Linux Daemon Writing HOWTO

Devin Watson

v1.0, May 2004

This document shows how to write a daemon in Linux using GCC. Knowledge of Linux and a familiarity with C are necessary to use this document. This HOWTO is Copyright by Devin Watson, under the terms of the BSD License.

---

1. Introduction: What is a Daemon?

A daemon (or service) is a background process that is designed to run autonomously,with little or not user intervention. The Apache web server http daemon (httpd) is one such example of a daemon. It waits in the background listening on specific ports, and serves up pages or processes scripts, based on the type of request.

Creating a daemon in Linux uses a specific set of rules in a given order. Knowing how they work will help you understand how daemons operate in userland Linux, but can operate with calls to the kernel also. In fact, a few daemons interface with kernel modules that work with hardware devices, such as external controller boards, printers,and PDAs. They are one of the fundamental building blocks in Linux that give it incredible flexibility and power.

Throughout this HOWTO, a very simple daemon will be built in C. As we go along, more code will be added, showing the proper order of execution required to get a daemon up and running.

2. Getting Started

First off, you'll need the following packages installed on your Linux machine to develop daemons, specifically:

- GCC 3.2.2 or higher
- Linux Development headers and libraries

If your system does not already have these installed (not likely, but check anyway), you'll need them to develop the examples in this HOWTO. To find out what version of GCC you have installed, use:

```
gcc --version
```

3. Planning Your Daemon

3.1 What Is It Going To Do?

A daemon should do one thing, and do it well. That one thing may be as complex as managing hundreds of mailboxes on multiple domains, or as simple as writing a report and calling sendmail to mail it out to an admin.

In any case, you should have a good plan going in what the daemon should do. If it is going to interoperate with some other daemons that you may or may not be writing, this is something else to consider as well.

3.2 How Much Interaction?

Daemons should never have direct communication with a user through a terminal. In fact, a daemon shouldn't communicate directly with a user at all. All communication should pass through some sort of interface (which you may or may not have to write), which can be as complex as a GTK+ GUI, or as simple as a signal set.

4. Basic Daemon Structure

When a daemon starts up, it has to do some low-level housework to get itself ready for its real job. This involves a few steps:

- Fork off the parent process
- Change file mode mask (umask)
- Open any logs for writing
- Create a unique Session ID (SID)
- Change the current working directory to a safe place
- Close standard file descriptors
- Enter actual daemon code

4.1 Forking The Parent Process

A daemon is started either by the system itself or a user in a terminal or script. When it does start, the process is just like any other executable on the system. To make it truly autonomous, a *child process* must be created where the actual code is executed. This is known as forking, and it uses the *fork()* function:

```
pid_t pid;

/* Fork off the parent process */
pid = fork();
if (pid < 0) {
        exit(EXIT_FAILURE);
}
/* If we got a good PID, then
   we can exit the parent process. */
if (pid > 0) {
        exit(EXIT_SUCCESS);
}
```

Notice the error check right after the call to *fork()*. When writing a daemon, you will have to code as defensively as possible. In fact, a good percentage of the total code in a daemon consists of nothing but error checking.

The *fork()* function returns either the process id (PID) of the child process (not equal to zero), or -1 on failure. If the process cannot fork a child, then the daemon should terminate right here.

If the PID returned from *fork()* did succeed, the parent process must exit gracefully. This may seem strange to anyone who hasn't seen it, but by forking,

the child process continues the execution from here on out in the code.

4.2 Changing The File Mode Mask (Umask)

In order to write to any files (including logs) created by the daemon, the file mode mask (umask) must be changed to ensure that they can be written to or read from properly. This is similar to running umask from the command line, but we do it programmatically here. We can use the *umask()* function to accomplish this:

```
pid_t pid, sid;

/* Fork off the parent process */
pid = fork();
if (pid < 0) {
        /* Log failure (use syslog if possible) */
        exit(EXIT_FAILURE);
}
/* If we got a good PID, then
   we can exit the parent process. */
if (pid > 0) {
        exit(EXIT_SUCCESS);
}

/* Change the file mode mask */
umask(0);
```

By setting the umask to 0, we will have full access to the files generated by the daemon. Even if you aren't planning on using any files, it is a good idea to set the umask here anyway, just in case you will be accessing files on the filesystem.

4.3 Opening Logs For Writing

This part is optional, but it is recommended that you open a log file somewhere in the system for writing. This may be the only place you can look for debug information about your daemon.

4.4 Creating a Unique Session ID (SID)

From here, the child process must get a unique SID from the kernel in order to operate. Otherwise, the child process becomes an orphan in the system. The pid_t type, declared in the previous section, is also used to create a new SID for the child process:

```
pid_t pid, sid;

/* Fork off the parent process */
pid = fork();
if (pid < 0) {
```

```
        exit(EXIT_FAILURE);
}
/* If we got a good PID, then
   we can exit the parent process. */
if (pid > 0) {
        exit(EXIT_SUCCESS);
}

/* Change the file mode mask */
umask(0);

/* Open any logs here */

/* Create a new SID for the child process */
sid = setsid();
if (sid < 0) {
        /* Log any failure */
        exit(EXIT_FAILURE);
}
```

Again, the *setsid()* function has the same return type as *fork()*. We can apply the same error-checking routine here to see if the function created the SID for the child process.

