Elegant C Programming

J. Sventek

March 11, 2022

Contents

Chapter 1

Linux System Calls

This chapter concentrates on the lowest level of interaction with the Linux operating system - the system calls. These are the entries to the kernel. They *are* the facilities that the operating system provides; everything else is built on top of them.

We will cover several major areas. First is the I/O system, the foundation beneath library routines like `fopen` and `fgets`. We'll talk more about the file system as well, particularly directories and inodes. Next comes a discussion of processes - how to run programs from within a program. After that, we will talk about signals and interrupts - what happens when you type *ctl-c*, and how to handle that sensibly in a program.

Many of our examples are useful programs that are not part of the Linux distribution. Even if they are not directly helpful to you, you should learn something from reading them, and they might suggest similar tools that you could build for your system.

Full details on the system calls are in Section 2 of the *Linux Programmer's Manual*; this chapter describes some of the most important parts, but makes no pretense of completeness.

1.1 Low-level I/O

The lowest level of I/O is a direct entry into the operating system. Your program reads or writes files in chunks of any convenient size. The kernel buffers your data into chunks that match the peripheral devices, and schedules operations on the devices to optimize their performance over all users.

1.1.1 File descriptors

All input and output is done by reading or writing files, because all peripheral devices, even your terminal, are files in the file system.

In the most general case, before reading or writing a file, it is necessary to inform the system of your intent to do so, a process called *opening* the file. If you are going to write on a file, it may also be necessary to *create* it. The systems checks your right to do so (Does the file exist? Do you have permission to access it?), and if all is well, returns a non-negative integer called a *file descriptor*. Whenever I/O is to be done on the file, the file descriptor is used instead of the name to identify the file. All information about an open file is maintained by the system; your program refers to the file only by the file descriptor. A `FILE` pointer, as defined in `<stdio.h>`, points to a structure that contains, among other things, the file descriptor; the macro `fileno(fp)`, defined in `<stdio.h>`, returns the file descriptor associated with a `FILE` pointer.

There are special arrangements to make terminal input and output convenient. When it is started by the shell, a program inherits three open files, with file descriptors 0, 1, and 2, called the standard input, the standard output, and the standard error output. All of these are by default connected to the terminal, so if a program only reads file descriptor 0 and writes file descriptors 1 and 2, it can do I/O without having to open files. If the program opens any other files, they will have file descriptors 3, 4, etc.

If I/O is redirected to or from files or pipes, the shell changes the default assignments for file descriptors 0 and 1 from the terminal to the named files. Normally file descriptor 2 remains attached to the terminal, so error messages can go there. Shell syntax such as `2>filename` and `2>&1` can be used to change standard error from the default. Note that any redirection of standard input, standard output, or standard error is done by the shell, not the program. (The program itself can further rearrange the standard file descriptors, but this is rare.)

1.1.2 File I/O - `read` and `write`

All input and output is done by two system calls, `read` and `write`, which are accessed from C by functions with the same name. For both, the first argument is a file descriptor. The second argument is an array of bytes that serves as the data source or destination. The third argument is the number of bytes to be transferred.

```
int fd, n, nread, nwritten;
char buf[SIZE];

nread = read(fd, buf, SIZE);
nwritten = write(fd, buf, n);
```

Each call returns a count of the number of bytes transferred. On reading, the number of bytes returned may be less than the number requested, because fewer than `n` bytes remained to be read. (When the file is to a terminal, `read` normally reads only up to the next newline, which is usually less than what was requested.) A return value of `0` implies end of file, and `-1` indicates an error of some sort. For writing, the value returned is the number of bytes actually written; an error has occurred if this isn't equal to the number requested to be written.

Table 1.1: Time (user+system, in seconds)

SIZE	Cygwin	Ubuntu SMP	Linux VM
1	61.58	40.92	12.37
10	6.39	4.22	1.26
100	0.63	0.50	0.14
512	0.14	0.14	0.02
1024	0.08	0.07	0.01
5120	0.03	0.04	0.00

While the number of bytes to be read or written is not restricted, the two most common values are `1`, which means one character at a time ("unbuffered"), and the size of a block on a disc, most often `512` or `1024` bytes. (See the value of `BUFSIZ` in `<stdio.h>`.)

