CHAPTER W21

System Programming IV: Practical Considerations

Objectives

- To explain the concept of restarting system calls
- To describe thread-safe system calls
- To discuss the concept of writing a program that starts running in the background and becomes a daemon
- To discuss setting signals and umask for a daemon
- To describe the concept of allowing only a single copy of a program to run
- To discuss saving a daemon's identity at a known location
- To explain the issue of detaching the terminal from a process
- To describe the issue of changing the working directory
- To explain the need to close inherited standard descriptors and open standard descriptors
- To describe the removal of completed child processes from the system
- To show and discuss the structure of a truly concurrent, connection-oriented production server
- To cover the system calls, library calls, commands, and primitives `chdir()`, `close()`, `fcntl()`, `flock()`, `fork()`, `free()`, `getpid()`, `goto`, `malloc()`, `open()`, `perror()`, `read()`, `setsid()`, `signal()`, `sprintf()`, `sleep()`, `strlen()`, `umask()`, `wait3()`, `write()`

W21.1 INTRODUCTION

In this chapter, we discuss some important issues related to system programming in general and the design of server processes in particular. The issues are blocking input/output (I/O), restarting system calls that are interrupted during their execution due to the arrival of a signal, using thread-safe system calls, reentrant code, ignoring signals and setting the umask of server processes, running server processes as daemons, ensuring that only one copy of a program may run at a given time, closing inherited descriptors in a process, ignoring I/O in a daemon, redirecting errors to a file, and cleaning up the child processes that have terminated. We have already discussed some of these issues in detail in Chapters 15–16 and Chapter W20. Here, we discuss some specific issues, particularly related to the design of server software. We describe two methods of file locking. Throughout the chapter, we use defensive coding.

W21.2 RESTARTING SYSTEM CALLS

We should write code that has the ability to restart system calls that are interrupted during their execution. Such situations arise under different scenarios depending on the system call. However, in all cases, the system call is in some kind of waiting state. The wait may be any one of the many types, including waiting for an I/O device to complete its input or output, a communication channel to become available for reading or writing, or a child process to terminate. The system calls involved may be `read()`, `readv()`, `write()`, `writev()`, `ioctl()`, `fcntl()`, `wait()`, `waitpid()`, `wait3()`, `wait4()`, `wait6()`, `select()`, and so on.

We discussed blocking I/O in detail in Chapters 16 and W20 while explaining the concept of a communication channel called the *Linux pipe*, which is used for communication between related processes. A `read()` or `write()` call may be interrupted when you perform a *blocking I/O* on a pipe. In the case of a pipe, blocking input means that the `read()` system call blocks, as there is nothing to read because the pipe is empty. The `write()` system call blocks if the pipe is full. Another example of blocking read is when the `read()` system call reads input interactively from a keyboard. The `read()` call blocks until the user enters keyboard input. Similarly, the `select()` system call blocks while waiting for an I/O request on a descriptor. Finally, a process may block for several other reasons too such as while waiting for a child process to terminate using any of the calls in the `wait()` class or a synchronization construct (e.g., semaphore, mutex, condition variable, or monitor) to be released.

If a system call is in a blocking state, it may be interrupted while waiting for a particular event to happen (the arrival of data into a pipe, the flow of data out of a pipe, the termination of a child process, etc.). Most modern Linux implementations restart interrupted system calls automatically. If you are not sure whether your code will run on such a system, you need to write code to explicitly handle the restarting of an interrupted system call. The following code snippet may be used for this purpose.

```
repeat:
       if ((nr = read(fd, buf, SIZE)) == -1)) {
           if(errno == EINTR) /* if interrupted system call */
              goto repeat;
           /* handle other errors */
       }
```

I/O-related calls are interrupted by a signal when they operate on slow devices or a communication channel such as a pipe. However, the calls in the `wait()` class are always interrupted when they receive a signal. By default, Linux with kernel 2.4.22 and later versions restarts system calls interrupted by signals. In Single UNIX Specification-compliant systems, interrupted system calls may be automatically restarted using the SA_RESTART flag of the `sigaction()` system call.

W21.3 THREAD-SAFE SYSTEM CALLS

In this section, we provide practical considerations that are an extension of the material presented in Chapter 16 on threads. We define, discuss, and give simple examples of two important concurrency topics: thread safety and reentrant functions.

Simply stated, a *thread-safe function* can be called simultaneously from multiple threads, even when the invocations use shared data, because all references to the shared data are serialized. In other words, each thread accesses shared data on a mutually exclusive basis after locking the data using synchronization primitives like *spinlocks* or *semaphores*.

A *reentrant function* can also be called simultaneously from multiple threads, but only if each invocation uses its own data. Therefore, a thread-safe function is always reentrant, but a reentrant function is not always thread safe. This means that with only one copy of a thread-safe or reentrant function in main memory, multiple threads or commands may execute it simultaneously. Most applications running on time-sharing systems are thread safe or reentrant, including compilers, word processors, and editors. All functions defined in the Single UNIX Specification (SUSv3) are guaranteed to be thread safe, with the exception of those listed in Table W21.1.