4.5 Changing The Working Directory

The current working directory should be changed to some place that is guaranteed to always be there. Since many Linux distributions do not completely follow the Linux Filesystem Hierarchy standard, the only directory that is guaranteed to be there is the root (/). We can do this using the *chdir()* function:

```
pid_t pid, sid;

/* Fork off the parent process */
pid = fork();
if (pid < 0) {
        exit(EXIT_FAILURE);
}
/* If we got a good PID, then
   we can exit the parent process. */
if (pid > 0) {
        exit(EXIT_SUCCESS);
}

/* Change the file mode mask */
umask(0);

/* Open any logs here */

/* Create a new SID for the child process */
sid = setsid();
if (sid < 0) {
        /* Log any failure here */
        exit(EXIT_FAILURE);
}
```

```
/* Change the current working directory */
if ((chdir("/")) < 0) {
        /* Log any failure here */
        exit(EXIT_FAILURE);
}
```

Once again, you can see the defensive coding taking place. The *chdir()* function returns -1 on failure, so be sure to check for that after changing to the root directory within the daemon.

4.6 Closing Standard File Descriptors

One of the last steps in setting up a daemon is closing out the standard file descriptors (STDIN, STDOUT, STDERR). Since a daemon cannot use the terminal, these file descriptors are redundant and a potential security hazard.

The *close()* function can handle this for us:

```
pid_t pid, sid;

/* Fork off the parent process */
pid = fork();
if (pid < 0) {
        exit(EXIT_FAILURE);
}
/* If we got a good PID, then
   we can exit the parent process. */
if (pid > 0) {
        exit(EXIT_SUCCESS);
}

/* Change the file mode mask */
umask(0);

/* Open any logs here */

/* Create a new SID for the child process */
sid = setsid();
if (sid < 0) {
        /* Log any failure here */
        exit(EXIT_FAILURE);
}

/* Change the current working directory */
if ((chdir("/")) < 0) {
        /* Log any failure here */
        exit(EXIT_FAILURE);
}


/* Close out the standard file descriptors */
close(STDIN_FILENO);
close(STDOUT_FILENO);
close(STDERR_FILENO);
```

It's a good idea to stick with the constants defined for the file descriptors, for the greatest portability between system versions.

5. Writing the Daemon Code

5.1 Initialization

At this point, you have basically told Linux that you're a daemon, so now it's time to write the actual daemon code. Initialization is the first step here. Since there can be a multitude of different functions that can be called here to set up your daemon's task, I won't go too deep into here.

The big point here is that, when initializing anything in a daemon, the same defensive coding guidelines apply here. Be as verbose as possible when writing either to the syslog or your own logs. Debugging a daemon can be quite difficult when there isn't enough information available as to the status of the daemon.

5.2 The Big Loop

A daemon's main code is typically inside of an infinite loop. Technically, it isn't an infinite loop, but it is structured as one:

```
pid_t pid, sid;

/* Fork off the parent process */
pid = fork();
if (pid < 0) {
        exit(EXIT_FAILURE);
}
/* If we got a good PID, then
   we can exit the parent process. */
if (pid > 0) {
        exit(EXIT_SUCCESS);
}

/* Change the file mode mask */
umask(0);

/* Open any logs here */

/* Create a new SID for the child process */
sid = setsid();
if (sid < 0) {
        /* Log any failures here */
        exit(EXIT_FAILURE);
}


/* Change the current working directory */
if ((chdir("/")) < 0) {
        /* Log any failures here */
        exit(EXIT_FAILURE);
```

```
}

/* Close out the standard file descriptors */
close(STDIN_FILENO);
close(STDOUT_FILENO);
close(STDERR_FILENO);

/* Daemon-specific initialization goes here */

/* The Big Loop */
while (1) {
   /* Do some task here ... */
   sleep(30); /* wait 30 seconds */
}
```

This typical loop is usually a *while* loop that has an infinite terminating condition, with a call to *sleep* in there to make it run at specified intervals.

Think of it like a heartbeat: when your heart beats, it performs a few tasks, then waits until the next beat takes place. Many daemons follow this same methodology.

6. Putting It All Together

6.1 Complete Sample

Listed below is a complete sample daemon that shows all of the steps necessary for setup and execution. To run this, simply compile using gcc, and start execution from the command line. To terminate, use the *kill* command after finding its PID.

I've also put in the correct include statements for interfacing with the syslog, which is recommended at the very least for sending start/stop/pause/die log statements, in addition to using your own logs with the *fopen()*/*fwrite()*/*fclose()* function calls.

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

int main(void) {

        /* Our process ID and Session ID */
        pid_t pid, sid;

        /* Fork off the parent process */
        pid = fork();
```

```
        if (pid < 0) {
                exit(EXIT_FAILURE);
        }
        /* If we got a good PID, then
           we can exit the parent process. */
        if (pid > 0) {
                exit(EXIT_SUCCESS);
        }

        /* Change the file mode mask */
        umask(0);

        /* Open any logs here */

        /* Create a new SID for the child process */
        sid = setsid();
        if (sid < 0) {
                /* Log the failure */
                exit(EXIT_FAILURE);
        }



        /* Change the current working directory */
        if ((chdir("/")) < 0) {
                /* Log the failure */
                exit(EXIT_FAILURE);
        }

        /* Close out the standard file descriptors */
        close(STDIN_FILENO);
        close(STDOUT_FILENO);
        close(STDERR_FILENO);

        /* Daemon-specific initialization goes here */

        /* The Big Loop */
        while (1) {
           /* Do some task here ... */

           sleep(30); /* wait 30 seconds */
        }
   exit(EXIT_SUCCESS);
}
```

From here, you can use this skeleton to write your own daemons. Be sure to add in your own logging (or use the syslog facility), and code defensively, code defensively, code defensively!