To illustrate, here is a program to copy its input to its output. Since the input and output can be redirected to any file or device, it will actually copy anything to anything - it's a bare-bones implementation of `cat`.

```
/* cat: minimal version */

#include <unistd.h>     /* for read() and write() */
#include <stdlib.h>/* for EXIT_SUCCESS */

#define UNUSED __attribute__((unused))
#define SIZE 512        /* arbitrary */

int main(UNUSED int argc, UNUSED char *argv[]) {
    char buf[SIZE];
    int n;

    while ((n = read(0, buf, sizeof buf)) > 0)
        write(1, buf, n);
    return EXIT_SUCCESS;
}
```

If the file size is not a multiple of `SIZE`, some `read` will return a smaller number of bytes to be written by `write`; the next call to `read` after that will return zero.

Reading and writing in chunks that match the disc will be most efficient, but even character-at-a-time I/O is feasible for modest amounts of data, because the kernel buffers your data; the main cost is the system calls. We timed this version of `cat` on a file of `44.1` MBytes, for several values of `SIZE` on several different types of systems [1]:

It is quite legal for several processes to be accessing the same file at the same time; indeed, one process can be writing while another is reading. If this isn't what you wanted, it can be disconcerting, but it's sometimes useful. Even though one call to `read` returns 0

[1]Cygwin: a 64-bit Windows 10 system; Ubuntu SMP: a 64-bit AMD 8-core Ubuntu system; Linux VM: a 64-bit Arch Linux VM running under VirtualBox on Windows 10

and thus signals end of file, if more data is written on that file, a subsequent `read` will find more bytes available. This observation is the basis of a program called `readslow`, which continues to read its input, regardless of whether it received an end of file or not. `readslow` is handy for watching the progress of a program:

```
$ slowprog >temp &
[1] 5213                 Process-id
$ ./readslow <temp | grep something
```

In other words, a slow program produces output in a file; `readslow`, perhaps in collaboration with some other programs, watches the data accumulate.

Structurally, `readslow` is identical to `cat` except that it loops instead of quitting when it encounters the current end of input. It has to use low-level I/O because the standard library routines continue to report `EOF` after the first end of file.

```
/* readslow: keep reading, waiting for more */

#include <unistd.h> /* for read(), write(), sleep() */

#define UNUSED __attribute__((unused))
#define SIZE 512    /* arbitrary */

int main(UNUSED int argc, UNUSED char *argv[]) {
    char buf[SIZE];
    int n;

    for (;;) {
        while ((n = read(0, buf, sizeof buf)) > 0)
            write(1, buf, n);
        sleep(10);
    }
}
```

The function `sleep` causes the program to be suspended for the specified number of seconds; it is described in `sleep(3)`. We don't want `readslow` to hammer away at the file continuously looking for more data; that would be very costly in CPU time. Thus this version of `readslow` copies its input up to the end of file, sleeps a while, then tries again. If more data arrives while it is asleep, it will be read by the next `read`. Note that `readslow` never returns of its own volition; you must kill the process, either by typing *ctl-c* to the terminal window, or by invoking the `kill` command to the shell with the processid of the process running `readslow`.

Exercise 1-1. Add a `-n` argument to `readslow` so the default sleep time can be changed to `n` seconds. `tail(1)` provides an option `-f` ("follow") that combines the functions of `tail` with those of `readslow`. Comment on this design. □

Exercise 1-2. What happens to `readslow` if the file being read is truncated? How would you fix it? Hint: read about `fstat` in Section 1.3. □

1.1.3 File creation - `open`, `creat`, `close`, and `unlink`

Other than the default standard input, output and error files, you must explicitly open files in order to read or write them. There are two system calls for this, `open` and `creat`.[2]

`open` is rather like `fopen` in `<stdio.h>`, except that instead of returning a file pointer, it returns a file descriptor, which is an `int`.

```
char *name;
int fd, rwmode;

fd = open(name, rwmode);
```

As with `fopen`, the `name` argument is a character string containing the filename. The access mode argument is different, however: `rwmode` is 0 for read, 1 for write, and 2 to open a file for both reading *and* writing. `open` returns `-1` if any error occurs; otherwise it returns a valid file descriptor.

It is an error to try to `open` a file that does not exist. The system call `creat` is provided to create new files, or to rewrite old ones.

```
int perms;

fd = creat(name, perms);
```

`creat` returns a file descriptor if it was able to create the file called `name`, and `-1` if not. If the file does not exist, `creat` creates it with the *permissions* specified by the `perms` argument. If the file already exists, `creat` will truncate it to zero length; it is not an error to `creat` a file that already exists. (The permissions will not be changed.) Regardless of `perms`, a `creat`ed file is open for writing.

There are nine bits of protection information associated with a file, controlling read, write and execute permission, so a 3-digit octal number is convenient for specifying them. For example, `0755` specifies read, write and execute permission for the owner, and read and execute permission for members of the group and everyone else. Don't forget the leading `0`, which is how octal numbers are specified in C.

