique.

Daemons and controlling terminals

Daemons should not have controlling terminals. If a daemon has a controlling terminal, it can receive signals from it that might cause it to halt or exit unexpectedly. The `setsid()` call in the code shown at the beginning of this post makes child A become the leader of a new session that only contains one process (itself). Child A also becomes the group leader of a new group and, more importantly, it will have no controlling terminal. If the parent process had one (for example, because the daemon was launched from an interactive shell), the `setsid()` call will break this association.

At this point you might wonder why to `fork()` in the first place instead of just calling `setsid()` and do the daemon stuff in the parent process. The reason is that `setsid()` will fail if the calling process is a process group leader. By calling `fork()`, we guarantee that the newly created process inherits the PGID from its parent and, therefore, is not a process leader.

The reason for the second `fork()` derives from the discussion above about whether the daemon could regain control of a controlling terminal. After the `setsid()` call, the newly created process (child A) is now a session leader and the POSIX standard leaves the door open for child A to reacquire a controlling terminal. The second `fork()` ensures that child B (the grandchild), where the daemon code will run, is not a session leader. Thus, any call to `open()` that could be manipulated to point it to a terminal file will not result in the daemon regaining a controlling terminal.

Some programmers believe that the double-fork technique is unreasonably paranoid. The daemon might not have a single call to `open()`. Even if it does, it really depends on implementation-defined specifics. Yet, if you want to be absolutely sure that your daemon cannot be tricked into acquiring a controlling terminal, the double-fork technique will give you some extra guarantees. This discussion applies to systems that follow System V semantics, such as AIX, HP-UX, and Solaris. Linux, despite not being a SysV variant, behaves like SysV for this particular case.

The BSD (and Linux) `daemon()` function

The `daemon()` function present in current Linux systems is for programs that want to detach from the controlling terminal and run as system daemons. The glibc implementation of this function is based on the function with the same name that has been present in BSD since 4.4BSD. Neither the Linux nor the BSD variants implement the double fork. In Linux, this means that a process that has been daemonized using `daemon()` might regain a controlling terminal.

Other basic steps SysV daemons should do

Traditional SysV daemons should execute a number of initial steps to prevent unwanted interactions with the environment. Chapter 13 of Stevens provides a more thorough discussion on this topic:

1. Call `umask()` to set the file mode creation mask to a value that turns off whatever permissions you want to deny in files created by the daemon.
2. Call `chdir()` to change the current working directory for the daemon to either the root directory or any other specific location where it will do its stuff.
3. Close all file descriptors that the parent might have opened and the daemon inherints.

These steps are not needed for modern Linux services because the new init systems (notably systemd) already take care of most of them. Furthermore, some of these steps interfere with system logging and monitoring in systemd, which might be a good or a bad thing depending on what kind of daemon you are writing. I will not discuss systemd daemons further here. That is an interesting topic that deserves its own post.

The daemon(7) Linux man page lists some extra steps, including:

1. Reset all signal handlers to their defaults unless you have a good reason to define your own signal handler for some signal(s).
2. Reset the signal mask using sigprocmask()
3. Remove or reset environmental variables that you do not need.
4. Connect stdin, stdout, and stderr to `/dev/null`.
5. Write the daemon PID to a file to ensure that the daemon runs only once.
6. Drop privileges, if possible.
7. Notify the process starting the daemon that the daemonization is complete.

Note that these actions make sense for a system daemon, so some of them might not be needed (or even recommended!) for other use cases.

Example: `minb` — a minimal bind shell

I wrote a minimal bind shell that illustrates the double-fork technique. It also performs some (not all, for a reason) of the extra steps that you should do on a simple implant like this. This example was actually the main reason for writing this post. When I gave it to some of my students, I realized many of them did not understand how to do robust daemonization.