TABLE W21.1 Nonthread-Safe Functions

asctime()	fcvt()	getpwnam()	nl_langinfo()
basename()	ftw()	getpwuid()	ptsname()
catgets()	gcvt()	getservbyname()	putc_unlocked()
crypt()	getc_unlocked()	getservbyport()	putchar_unlocked()
ctime()	getchar_unlocked()	getservent()	putenv()
dbm_clearerr()	getdate()	getutxent()	pututxline()
dbm_close()	getenv()	getutxid()	rand()
dbm_delete()	getgrent()	getutxline()	readdir()
dbm_error()	getgrgid()	gmtime()	setenv()
dbm_fetch()	getgrnam()	hcreate()	setgrent()
dbm_firstkey()	gethostbyaddr()	hdestroy()	setkey()
dbm_nextkey()	gethostbyname()	hsearch()	setpwent()
dbm_open()	gethostent()	inet_ntoa()	setutxent()
dbm_store()	getlogin()	l64a()	strerror()
dirname()	getnetbyaddr()	lgamma()	strtok()
dlerror()	getnetbyname()	lgammaf()	ttyname()
drand48()	getnetent()	lgammal()	unsetenv()
ecvt()	getopt()	localeconv()	wcstombs()
encrypt()	getprotobyname()	localtime()	wctomb()
endgrent()	getprotobynumber()	lrand48()	
endpwent()	getprotoent()	mrand48()	
endutxent()	getpwent()	nftw()	

You should not use non-reentrant functions in signal handlers because a signal may occur while the control is in a non-reentrant function when another signal is received. Clearly, in such cases, the program may produce a wrong result because of a race condition caused by the multiple simultaneous executions of a non-reentrant function. For example, the `malloc()` function maintains a linked list of the dynamically allocated areas. We should not use `malloc()` in a signal handler because it may cause problems due to race condition. Suppose we use `malloc()` in a signal handler, a signal occurs, and control goes to the signal handler; the `malloc()` function runs as part of the signal handler and it is in the middle of updating the list pointers when the second signal occurs. The linked list will then become corrupt. Similarly, functions that access global (static) variables are also prone to being non-reentrant and nonthread safe. No system calls and no library functions are guaranteed to be reentrant that (a) use `malloc()` or `free()`, (b) use global data structures, or (c) are part of the Standard I/O library. All of the system calls we have discussed in this book are reentrant.

Operating systems and thread libraries provide synchronization primitives for writing thread-safe and reentrant functions, including semaphores and *mutexes*. We do not cover these primitives in this book but you may read more about them in books or Internet sources on pthread programming and/or Linux system programming that discuss these topics more comprehensively.

W21.4 RUNNING PROCESSES IN BACKGROUND: DAEMONS

We discussed in detail the execution of background processes and daemons in Chapter 10. A system process that provides services to users or processes is known as a *daemon*. A common use of daemons is in server processes. Examples of some commonly known daemons are the http (Web) server (httpd), Secure Shell server (sshd), Linux superserver (inetd and xinetd), printer server (lpd), and kernel thread daemon (kthreadd).

In this section, we discuss how you can write code for a process that, when executed, automatically becomes a daemon. One of the characteristics of daemons is that they are disconnected from standard files. Creating a daemon is rather simple. The issue boils down to creating a child process, letting the parent process exit, and allowing the child process to continue and become a daemon. The **daemon.c** program in the following session illustrates the concept. After the parent process exits, the child process, now a daemon, displays its process ID (PID) before entering an infinite loop to read client requests and respond to them.

```
$ cat daemon.c
#include <unistd.h>
#define SIZE 32

int main(void)
{
        pid_t pid;
        char buf[SIZE];

        pid = fork();
        if (pid == -1) {
            perror("Fork failed");
            exit(1);
        }
        /* Parent process */
        if (pid > 0) {
            exit(0);
        }
        /* Child process: Continues and becomes the server */
        /* process running forever as a daemon             */
        (void) sprintf(buf, "Daemon PID: %d\n", getpid());
        (void) write(1, buf, strlen(buf));
        while (1) {
            /* Wait for a client request */
            /* Serve the client request  */
            sleep(10);     /* Dummy code */
        }
}
$ gcc -w daemon.c -o daemon
$ ./daemon
Daemon PID: 7891
$ ps
  PID TTY          TIME CMD
 7863 pts/1    00:00:00 bash
 7891 pts/1    00:00:00 daemon
 7902 pts/1    00:00:00 ps
```

```
$ kill -9 7891
$ ps
  PID TTY          TIME CMD
 7863 pts/1    00:00:00 bash
 7907 pts/1    00:00:00 ps
$
```

Exercise W21.1

Provide the names of ten Linux daemons and their purposes.

Exercise W21.2

Compile and run the preceding **daemon.c** program to make sure it works on your system as expected.

W21.5 IGNORING SIGNALS

We discussed the issue of signals and signal handling in detail in Chapter W20. In this section, we emphasize that when you write code for a server process, you must handle signals appropriately.

Sometimes, you need to write server processes that can initialize their data structures using a text-based configuration file, and you want such servers to be able to reconfigure themselves dynamically without stopping. You can do so by sending the process a particular type of signal and invoking a function that reconfigures the server. The Linux superserver, inetd, is an example of a server process that reconfigures itself on `SIGHUP`. Thus, it contains a line of code similar to the following:

```
signal(SIGHUP, sig_hup);
```

where the `sig_hup()` function reconfigures the internal data structures of inetd. The configuration file for inetd is **/etc/inetd.conf**.