To illustrate, here is a simplified version of `cp`. The main simplification is that our version copies only one file, and does not permit the second argument to be a directory. Another blemish is that our version does not preserve the permissions of the source file; we will show how to remedy this later.

[2] Ken Thompson was once asked what he would do differently if he were redesigning the UNIX system. His reply: "I'd spell `creat` with an `e`."

```c
/* cp: minimal version */

#include <stdio.h>  /* for BUFSIZ, fprintf() */
#include <unistd.h> /* for read(), write(), close() */
#include <stdlib.h> /* for EXIT_SUCCESS, EXIT_FAILURE, and exit() */
#include <fcntl.h>  /* for open(), creat() */
#include <errno.h>  /* for errno, sys_nerr, sys_errlist[] */

#define PERMS 0644  /* RW for owner, R for group, others */

char *progname = NULL;

void error(char *s1, char *s2);

int main(int argc, char *argv[]) {
    int f1, f2, n;
    char buf[BUFSIZ];

    progname = argv[0];
    if (argc != 3)
        error("Usage: %s from to", progname);
    if ((f1 = open(argv[1], 0)) == -1)
        error("can't open %s", argv[1]);
    if ((f2 = creat(argv[2], PERMS)) == -1)
        error("can't create %s", argv[2]);
    while ((n = read(f1, buf, BUFSIZ)) > 0)
        if (write(f2, buf, n) != n)
            error("write error", NULL);
    return EXIT_SUCCESS;
}
```

We will discuss `error` in the next sub-section.

There is a limit (typically 4096-8192; look for `NOFILE` in `<sys/param.h>`) on the number of files that a program may have open simultaneously. Accordingly, any program that intends to process many files must be prepared to reuse file descriptors. The system call `close` breaks the connection between a filename and the file descriptor, freeing the file descriptor for use with some other file. Termination of a program via `exit` or return from `main()` closes all open files.

The system call `unlink` removes a filename from a directory. If that was the last link to the actual file, it is removed from the file system.

1.1.4 Error processing - `errno`

The system calls discussed in this section, and in fact all system calls, can incur errors. Usually they indicate an error by returning a value of `-1`. Sometimes it is nice to know what specific error occurred; for this purpose, all system calls, when appropriate, leave an error number in an external integer called `errno`. The legal mnemonics for error numbers are listed in the `errno(3)` manual page. By using `errno`, your program can, for example, determine whether an attempt to open a file failed because it did not exist or because you lacked permission to read it. This is also an array of character strings, `sys_errlist`, indexed by `errno`, that translates each number into a meaningful string. Our version of `error` uses these data structures:

```
void error(char *s1, char *s2) {
    int errsav = errno;

    if (progname != NULL)
        fprintf(stderr, "%s: ", progname);
    fprintf(stderr, s1, s2);
    if (errsav > 0 && errsav < sys_nerr)
        fprintf(stderr, " (%s)", sys_errlist[errsav]);
    fprintf(stderr, "\n");
    exit(EXIT_FAILURE);
}
```

`errno` is initially zero, and should always be less than `sys_nerr`. It is not reset to zero when things go well, however, so you must reset it after each error if your program intends to continue.

Here is how error messages appear with this version of `cp`:

```
$ ./cp foo bar
./cp: can't open foo (No such file or directory)
$ date >foo; chmod a-r foo                Make an unreadable file
$ ./cp foo bar
./cp: can't open foo (Permission denied)
$
```

1.1.5 Random access - `lseek`

File I/O is normally sequential - each `read` or `write` takes place in the file right after the previous one. When necessary, however, a file can be read or written in an arbitrary order. The system call `lseek` provides a way to move around in a file without actually reading or writing:

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

int fd, origin;
off_t offset, pos;

pos = lseek(fd, offset, origin);
```

forces the current position in the file whose descriptor is `fd` to move to position `offset`, which is taken relative to the location specified by `origin`. Subsequent reading or writing will begin at that position. `origin` can take on the values `SEEK_SET`, `SEEK_CUR`, or `SEEK_END` to specify that the offset is to be measured from the beginning, the current position, or the end of file, respectively. The value returned is the new offset from the beginning of the file, or `(off_t) -1` for an error. For example, to append to a file, seek to the end before writing:

```
lseek(fd, 0, SEEK_END);
```

To get back to the beginning ("rewind"):

```
lseek(fd, 0, SEEK_SET);
```

To determine the current position:

```
pos = lseek(fd, 0, SEEK_CUR);
```

With `lseek`, it is possible to treat files more or less like large arrays, at the price of slower access. For example, the following function reads any number of bytes from anywhere in a file.

