Table of Contents

1
Daemon Processes

A daemon is a system background process; typically, it comes alive as part of system initialization, and it only terminates upon system shutdown (or reboot). Daemon processes take on the job of system servers. They monitor the system in some manner or provide some service to other applications. Often, and as a tradition, the name of daemon processes end with the letter *d*; for example, the well known SSH server process sshd, the (older) internet super server process inetd, the FTP server ftpd, the power management server acpid, the printer server cupsd, the modern systemd itself, and so many of its ilk—systemd-mount, systemd-run, and so on.

Interestingly, the Linux kernel folks also follow this naming tradition; several kernel threads serve as daemon-like servers, such as `kthreadd`, `khungtaskd`, and `khugepaged`.

In this chapter, the intention is to give the reader a broad overview of:

- The types of daemons one can construct
- A little on how to go construct a daemon

Types of daemons

Today, we distinguish between two types or styles or schemes of daemon processes, based on their internal architecture and the manner in which to initialize and run them. We have:

- The traditional SysV daemon process
- The new-style daemon process

In the following, we give the reader an overview of both.

The traditional SysV daemon

This is the old-style, traditional SysV Unix implementation for a daemon process.

By the way, what exactly is this SysV thing people talk about? Firstly, it's pronounced *System Five*; the *V* represents the Roman numeral five. Original Unix evolved into its modern variant (remember, all this is now pretty ancient!) called SysV Unix, the first commercial variant of Unix. It started with AT&T releasing SVR1-SysV release 1 in 1983, with the latest and greatest **SysV Rel 4** (**SVR4**) launching in October 1988.

The man page on `daemon(7)` (`http://man7.org/linux/man-pages/man7/daemon.7.html`) is extremely detailed and guides the application developer on how exactly to create both the old-style, traditional and the modern, new-style daemon process; look it up at `https://www.freedesktop.org/software/systemd/man/daemon.html`. The daemon process must fulfill certain prerequisites to qualify; to do so, we write code that will *daemonize* a process. The code will perform several steps in order to daemonize itself. They are summarized as follows:

1. Shutting down all open files from descriptor 3 onward, if they exist (shown in the following code).
2. Reset all signal handlers and the signal mask.
3. Sanitize environment variables as required.
4. `fork(2)` and have the child call `setsid(2)` to create an independent session (and detach from any controlling terminal device).
5. Within the child process, `fork(2)` again (so that the child is guaranteed to not get a terminal).
6. Have the first child die (call `exit(2)`) so that only the second child—the actual daemon process stays alive and is reparented by init (or systemd, PID 1).
7. Have the daemon set its `stdin`, `stdout`, and `stderr` to the null device, `/dev/null` (shown in the code that follows).
8. Reset the process umask, change the working directory to root (/).
9. Set up a lockfile for the daemon (typically, `/run/mydaemon.pid`) such that the daemon starts exactly once.
10. If applicable (if it's started as root or other privilege), have the daemon drop all privileges (recall our discussions in `Chapter 7`, *Process Credentials*).
11. Notify the original parent that the daemon setup is done.
12. Have the original parent process die (call `exit(2)`).

Now the daemon process is up and running. By following the steps detailed within this man page, we present a `C` function that can daemonize a process in the traditional SysV Unix manner (`ch20/daemon_trad.c`).

For the reader's convenience, we have reproduced the numbered steps required to set up a traditional-style SysV daemon process directly from the man page on `daemon(7)`. The step is mentioned in the code as a numbered comment; the actual code to implement that step follows the comment.

```
static int daemonize_traditional(void)
{
  int i, fd_null;
  struct rlimit rlim;
  struct sigaction act;
  sigset_t sigmask;

  /* 1. Close all open file descriptors except standard input, output,
   * and error (i.e. the first three file descriptors 0, 1, 2). This
   * ensures that no accidentally passed file descriptor stays around
   * in the daemon process. On Linux, this is best implemented by
   * iterating through /proc/self/fd, with a fallback of iterating
from
   * file descriptor 3 to the value returned by getrlimit()
   * for RLIMIT_NOFILE. */
  if (prlimit(0, RLIMIT_NOFILE, 0, &rlim) < 0) {
      WARN("prlimit RLIMIT_NOFILE failed\n");
      return -1;
  }
  printf("%d:%s: [+] step 1 : close fd's (3 to %ld)\n",
         getpid(), __func__, rlim.rlim_max);
  for (i = 3; i < rlim.rlim_max; i++)
      (void)close(i);
...
```