W21.6 CHANGING UMASK

It is important to set the umask in the server process so that all the files, including logs, created by the server process have the desired access privileges, regardless of the mode specified in an `open()` or `creat()` system call. You can do so by using the `umask()` system call, as follows:

```
(void) umask(027);
```

We discussed in detail the setting of the umask at the command line and how the permission bits of a newly created file are set in Chapter 5. In the following session, we show the code for **daemon.c** after including signal handling and setting the umask. The compilation and running of the program on our Linux system shows that the program works correctly and the daemon does not terminate when we send it the `SIGHUP` and `SIGINT` signals using the `kill` commands. Eventually, we terminate `daemon` using the `kill -9 7937` command.

```
$ cat daemon.c
#include <unistd.h>
#include <sys/signal.h>

#define SIZE 32

int main(void)
{
       pid_t pid;;
       char buf[SIZE];

       pid = fork();
       if (pid == -1) {
           perror("Fork failed");
           exit(1);
       }
       /* Parent process */
       if (pid > 0) {
           exit(0);
```

```
        }
        /* Child process: Continues and becomes the server */
        /* process running forever as a daemon             */

        /* Ignore signals */
        signal(SIGHUP, SIG_IGN);
        signal(SIGINT, SIG_IGN);

        /* Set umask */
        (void) umask(027);

        /* Display Daemon's PID */
        (void) sprintf(buf, "Daemon PID: %d\n", getpid());
        (void) write(1, buf, strlen(buf));

        /* Code for the server */
        while (1) {
            /* Wait for a client request */
            /* Serve the client request  */
            sleep(10);     /* Dummy code */
        }
}
$ gcc -w daemon.c -o daemon
$ daemon
Daemon PID: 7937
$ kill -0 7937
$ ps
  PID TTY          TIME CMD
 7863 pts/1    00:00:00 bash
 7937 pts/1    00:00:00 daemon
 7938 pts/1    00:00:00 ps
$ kill -2 7937
$ ps
  PID TTY          TIME CMD
 7863 pts/1    00:00:00 bash
 7937 pts/1    00:00:00 daemon
 7939 pts/1    00:00:00 ps
$ kill -9 7937
$ ps
  PID TTY          TIME CMD
 7863 pts/1    00:00:00 bash
 7945 pts/1    00:00:00 ps
$
```

Exercise W21.3

Repeat the preceding shell session to make sure that the **daemon.c** program works correctly on your Linux system.

Exercise W21.4

What services does inetd, the Linux superserver, offer on your system? List all the connection-oriented and connectionless services that the configuration file for inetd contains.

W21.7 RUNNING A SINGLE COPY OF A PROGRAM

In this section, we discuss how we can make sure that only a single copy of a process runs. Some daemons are designed so that only a single copy of the daemon runs. One of the reasons for this may be that the daemon needs exclusive access to an object such as a file or device. There may be other reasons—for example, if multiple instances of cron start running, each would run a scheduled operation. This would not only result in duplicate operations but possibly an error too. Similarly, if a server goes down and two system administrators restart it, then the end result would be unpredictable.

You can ensure that a single instance of a daemon runs by including a code snippet at the beginning of the server program that accesses a lock of some sort. The lock may be a *spin lock* (i.e., a *binary semaphore*), a lock for exclusive access of a record, or a lock for exclusive access of a file. The actual code for the service offered by the server is executed only after this lock has been acquired, ensuring that only one copy of the program executes. In Linux, daemons are usually designed to use a lock file for this purpose. The lock files for Linux daemons are located in the **/run** directory. The naming convention used for lock files is daemon_name.pid as in inetd.pid. A lock file contains the PID of the corresponding daemon. The following session shows the names of 11 daemons running on our Linux system and that the PID of inetd is 854.

```
$ ls -l /run | grep .pid | more
-rw-r--r--  1 root  root     4 Aug  5 13:49 acpid.pid
srw-rw-rw-  1 root  root     0 Aug  5 13:49 acpid.socket
-rw-r--r--  1 root  root     5 Aug  5 13:56 console-kit-daemon.pid
-rw-r--r--  1 root  root     4 Aug  5 13:50 crond.pid
-rw-r--r--  1 root  root     4 Aug  5 13:49 inetd.pid
-rw-r--r--  1 root  root     5 Aug  5 13:50 irqbalance.pid
-rw-r--r--  1 root  root     4 Aug  5 13:49 lvmetad.pid
-rw-r--r--  1 root  root     5 Aug  5 13:50 mdm.pid
-rw-r--r--  1 root  root     4 Aug 20 18:02 ntpd.pid
-rw-r--r--  1 root  root     3 Aug  5 13:49 rsyslogd.pid
-rw-r--r--  1 root  root     5 Aug 20 18:02 sshd.pid
$ more /run/inetd.pid
854
$ ps -e | grep inetd
  854 ?        00:00:00 inetd
$
```

Files may be locked for mutually exclusive access by using the `flock()` system call. Here is a brief description of the `flock()` system call:

`#include <sys/file.h>`
`int flock(int fd, int operation);`
Success: 0 **Failure:** –1 and kernel variable `errno` set to indicate the type of error

Here, `fd` is the descriptor of an open file and `operation` specifies the type of access requested. The possible values of the `operation` argument are shown in Table W21.2. The `flock()` system call may fail for the reasons listed in Table W21.3.