```
/* read n bytes from position pos */
int get(int fd, off_t pos, char *buf, int n) {
    if (lseek(fd, pos, SEEK_SET) == -1)
        return -1;
    else
        return read(fd, buf, n);
}
```

Exercise 1-3. Modify `readslow` to handle a filename argument if one is present. Add a `-e` option

```
$ ./readslow -e
```

to seek to the end of input before beginning. What does `lseek` do on a pipe? □

1.2 File system: directories

The next topic is how to walk through the directory hierarchy. While Linux provides system calls for such perambulation, all of the manual pages strongly advise using the POSIX-conforming C library interface, such as `opendir(3)`, `readdir(3)`, and `closedir(3)`. As a result, we will restrict ourselves to using the POSIX-conforming interface.[3]

[3]The interested reader who wishes to focus on the Linux system calls should consult `getdents(2)` and `getdents64(2)`; note that there are no glibc wrappers for these system calls, which means that you must invoke them using `syscall(2)`.

First, let's write a simple program to list the contents of one or more directories, `simple-ls`:

```
/* simple-ls: list contents of a directroy, one per line - version 1 */

#include <sys/types.h> /* needed by dirent.h below */
#include <dirent.h> /* for DIR, struct dirent, opendir(), readdir(),
                       closedir() */
#include <errno.h>  /* for errno, sys_errlist[] */
#include <stdio.h>  /* for stderr, fprintf(), printf() */
#include <string.h> /* for strlen() */
#include <stdlib.h> /* for EXIT_SUCCESS */

int main(int argc, char *argv[]) {
    int i;
    DIR *dd;
    struct dirent *dent;

    if (argc == 1) {      /* list current working directory */
        argc = 2;
        argv[1] = ".";
    }
    for (i = 1; i < argc; i++) {
        int n;
        char *sepstr;

        if ((dd = opendir(argv[i])) == NULL) {
            fprintf(stderr, "Error opening directory %s (%s)\n",
                    argv[i], sys_errlist[errno]);
            continue;
        }
        n = strlen(argv[i]) - 1;
        sepstr = (argv[i][n] == '/') ? "" : "/";
        while ((dent = readdir(dd)) != NULL)
            printf("%s%s%s\n", argv[i], sepstr, dent->d_name);
        closedir(dd);
    }
    return EXIT_SUCCESS;
}
```

`simple-ls` iterates over the supplied arguments which are directories. For each directory, we open it (using `opendir`), read each element in that open directory (using `readdir`), and close the directory (using `closedir`). To use the directory calls, one must include `<sys/types.h>` and `<dirent.h>`. We did not use `error` from Section 1.1, since we did not want to exit from the program if we were unable to open a directory; therefore, we included `<errno.h>`, and simply index into `sys_errlist[]` using `errno` to notify the user of failure to open the directory.

The code also illustrates declaration of variables within a block of statements (`n`, and `sepstr`), since these are only required in the body of the `for` loop. Finally, we have used the tertiary operator in C (`cond ? if_true : if_false`) to assign a value to `sepstr` based upon the last character in the directory name. `opendir` does not mind if a directory name has a trailing `/` or not, so `simple-ls` should not mind, either.

Let's illustrate use of these functions by writing a function `spname` that tries to cope with misspelled filenames. The function

```
n = spname(name, newname);
```

searches for a file with a name "close enough" to `name`. If one is found, it is copied into `newname`. The value `n` returned by `spname` is `-1` if nothing close enough was found, `0` if there was an exact match, and `1` if a correction was made.

`spname` is a convenient addition to any interactive program that solicits file names from the user; if the user misspells the name, the program can ask the user if they really meant something else, as in:

```
$ foo /urs/srx/ccmd/foo/spnam.c   Horribly botched name
"/usr/src/cmd/foo/spname.c"? y    Suggested correction accepted
* * *                             Whatever foo does with that filename
```

As we will write it, `spname` will try to correct, in each component of the filename, mismatches in which a single letter has been dropped or added, or a single letter is wrong, or a pair of letters exchanged; all of these are illustrated above. This is a boon for sloppy typists.

The operation of `spname` is straightforward enough, although there are a lot of boundary conditions to get right. Suppose the file name is `/d1/d2/f`. The basic idea is to peel off the first component (`/`), then search that directory for a name close to the next component (`d1`), then search that directory for something near `d2`, and so on, until a match has been found for each component. If at any stage there isn't a plausible candidate in the directory, the search is abandoned.

We have divided the job into three functions. `spname` itself isolates the components of the path and builds them into a "best match so far" filename. It calls `mindist`, which searches a given directory for the file that is closest to the current guess, using a third function, `spdist`, to compute the distance between two names.