In the preceding code block, we see the code to shut down open files from descriptor 3 onward. We show a few more significant steps as follows:

```
#define SET_TO_NULDEV(orig_fd, new_fd) do {             \
  if (dup2(orig_fd, new_fd) < 0) {                      \
      WARN("dup2(" #orig_fd ", " #new_fd ") failed\n"); \
      return -9;                                        \
  }                                                     \
} while(0)

...
  /* 7. In the child, call fork() again, to ensure that the daemon
```

```
    * can never re-acquire a terminal again. */
     printf("%d:%s: [+] step 7 : second fork\n", getpid(), __func__);
     switch (fork()) {
     case -1:
       WARN("fork #2 failed!\n");
       return -7;
     case 0: // Second child; *the ACTUAL DAEMON*
       printf(" %d:%s: the ACTUAL DAEMON !\n",
              getpid(), __func__);

       /* 9. In the daemon process, connect /dev/null to
        * standard input, output, and error. */
       printf("%d:%s: [+] step 9 : connect null dev to fd's 0,1,2\n",
            getpid(), __func__);
       fd_null = open("/dev/null", O_RDWR);
       if (fd_null < 0) {
           WARN("open on null dev failed\n");
           return -9;
       }
       /* From this point on, the printf's disappear! */
       SET_TO_NULDEV(fd_null, STDIN_FILENO);
       SET_TO_NULDEV(fd_null, STDOUT_FILENO);
       SET_TO_NULDEV(fd_null, STDERR_FILENO);
       close(fd_null);
...
   default: // (original) parent
  /* 15. Call exit() in the original process. The process that invoked
   * the daemon must be able to rely on that this exit() happens after
   * initialization is complete and all external communication channels
   * are established and accessible. */
     printf("%d:%s: [+] step 15 : original parent exit when daemon is
fully initialized\n", getpid(), __func__);
     exit(EXIT_SUCCESS);
   } // first fork
```

The reader can see the complete traditional SysV daemonize code from the book's GitHub repository and try it out to see the output.

One of the issues with daemonizing a process is that debugging it now becomes difficult (no controlling tty, and thus no possibility of `printf(3)`, signals reset). Thus, it's recommended that (during development at least, if not always) you keep a fallback debug mode option (say, by passing a `-d` parameter) to have the process start up in such a way that it can be debugged and emit verbose logs. In fact, recall our coverage of `valgrind(1)` from `Chapter 6`, *Debugging Tools for Memory Issues*. One of the most common FAQs regarding Valgrind is how does one debug a daemon process with Valgrind? We provide a link in `Chapter 6`, *Debugging Tools for Memory Issues*, Valgrind's manual that addresses this very issue (with the **Valgrind-GDB** (**vgdb**) mode) in the *Further reading* section on the GitHub repository.

Examples of traditional old-style daemon processes are the inetd super server process, the well-known SSH server sshd, ftpd, acpid, cupsd, fingerd. The interested reader can certainly browse the internet for their source code.

Interestingly, the **Linux Tracing Toolkit next generation** (**LTTng**) project sets up a daemon process called lttng-sessiond daemon; it seems to be architected as an old-style daemon (one can see the code on its GitHub repository: `https://github.com/lttng/lttng-tools/blob/a503e1ef71bfe98526469205fc2956cc65954019/src/common/daemonize.c`.