TABLE W21.2 Possible Values of the `operation` Parameter and Their Effect on the File Referred to by the Descriptor `fd`

Value of **`operation`**	Effect on the File Referred to by **`fd`**
`LOCK_SH`	Lock the file for shared access
`LOCK_EX`	Lock the file for mutually exclusive access
`LOCK_NB`	Do not block while locking the file (i.e., nonblocking locking)
`LOCK_UN`	Unlock the file

TABLE W21.3 Reasons for `flock()` to Fail

Reason for Failure	Value of `errno`
The `LOCK_NB` option was specified and the file was already locked.	`EWOULDBLOCK`
The `fd` argument is not a valid descriptor.	`EBADF`
The `fd` argument does not refer to a file.	`EINVAL`
The `fd` argument refers to an object that does not support file locking.	`EOPNOTSUPP`
No locks are available.	`ENOLCK`

You can use the following piece of code to lock a file for mutual exclusion. You open or create the `LOCKFILE` and lock it for exclusive access using the `flock()` system call. We use the nonblocking call so that, if the given file is already locked, the program terminates. Since we do not have the permission to write to the **/run** directory, we place the lock file in the **/home/sarwar/Servers** directory—that is, a user's home directory in Linux.

```
#include <sys/file.h>
#include <sys/stat.h>
#define LOCKFILE "/home/sarwar/Servers/testd.lock"
```

```
#define LOCKMODE (S_IRUSR|S_IWUSR|S_IWGRP|S_IROTH)
...
/* Open (or create) LOCKFILE */
lock = open(LOCKFILE, O_RDWR | O_CREAT, LOCKMODE);
if (lock < 0) {
    perror("open failed to create LOCKFILE");
    exit(1);
}
/* Lock LOCKFILE for mutually exclusive access */
if (flock(lock, LOCK_EX | LOCK_NB)) {
    perror("flock failed to obtain exclusive lock for file");
    exit(1);
}
```

The `flock()` system call is simple and portable across `fork()`. However, in Linux kernels up to 2.6.11, `flock()` may only be used to lock files on a local file system; it does not lock files over Network File System (NFS). You can use `fcntl()` byte-range locking over NFS on such Linux systems. But, since Linux 2.6.12, NFS clients support `flock()` locks.

The `fcntl()` system call may also be used for performing such operations on open files that may not be performed with other system calls, including locking files for shared or exclusive access. The `fcntl()` system call is a little harder to use because it does not hold locks across `fork()`, and the lock on a file is released when a `close()` system call is used on its descriptor. Here is a brief description of the `fcntl()` system call:

```
#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>
```

```
int fcntl(int fd, int cmd, /* arg */ ...);
```

Success: 0

Failure: –1 and kernel variable `errno` set to indicate the type of error

The `cmd` argument is the operation to be performed on the open file with descriptor `fd`. The data type, value, and use of the last argument depend on the value of the `cmd` argument. In this section, we are only interested in discussing commands that are used for file locking. There are two commands that may be used for this purpose: `F_SETLK` and `F_SETLKW`. Each command sets or clears a file segment lock according to the description of the lock given in the third argument via a pointer to a variable of type `struct flock`. The lock may be shared (read) or exclusive (write).

With `F_SETLK` as the third argument, the `fcntl()` call returns immediately with a value of –1 if a lock cannot be set. The `F_SETLKW` command is similar to the `F_SETLK` command, except that in the case of `F_SETLKW` the calling process waits (i.e., blocks) if other locks have blocked a shared or mutually exclusive lock. The process stays blocked until the request is satisfied. During this wait, if `fcntl()` is interrupted, it returns –1 and `errno` is set to `EINTR`.

The `flock` structure has at least the following fields:

```
struct flock {
   short   l_type;     /* lock operation type                    */
   short   l_whence;   /* lock base indicator                    */
   off_t   l_start;    /* starting offset from base              */
   off_t   l_len;      /* lock length in consecutive bytes;      */
                       /* l_len == 0 means until end of file     */
   int     l_sysid;    /* system ID running process holding lock */
   pid_t   l_pid;      /* process ID of process holding lock     */
   ...
}
```

The value of `l_whence` may be `SEEK_SET`, `SEEK_CUR`, or `SEEK_END`, to indicate that the relative offset `l_start` bytes is from the start of the file, the current position of the file pointer, or the EOF, respectively. The file will be locked if `l_len` and `l_start` are set to 0 each and `l_whence` is set to `SEEK_SET`. The `l_type` field is set to `F_RDLCK` for shared (read) lock and `F_WRLCK` for exclusive (write) lock.

In order for our code to be portable to all Linux systems, we have made a few changes to it. These changes are as follows.