```
/* spname: return correctly spelled filename */
/*
 * int spname(char *oldname, char newname[]);
 *  returns -1 if no reasonable match to oldname,
 *           0 if exact match,
 *           1 if corrected.
 *  stores corrected name in newname
 */

#include <sys/types.h> /* needed by dirent.h below */
#include <dirent.h>    /* needed for DIR, struct dirent, opendir(),
                          readdir(), closedir() */
#include <string.h>    /* needed for strcmp() */

#define DIRSIZE 256    /* size of d_name in a struct dirent */

int mindist(char *dir, char *guess, char best[]);
int spdist(char *s, char *t);

int spname(char *oldname, char newname[]) {
    char *p, guess[DIRSIZE], best[DIRSIZE];
    char *new = newname, *old = oldname;

    for (;;) {
        while (*old == '/')
            *new++ = *old++;
        *new = '\0';
        if (*old == '\0')       /* exact or corrected */
            return strcmp(oldname, newname) != 0;
        p = guess;              /* copy next component into guess */
        for ( ; *old != '/' && *old != '\0'; old++)
            if (p < guess+DIRSIZE)
                *p++ = *old;
        *p = '\0';
        if (mindist(newname, guess, best) >= 3)
            return -1;          /* hopeless */
        for (p = best; *new = *p++; )   /* add to end of newname */
            new++;
    }
}
```

```
int mindist(char *dir, char *guess, char best[]) {
    /* set best, return distance 0..3 */
    int d, nd;
    DIR *dd;
    struct dirent *dent;

    if (dir[0] == '\0')                 /* current directory */
        dir = ".";
    d = 3;                              /* minimum distance */
    if ((dd = opendir(dir)) == NULL)
        return d;
    while ((dent = readdir(dd)) != NULL) {
        nd = spdist(dent->d_name, guess);
        if (nd <= d && nd != 3) {
            strcpy(best, dent->d_name);
            d = nd;
            if (d == 0)                 /* exact match */
                break;
        }
    }
    closedir(dd);
    return d;
}
```

If the directory name given to `mindist` is empty, '.' is searched. Note that the distance test is

```
if (nd <= d ...)
```

instead of

```
if (nd < d ...)
```

so that any other single character file name is a better match than '.', which is always the first entry in a directory.

```c
/* spdist: return distance between two names */
/*
 * very rough spelling metric:
 * 0 if the strings are identical
 * 1 if two chars are transposed
 * 2 if one char wrong, added, or deleted
 * 3 otherwise
 */

#define EQ(s,t) (strcmp(s,t) == 0)

int spdist(char *s, char *t) {
    while (*s++ == *t)
        if (*t++ == '\0')
            return 0;                   /* exact match */
    if (*--s) {
        if (*t) {
            if (s[1] && t[1] && *s == t[1] &&
                *t == s[1] && EQ(s+2, t+2))
                return 1;               /* transposition */
            if (EQ(s+1, t+1))
                return 2;               /* 1 char mismatch */
        }
        if (EQ(s+1, t))
            return 2;                   /* extra character */
    }
    if (*t && EQ(s, t+1))
        return 2;                       /* missing character */
    return 3;
}
```

Once we have `spname`, integrating spelling correction into `foo` (our interactive program) is easy:

```c
#include <stdio.h>  /* for fopen() and fgets() */
#include <stdlib.h> /* for EXIT_FAILURE and exit() */

int ttyin(void) {
    char buf[BUFSIZ];
    static FILE *tty = NULL;

    if (tty == NULL)
        tty = fopen("/dev/tty", "r");
    if (fgets(buf, BUFSIZ, tty) == NULL || buf[0] == 'q')
        exit(EXIT_FAILURE);
    else
        return buf[0];
}

int main(int argc, char *argv[]) {
```

```
    int i, start;
    char buf[BUFSIZ];
    FILE *fp;