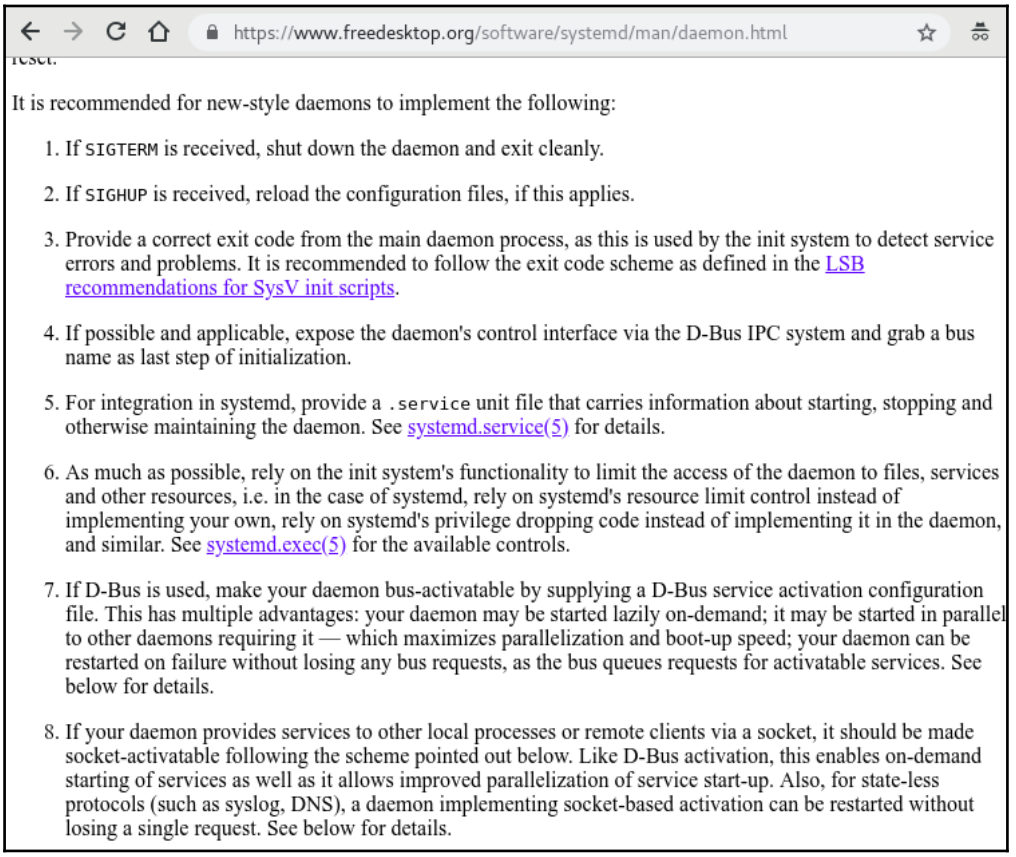
The new-style daemon

The new-style daemon process is the modern one and, obviously, the recommended style to use nowadays. It is quite heavily based and dependent upon the modern init framework, systemd.

As you might be aware, systemd is the modern init framework and service-manager system for Linux. It is far superior to the older SysV init framework in terms of both internal/external design interfaces and performance. We refer the reader to the (dozens of!) man pages on `systemd(1)` utilities, commands, and configuration files. Systemd is in itself a very large topic, which we will not delve into here; for the reader's convenience, several systemd resource page links can be found in the *Further reading* section on the GitHub repository.

One of the advantages of the new-style daemon setup is that the init framework (typically systemd) itself will guarantee that the daemon will start in a clean process context (with the environment sanitized, descriptors closed, signals reset, and so on). The programmer should not perform the initialization steps that were required for the traditional-style daemon here (it might interfere with the modern framework and is thus best avoided). Also note that the daemon process's `stdin` will be the null device, `stdout`, and `stderr` will be automatically connected to systemd's journald logging service.

As mentioned earlier, the man page on `daemon(7)` (`http://man7.org/linux/man-pages/man7/daemon.7.html`) is extremely detailed, and guides the application developer on how exactly to create both the traditional and the modern daemon process; thus, just to give the reader a flavor of it, we show a screenshot from the man page (it does not show all the steps; please refer the actual man page to see all of them):



The same man page further provides details on how to activate a new-style daemon and details on integrating the daemon process into the modern systemd framework.

Examples of modern daemon processes are `systemd(1)` (and its dozens of cousins that form the framework) and the GNOME framework daemons (`gvfsd`, `fwupd`, `gnome-keyring-daemon`).

Summary

An overview of what a daemon process is, as well as an overview of both the older, traditional SysV daemon and the modern daemon process was presented in this chapter, along with an explanation of the steps required to implement them.

The next, and final, chapter in this book will deal with something the aspiring developer must keep in mind at all times: industry best practices as well as useful troubleshooting tips.

Index