1. Include the following header files in the already existing list of header files.

```
#include <fcntl.h>
#include <unistd.h>
#include <sys/file.h>
#include <sys/stat.h>
```

2. Create the **~/Servers** directory on your system.
3. The preprocessor directive to define `LOCKFILE` should be as follows:

```
#define LOCKFILE "/home/sarwar/Servers/testd.lock"
```

4. Use a variable of the `struct flock` type.

```
struct flock lock;
```

5. The file-locking code should be as follows:

```
/* Lock LOCKFILE for mutually exclusive access */
if (flock(lock, LOCK_EX | LOCK_NB)) {
    perror("Daemon already running");
    close(lockfd);
    exit(1);
}
```

We enhance the **daemon.c** program discussed in Section W21.6 with the code fragments discussed in this section, according to steps 1—5. The following session shows the new version of the **daemon.c** program, its compilation, and the running of the executable code on our Linux machine. Because of the lock, only one instance of the daemon may run at any given time.

```
$ cat lock_daemon.c
#include <fcntl.h>
#include <unistd.h>
#include <sys/file.h>
#include <sys/signal.h>
#include <sys/stat.h>

#define SIZE 32
#define LOCKMODE (S_IRUSR|S_IWUSR|S_IWGRP|S_IROTH)
#define LOCKFILE "/home/sarwar/Servers/testd.lock"

int main(void)
{
        pid_t pid, lockfd;
        char buf[SIZE];
        struct flock lock;

        pid = fork();
        if (pid == -1) {
            perror("Fork failed");
            exit(1);
        }
        /* Parent process */
        if (pid > 0) {
            exit(0);
        }
        /* Child process: Continues and becomes the server */
        /* process running forever as a daemon             */

        /* Ignore signals */
        signal(SIGHUP, SIG_IGN);
        signal(SIGINT, SIG_IGN);

        /* Set umask */
        (void) umask(027);

        /* Allow only a single copy of the daemon */
        /* Open (or create) LOCKFILE              */
        lockfd = open(LOCKFILE, O_RDWR | O_CREAT, LOCKMODE);
        if (lockfd == -1) {
            perror("open failed to create LOCKFILE");
            exit(1);
        }

       /* Lock LOCKFILE for mutually exclusive access */
        if (flock(lockfd, LOCK_EX | LOCK_NB)) {
            perror("Daemon already running");
            close(lockfd);
            exit(1);
        }

        (void) sprintf(buf, "Daemon PID: %d\n", getpid());
        (void) write(1, buf, strlen(buf));
        while (1) {
            /* Wait for a client request */
            /* Serve the client request  */
            sleep(10);     /* Dummy code */
        }
}
$ gcc -w lock_daemon.c -o daemon
$ daemon
Daemon PID: 16197
$ ps
  PID TTY          TIME CMD
15771 pts/3    00:00:00 bash
16197 pts/3    00:00:00 daemon
16207 pts/3    00:00:00 ps
$ daemon
Daemon already running: Resource temporarily unavailable
$ ps
  PID TTY          TIME CMD
15771 pts/3    00:00:00 bash
16197 pts/3    00:00:00 daemon
16199 pts/3    00:00:00 ps
$ kill -9 16197
$ ps
  PID TTY          TIME CMD
15771 pts/3    00:00:00 bash
16208 pts/3    00:00:00 ps
$
```

Exercise W21.5

Repeat the preceding session on your system. Does it work?

W21.8 LOCATING A DAEMON

To easily locate a daemon that is misbehaving or has crashed, system programmers normally record its PID in the lock file as soon as the daemon has been created and the relevant file has been locked for exclusive access. You may use the following code snippet after the file-locking code shown in Section W21.7 has been executed.

```
(void) sprintf(buf, "Daemon Name: %d\n", getpid());
(void) write(lock, buf, strlen(buf));
```

Exercise W21.6

Insert the preceding code in the **daemon.c** program used in Exercise 21.5. Compile and execute the program. Does it work? How do you know?

W21.9 DETACHING THE TERMINAL FROM A DAEMON

Since daemons run in the background, there is no need to have any terminal attached to them because they are used to either provide operating system services to users including system administrators without human intervention (e.g., systemd, kthreadd, cron) or communicate with other processes (clients) in a client–server model such as a Web server (httpd). You can detach the terminal from a daemon by using the `setsid()` system call. The `setsid()` system call creates a new session with the calling process as its session leader. The calling process is also the group leader of the newly created process group and has no controlling terminal. Here is a brief description of this call:

`#include <unistd.h>`
`pid_t setsid();`
Success: Process group ID of the new process group, which is the same as the PID of the caller process **Failure:** –1 and kernel variable `errno` set to indicate the type of error

The call may fail and `errno` set to `EPERM` if the caller process is already a process group leader or the process group ID of another process matches the PID of the caller process.

You may use the following piece of code to detach the terminal from a process.

```
pid_t sid;
...
if ((sid = setsid()) == -1) {
     perror("setsid failed");
     exit(1);
}
```

Exercise W21.7

Enhance the **daemon.c** program created in Exercise W21.6 with the preceding code fragment. Compile and execute the program.

W21.10 CHANGING THE CURRENT WORKING DIRECTORY

It is important to change the current working directory for a server to a known and safe directory but not your home directory. Doing so is useful for several reasons:

1. It is easy to locate the `core` file if the system administrator terminates a misbehaving server or a server crashes (aborts) for some reason.
2. If a process is running in a directory, the file system that contains this directory cannot be unmounted without first terminating the service. Thus, you as system administrator would not be able to perform any tasks that require this file system (containing your home directory) to be unmounted.

It is difficult to identify a single directory that is appropriate for all servers. For this reason, the root directory is chosen for almost all servers. You make the root directory the current working directory for your server by adding the following piece of code to your server:

```
if ((chdir("/")) == -1) {
      perror("chdir failed");
      exit(1);
}
```

Exercise W21.8

Enhance the **daemon.c** program created in Exercise W21.7 with the preceding piece of code. Compile and execute the program.