/* whatever code needed to get to file name arguments */
    for (i = start; i < argc; i++) {
        switch (spname(argv[i], buf)) {
          case -1: /* no match possible */
                   /* error message */
                   break;
          case 1:  /* corrected */
                   fprintf(stderr, "\"%s\"? ", buf); fflush(stderr);
                   if (ttyin() == 'n')
                       break;
                   argv[i] = buf;
                   /* fall through to the next case */
          case 0:  fp = fopen(argv[i], "r");
                   /* process fp */
                   fclose(fp);
        }
    }
    return EXIT_SUCCESS;
}
```

Spelling correction is not something to be blindly applied to every program that uses filenames; it's not at all suitable for programs that are not interactive.

Exercise 1-4. How much can you improve on the heuristic for selecting the best match in `spname`? For example, it is foolish to treat a regular file as if it were a directory; this can happen with the current version. □

Exercise 1-5. The name `tx` matches whichever file named `tc` comes last in the directory, for any single character `c != x`. Can you invent a better distance measure? Implement it and see how well it works for real users. □

Exercise 1-6. Modify `spname` to return a name that is a prefix of the desired name if no closer match can be found. How should ties be broken if there are several names that all match the prefix? □

Exercise 1-7. What other programs could profit from `spname`? Design a standalone program that would apply correction to its arguments before passing them along to another program, as in

```
fix prog filenames...
```

Can you write a version of `cd` that uses `spname`? How would you install it? □

1.3 File system: inodes

In this section, we will discuss system calls that deal with the file system, in particular, with the information about files, such as size, dates, permissions, and so on. These system calls allow you to obtain at all of the metadata associated with a file.

Let's dig into the inode itself. Part of the inode is described by a structure called `stat`, defined in `<sys/stat.h>`:

```
struct stat { /* structure returned by stat and fstat */
    dev_t     st_dev;        /* ID of device containing file */
    ino_t     st_ino;        /* inode number */
    mode_t    st_mode;       /* file type and mode */
    nlink_t   st_nlink;      /* number of hard links */
    uid_t     st_uid;        /* user ID of owner */
    gid_t     st_gid;        /* group ID of owner */
    dev_t     st_rdev;       /* device ID (if special file) */
    off_t     st_size;       /* total size, in bytes */
    blksize_t st_blksize;    /* blocksize for filesystem I/O */
    blkcnt_t  st_blocks;     /* number of 512B blocks allocated */
    struct timespec st_atim; /* time of last access */
    struct timespec st_mtim; /* time of last modification */
    struct timespec st_ctim; /* time of last status change */
#define st_atime st_atim.tv_sec   /* backward compatibility */
#define st_mtime st_mtim.tv_sec
#define st_ctime st_ctim.tv_sec
};
```

Most of the fields are explained by the comments. Types like `dev_t` and `ino_t` are defined in `<sys/types.h>`. The `st_mode` entry contains a set of flags describing the file; for convenience, the flag definitions are also part of `<sys/stat.h>`:

```
#define S_IFMT   0170000  /* bit mask for the file type bit field */
#define S_IFSOCK 0140000  /* socket */
#define S_IFLNK  0120000  /* symbolic link */
#define S_IFREG  0100000  /* regular file */
#define S_IFBLK  0060000  /* block device */
#define S_IFDIR  0040000  /* directory */
#define S_IFCHR  0020000  /* character device */
#define S_IFIFO  0010000  /* FIFO */

#define S_ISUID  04000    /* set-user-ID bit */
#define S_ISGID  02000    /* set-group-ID bit */
#define S_ISVTX  01000    /* sticky bit */

#define S_IRWXU  00700    /* owner has read, write & execute permission */
#define S_IRUSR  00400    /* owner has read permission */
#define S_IWUSR  00200    /* owner has write permission */
#define S_IXUSR  00100    /* owner has execute permission */
```

```
#define S_IRWXG  00070    /* group has read, write & execute permission */
#define S_IRGRP  00040    /* group has read permission */
#define S_IWGRP  00020    /* group has write permission */
#define S_IXGRP  00010    /* group has execute permission */

#define S_IRWXO  00007    /* other has read, write & execute permission */
#define S_IROTH  00004    /* other has read permission */
#define S_IWOTH  00002    /* other has write permission */
#define S_IXOTH  00001    /* other has execute permission */
```

The inode for a file is accessed by a pair of system calls named `stat` and `fstat`. `stat` takes a filename and returns inode information for that file (or `-1` if there is an error). `fsat` does the same from a file descriptor for a file that is already opened (*not* from a `FILE` pointer). That is,

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

char *name;
int fd;
struct stat stbuf;

stat(name, &stbuf);
fstat(fd, &stbuf);
```