W21.11 CLOSING INHERITED STANDARD DESCRIPTORS AND OPENING STANDARD DESCRIPTORS

Since the child process, now the server process, inherits all open descriptors of its parent, it is important for the server process to close all of these open descriptors so that it does not fall short of descriptors while serving client requests. However, you first need to identify the open descriptors and then close them. If the child process does not inherit any open descriptor, then it needs to close only the standard descriptors. This is usually the case, unless you overlay the code for the child process with another executable using a call from the `exec()` family and the descriptors are not marked *close-on-exec*. You can close the standard descriptors using the following code snippet:

```
int fd;
```

```
...
for (fd = 2; fd >= 0; fd--)
if (close(fd) == -1) {
    perror("close failed");
    exit(1);
}
```

Most daemons do not explicitly deal with standard descriptors. However, many library functions assume that standard descriptors are open. Thus, to make such library calls work properly in your daemons, you should open the three standard files but attach them to a benign device. The Linux black hole, **/dev/null**, is one such device that returns EOF on a read and consumes whatever you write to it. After closing all inherited descriptors, you may use the following code fragment to open standard files and attach them all to **/dev/null**.

```
int fd;
...
if ((fd=open("/dev/null",O_RDWR)) == -1) {
      perror("open failed");
      exit(1);
}
(void) dup(fd);
(void) dup(fd);
```

Exercise W21.9
Enhance the **daemon.c** program created in Exercise W21.8 with the preceding code snippet. Compile and execute the program.

W21.12 Waiting for All Child Processes to Terminate

As discussed in Chapter W20, several Internet services such as ftp are offered via server processes that provide services to client processes through multiple *slave processes*, one per client. The slave processes are precreated and/or created dynamically by the main server process when needed. The main server process is also known as the *master server* process. Such servers are known as concurrent, connection-oriented servers. Such a server runs in an infinite loop with the code structure shown in Figure W21.1.

```
while (1) {
     wait for a client request
     create a slave process when a client request arrives
     slave handles the client request
     dispose the slave process
}
```

Figure W21.1 Structure of a concurrent server

Because such concurrent servers keep creating slave processes as client requests arrive, it is important to terminate a slave process properly and remove it from the system after it has provided its service to the client process. You can do so by using the `exit()` and `wait()` calls in tandem, in the slave and master processes, respectively. If the master server process does not remove the slave processes after they have provided their services, a large number of zombie processes will be created in the system. The problem is that, since a concurrent server has to wait for the next client request after creating a slave process, it cannot explicitly wait for a slave process to terminate by using the `wait()` system call. Fortunately, Linux signals come to our rescue!

Recall that when a process terminates, the Linux kernel sends the `SIGCHLD` signal to its parent. A concurrent server process may use this feature to intercept all `SIGCHLD` signals to remove the terminating slave processes from the system by using the code structure shown in Figure W21.2.

```
void zombie_gatherer(int);
...
int main(...)
{
   ...
   signal (SIGCHLD, zombie_gatherer);
```

```
      while(1) {
         wait for a client request
         create a slave process when a client request arrives
         slave handles the client request
      }
      ...
   }

   void zombie_gatherer(int signal)
   {
      int status;

      while (wait3(&status, WNOHANG, 0) >= 0)
         ;
   }
```

Figure W21.2 Structure of a concurrent, connection-oriented server

Remember that the `WNOHANG` option for the `wait3()` system call makes it a nonblocking call in the sense that if it does not find a child process that has performed `exit()`, it returns –1. When `wait3()` returns –1, the control returns from the `zombie_gatherer()` function to the line of code in the main function that was interrupted by `SIGCHLD`.

A server like Apache will retain a number of child processes (slaves). The server always ensures a minimum number of children (e.g., ten), so that if you only have eight, it generates two more, and it also maintains a child until a number of uses has been reached. Also, a server can start a new child process as soon as an old one is killed off rather than waiting for a new request to come in. This is more efficient in that the request will not have to wait for a child to be spawned. To make it *fail safe*, Apache also uses three to five master processes, with one acting as *leader server* and the rest as *standby servers*. When the leader crashes or is brought down for maintenance, one of the standby servers automatically takes over as leader without disrupting the running services.

Exercise W21.10
Enhance the **daemon.c** program created in Exercise W21.9 with the code to remove the children that have completed their tasks and terminated. Compile and execute the program. Since you are using the `wait3()` system call, make sure to include the **<sys/wait.h>** file in your program.