For example, suppose that we want to modify `simple-ls` to show the type of each directory entry that we obtain from a directory.

```
/* simple-ls: list contents of a directory, one per line - version 2 */

#include <sys/types.h> /* needed for dirent.h below */
#include <dirent.h>    /* for DIR, struct dirent, opendir(), readdir(),
                          closedir() */
#include <errno.h>     /* for errno, sys_errlist[] */
#include <sys/stat.h>  /* for struct stat, stat(), S_* */
#include <stdio.h>     /* for BUFSIZ, sprintf(), printf() */
#include <string.h>    /* for strlen() */
#include <stdlib.h>    /* for EXIT_SUCCESS */

int main(int argc, char *argv[]) {
    int i;
    DIR *dd;
    struct dirent *dent;
    struct stat sb;

    if (argc == 1) { /* list current working directory */
        argc = 2;
        argv[1] = ".";
```

```
    }
    for (i = 1; i < argc; i++) {
        int n;
        char *sepstr;

        if ((dd = opendir(argv[i])) == NULL) {
            fprintf(stderr, "Error opening directory %s (%s)\n",
                    argv[i], sys_errlist[errno]);
            continue;
        }
        n = strlen(argv[i]) - 1;
        sepstr = (argv[i][n] == '/') ? "" : "/";
        while ((dent = readdir(dd)) != NULL) {
            char *p, buf[BUFSIZ];
            int filetype;

            sprintf(buf, "%s%s%s", argv[i], sepstr, dent->d_name);
            stat(buf, &sb);
            filetype = sb.st_mode & S_IFMT;
            if (filetype == S_IFSOCK)
                p = "skt";
            else if (filetype == S_IFLNK)
                p = "slk";
            else if (filetype == S_IFREG)
                p = "reg";
            else if (filetype == S_IFBLK)
                p = "blk";
            else if (filetype == S_IFDIR)
                p = "dir";
            else if (filetype == S_IFCHR)
                p = "chr";
            else if (filetype == S_IFIFO)
                p = "ffo";
            else
                p = "unk";
            printf("%s %s\n", p, buf);
        }
        closedir(dd);
    }
    return EXIT_SUCCESS;
}
```

After we obtain each entry from the directory, we construct the full path for the entry, and invoke `stat`. Using a bit `AND` operation on the `st_mode` field, we obtain the file type which can then be compared with the defined constants `S_IFDIR`, `S_IFREG`, etc.

1.3.1 sv: An illustration of error handling

We are next going to write a program called `sv`, similar to `cp`, that will copy a set of files to a directory, but change each target file only if it does not currently exist or if the target file is older than the source file.

`sv` stands for "save"; the idea is that `sv` will not overwrite something that appears to be more up to date. `sv` uses more of the information in the inode than `simple-ls` does.

The design we will use for `sv` is this:

```
$ sv file1 file2 ... dir
```

copies `file1` to `dir/file1`, `file2` to `dir/file2`, etc., except that when a target file is newer than its source file, no copy is made and a warning is printed. To avoid making multiple copies of linked files, `sv` does not allow `/`'s in any of the source filenames.

```
/* sv: save new/modified files */
#include <stdio.h>     /* for BUFSIZ, sprintf(), fprintf() */
#include <sys/stat.h>  /* for struct stat, stat(), S_* */
#include <errno.h>     /* for errno, sys_nerr, sys_errlist[] */
#include <string.h>    /* for strchr() */
#include <fcntl.h>     /* for open(), creat() */
#include <stdlib.h>    /* for EXIT_SUCCESS, exit() */
#include <unistd.h>    /* for read(), write(), close() */

char *progname;

void sv(char *file, char *dir);
void error(char *s1, char *s2);

int main(int argc, char *argv[]) {
    int i;
    struct stat sb;
    char *dir = argv[argc-1];

    progname = argv[0];
    if (argc <= 2)
        error("Usage: $s file ... dir", progname);
    if (stat(dir, &sb) == -1)
        error("can't access directory %s", dir);
    if ((sb.st_mode & S_IFMT) != S_IFDIR)
        error("%s is not a directory", dir);
    for (i = 1; i < argc-1; i++)
        sv(argv[i], dir);
    return EXIT_SUCCESS;
}
```

The times in the inode are in seconds-since-long-ago (0:00 GMT, 1 January 1970, also known as the *epoch*), so older files have smaller values in their `st_mtime` field. [4]

```
void sv(char *file, char *dir) { /* save file in dir */
    struct stat sti, sto;
    int fin, fout, n;
    char target[BUFSIZ], buf[BUFSIZ];

    sprintf(target, "%s/%s", dir, file);
    if (strchr(file, '/') != NULL)
        error("won't handle /'s in %s", file);
    if (stat(file, &sti) == -1)
        error("can't stat %s", file);
    if (stat(target, &sto) == -1) /* target does not exist */
        sto.st_mtime = 0; /* so make it look old */
    if (sti.st_mtime < sto.st_mtime) /* target is newer */
        fprintf(stderr, "%s: %s not copied\n", progname, file);
    else if ((fin = open(file, 0)) == -1)
        error("can't open file %s", file);
    else if ((fout = creat(target, sti.st_mode)) == -1)
        error("can't create %s", target);
    else
        while ((n = read(fin, buf, sizeof buf)) > 0)
            if (write(fout, buf, n) != n)
                error("error writing %s", target);
    close(fin);
    close(fout);
}
```

We used `creat` instead of the standard I/O functions so that `sv` can preserve the mode of the input file.