W21.13 COMPLETE SAMPLE SERVER

In this section, we demonstrate the complete server program after including in the **daemon.c** program all of the features discussed in this chapter. Since we use the `wait3()` system call in the `zombie_gatherer()` function, we need to include the **<sys/wait.h>** file in the list of existing header files. The complete server code is in the **test_server.c** file shown in the following session.

```
$ cat test_server.c
#include <fcntl.h>
#include <unistd.h>
#include <sys/file.h>
#include <sys/signal.h>
#include <sys/stat.h>
#include <sys/wait.h>

#define SIZE 32
#define LOCKMODE (S_IRUSR|S_IWUSR|S_IWGRP|S_IROTH)
#define LOCKFILE "/home/sarwar/Servers/testd.lock"

void zombie_gatherer(int);

int main(void)
{
        int fd, lockfd;
        pid_t pid, sid;
        char buf[SIZE];
        struct flock lock;

        pid = fork();
```

```
 if (pid == -1) {
     perror("fork failed");
     exit(1);
 }
 /* Parent process */
 if (pid > 0) {
     exit(0);
 }
 /* Child process: Continues and becomes the server */
 /* process running forever as a daemon             */

 /* Ignore signals */
 signal(SIGHUP, SIG_IGN);
 signal(SIGINT, SIG_IGN);

 /* Intercept SIGCHLD and cleanup the terminating child/Children */
 signal (SIGCHLD, zombie_gatherer);

 /* Set umask */
 (void) umask(027);

 /* Allow only a single copy of the daemon */
 /* Open (or create) LOCKFILE              */
 lockfd = open(LOCKFILE, O_RDWR | O_CREAT, LOCKMODE);
 if (lockfd == -1) {
     perror("open failed to create LOCKFILE");
     exit(1);
 }

/* Lock LOCKFILE for mutually exclusive access */
 if (flock(lockfd, LOCK_EX | LOCK_NB)) {
     perror("Daemon already running");
     close(lockfd);
     exit(1);
 }

 /* Save daemon PID in the lock file */
 (void) sprintf(buf, "testd PID: %d\n", getpid());
 (void) write(lockfd, buf, strlen(buf));

 /* Detach terminal from the daemon
 if ((sid = setsid()) == -1) {
     perror("setsid failed");
     exit(1);
 }

 /* Change working directory */
 if ((chdir("/")) == -1) {
     perror("chdir failed");
     exit(1);
 }

 /* Close inherited standard descriptors */
 for (fd = 2; fd >= 0; fd--)
 if (close(fd) == -1) {
     perror("close failed");
     exit(1);
 }

 /* Open standard descriptors */
 if ((fd=open("/dev/null",O_RDWR)) == -1) {
     perror("open failed");
     exit(1);
 }
 (void) dup(fd);
 (void) dup(fd);

 /* Main server loop */
 while (1) {
     /* Wait for a client request */
     /* Serve the client request  */
     sleep(10);     /* Dummy code */
```

```
        }
}

void zombie_gatherer(int signal)
{
        int status;

        while (wait3(&status, WNOHANG, 0) >= 0)
                ;
}
$
```

In the following session, we compile and run this program on our Linux machine to show how the daemon works as coded. The executable code is saved in the **testd** file. We run this daemon with the `testd` command. The output of the `ps` command shows that the PID of the daemon is 16495. The outputs of the `ls -l /home/sarwar/Servers` shows that the **testd.lock** file is in fact created and output of the `more /home/sarwar/Servers/testd.lock` command shows that the **testd.lock** file contains the PID of the daemon. When we try to rerun the daemon, we get an error message that the daemon is already running, confirming that, as programmed, only one instance of the daemon may run at any given time. We remove the daemon from the system with the `kill -9 16495` command. If you don't remove the daemon, it will continue to run even after you have logged out because it ignores the `SIGHUP` signal.

```
$ gcc -w test_server.c -o testd
$ testd
$ ps
  PID TTY          TIME CMD
15771 pts/3    00:00:00 bash
16495 pts/3    00:00:00 testd
16496 pts/3    00:00:00 ps
$ more ~/Servers/testd.lock
testd PID: 16495
$ ls -l ~/Servers
total 4
-rw------- 1 sarwar faculty 17 Aug 21 15:29 testd.lock
$ testd
Daemon already running: Resource temporarily unavailable
$ ps
  PID TTY          TIME CMD
15771 pts/3    00:00:00 bash
16495 pts/3    00:00:00 testd
16509 pts/3    00:00:00 ps
$ kill -9 16495
$ ps
  PID TTY          TIME CMD
15771 pts/3    00:00:00 bash
16512 pts/3    00:00:00 ps
$
```

Exercise W21.11

Compile and execute the **test_server.c** program discussed in this section. Does the program work on your system as expected? Report any error that your program generates and give reasons for the errors.

W21.14 STRUCTURE OF A PRODUCTION SERVER

A production server must handle all of the issues discussed in this chapter. The structure of a concurrent, connection-oriented production server with true process-level concurrency is shown in Figure W21.3.

1. Create a child process
2. Terminate the parent process, making the child execute in the background as a daemon
3. Set signals according to the requirements of the server (ignored, default action

taken, or programmer-defined action taken)

4. Set the umask
5. Open a file for mutually exclusive access to ensure that only one copy of the daemon may run at a given time; such a file is known as the lock file
6. Save the PID of the daemon in the lock file for quick identification of the daemon/server
7. Detach the terminal from the daemon
8. Change current working directory
9. Close inherited standard descriptors
10. Open the standard files and attach them to a benign device such as /dev/null
11. Execute the true server code using an infinite loop as shown later

```
while (1) {   /* Master server process */
    a. Wait for a client request
    b. Accept the client request
    c. Create a slave process, hand over the client request to the slave process,
       and go back to wait for another client request
    d. Slave process serves the client request(s) and exits
    e. Remove from the system the slave process that has completed its work
}
```

Figure W21.3 Structure of a truly concurrent, connection-oriented production server

W21.15 WEB RESOURCES

Table W21.4 lists useful Web sites for Linux system programming and related topics.

TABLE W21.4 Web Resources for Practical Consideration in Server Design

https://www.linux.org/ https://www.linuxmint.com/ https://www.ubuntu.com/ https://www.kernel.org/ https://www.redhat.com/en https://www.linuxfoundation.org/	Some useful home pages for Linux, its different flavors, Linux kernel archives, and use of Linux to build sustainable open-source ecosystems.
https://gavv.github.io/blog/file-locks/	An excellent page on file locking in Linux.
https://www.netbsd.org/docs/guide/en/chap-inetd.html https://www.freebsd.org/doc/handbook/network-inetd.html https://en.wikipedia.org/wiki/Xinetd https://linux.die.net/man/8/xinetd http://www.omnisecu.com/gnu-linux/redhat-certified-engineer-rhce/linux-xinetd-super-server-daemon.php	Excellent pages on the UNIX superservers inetd and xinetd.

SUMMARY

We discussed a number of important topics, primarily related to server design. We discussed the issue of interrupted system calls, the system calls that may be interrupted and the circumstances under which these interruptions may occur, and restarting interrupted system calls. We then discussed thread-safe and reentrant functions, their importance, and what may cause a function to be nonthread safe and non-reentrant.

We then discussed the various issues important for the design of a production server. These issues are: creating a daemon, setting signals and umask, ensuring that only a single copy of a process may run at a given time, saving the PID of the daemon at a known place, detaching the terminal from the daemon, changing the working directory, closing inherited standard descriptors, and opening standard descriptors and attaching them to a benign device; finally, we presented a simple server that has all of these features. Along the way, we discussed two methods of file locking using the `flock()` and `fcntl()` system calls. At the end, we outlined the steps necessary for the creation of a concurrent, connection-oriented production server that uses slave processes to serve clients.

Throughout the chapter, we showed the uses of the various system calls and library functions in small C code fragments and programs to illustrate different issues related to the design of Linux servers. Finally, we combined all the code fragments to get the final version of a full-fledged test server for Linux.

Questions and Problems

1. When does the issue of restarting a system call arise?
2. What is a thread-safe system call? What are the issues that make a system call not thread safe.
3. What is reentrant code? Why is this property important in a program? Give examples of a few common applications or tools that are reentrant.
4. What are the implications of reentrant code from the point of view of memory management on a time-sharing system? Give a small example to illustrate your answer.
5. Which of the following functions are thread safe and/or reentrant? Explain your answers.

a.
```
int test;
void swap(int *first, int *b)
{
    test = *first;
    *first = *b;
    *b = test;
}
```

b.
```
int test;
void swap(int *first, int *b)
{
    int q;

    q = test;
    test = *first;
    *first = *b;
    *b = test;
    test = q;
}
```

c.
```
int sec_var = 1;

int first()
{
    sec_var = sec_var + 2;
    return sec_var;
}
```

d.
```
int sec()
{
    return first() + 2;
}
```

e.
```
int first(int i)
{
    return i + 2;
}
```

f.
```
int sec(int i)
{
    return first(i) + 2;
}
```

6. What is a daemon? Name five daemons in a typical Linux system. Write down the purpose of each.
7. Why should a server close all inherited descriptors?
8. Why does a server process open the standard descriptors after closing all inherited descriptors? Why are the newly opened standard descriptors attached to a device like **/dev/null**?
9. What kind of locks can be placed on a file using the `flock()` system call? What is the purpose of each lock?
10. Suppose a process has locked a file **ABC** for exclusive access using the `flock()` system call. The process forks a child and the child unlocks the file. Will the parent still have exclusive access to file? Explain your answer.
11. Why should a daemon be coded such that only one instance of it may execute at any given time? What is the most common method of doing so? Write a small server program in C that uses this method. Compile and run the program, and show that your method works.
12. Consider the code shown in Section W21.7 for locking a file for mutually exclusive access by a process. Change

```
#define LOCKFILE "/home/sarwar/Servers/testd.lock"
```

to

```
#define LOCKFILE "~/Servers/testd.lock"
```

Make a program out of the code with this change. Compile and run the program. Does the program work as expected? If it produces any errors, identify the buggy statement in the program, correct it, and show that the corrected version works properly.

13. The `flock()` and `fcntl()` system calls may be used to lock files for mutually exclusive access. Which of these locks may be passed on to children? Which are supported on most Linux platforms?
14. Write a connection-triggered server that becomes a daemon and provides the date service; further, it allows only a single copy of the daemon to run. Test run the server with a client and show the complete session that captures the running of the server and client. Test the client–server model by running it on the same machine and on different machines.
15. In some NIX systems, the signal handler is established again after the signal has occurred. Why? On such systems, what would happen if:
    a. The signal handler is not reestablished after the signal has occurred?
    b. The signal occurs before the signal handler is reestablished?
16. The `malloc()` and `free()` functions are not reentrant. What do you think are the reasons?
17. What is the purpose of the `WNOHANG` option in the `wait3()` system call?

Advanced Questions and Problems

18. What is a slow system call? Give five examples of such calls.
19. A call related to disk I/O is not called slow. Why?
20. What is the purpose of the kthreadd process in Linux? What is its PID?
21. Give the command to determine the current number of children of kthreadd.

21. What are the functions of the following Linux daemons: ksoftirqd, kworker, rcu_sched, rcu_bh, migration, and watchdog. What are their PIDs?