Although the `sv` program is rather specialized, it does indicate some important ideas. Many programs are not "system programs" but may still use information maintained by the operating system and accessed through system calls. For such programs, it is crucial that the representation of the information appear only in standard header files like `<sys/stat.h>` and `<dirent.h>`, and that programs include those files instead of embedding the actual declarations in themselves. Such code is much more likely to be portable from one system to another.

It is also worth noting that at least two thirds of the code in `sv` is error checking. In the early stages of writing a program, it's tempting to skimp on error handling, since it is a diversion from the main task. And once the program "works," it's hard to be enthusiastic about going back to put in the checks that convert a private program into one that works regardless of what happens.

`sv` isn't proof against all possible disasters - it doesn't deal with interrupts at awkward times, for instance - but it's more careful than most programs. To focus on just one point for a moment, consider the final `write` statement. It is rare that a `write` fails, so many

[4] The times are actually seconds and nanoseconds since the epoch; note that `st_mtime` is defined to be `st_mtim.tv_sec`; our code ignores the nanoseconds.

programs ignore the possibility. But discs run out of space; users exceed quotas; communications lines break. All of these can cause write errors, and you are a lot better off if you hear about them than if the program silently pretends that all is well.

The moral is that error checking is tedious but important. We have been cavalier in most of the programs in this book because of space limitations and to focus on more interesting topics. But for real, production programs, you can't afford to ignore errors.

Exercise 1-8. Write a program `watchfile` that monitors a file and prints the file from the beginning each time it changes. When would you use it? □

Exercise 1-9. `sv` is quite rigid in its error handling. Modify it to continue even if it can't process a particular file. □

Exercise 1-10. Make `sv` recursive - if one of the source files is a directory, that directory and its files are processed in the same manner. Make `cp` recursive. Discuss whether `cp` and `sv` ought to be the same program, so that `cp -v` doesn't do the copy if the target is newer. □

Exercise 1-11. Write the program `random`:

```
$ random filename
```

produces one line chosen at random from `filename`. Given a file `people` of names, `random` can be used in a program called `scapegoat`, which is valuable for allocating blame:

```
$ cat scapegoat
echo "It's all `random people`'s fault"!
$ ./scapegoat
It's all Joe's fault!
```

Make sure that `random` is fair regardless of the distribution of line lengths. □

1.4 Processes

This section describes how to execute one program from within another. There are higher-level library functions, like `system(3)`, but we are going to focus on the system calls that enable you to construct arbitrary trees of processes.

1.4.1 Low-level process creation - `execlp` and `execvp`

The most basic operation is to execute another program *without returning*, by using the system call `execlp`. For example, to print the date as the last action of a running program, use

```
#include <unistd.h>

execlp("date", "date", NULL);
```

The first argument to `execlp` is the filename of the command; `execlp` extracts the search path (i.e. `"PATH"`) from your environment and does the same search as the shell does. The second and subsequent arguments are the command name and the arguments for the command; these become the `argv` array for the new program. The end of the list is marked by a `NULL` argument. (Read `execve(2)` for insight on the design of `execlp`.)

The `execlp` call overlays the existing program with the new one, runs that, then exits. The original program gets control back only when there is an error, for example if the file cannot be found or is not executable:

```
execlp("date", "date", NULL);
fprintf(stderr, "Couldn't execute 'date'\n");
exit(1);
```

A variant of `execlp` called `execvp` is useful when you don't know in advance how many arguments there are going to be. The call is

```
execvp(filename, argp);
```

where `argp` is an array of pointers to the arguments (such as `argv`); the last pointer in the array must be `NULL` so `execvp` can tell where the list ends. As with `execlp`, `filename` is the file in which the program is found, and `argp` is the `argv` array for the new program; `argp[0]` is the program name.

Neither of these routines provides expansion of metacharacters like `<`, `>`, `*`, quotes, etc., in the argument list. If you want this type of processing, use `execlp` to invoke the shell `/usr/bin/sh` which then does all of the work. Construct a string `commandline` that contains the complete command as it would have been typed at the terminal (without the final newline), then invoke

```
execlp("/usr/bin/sh", "sh", "-c", commandline, NULL);
```

The argument `-c` says to treat the next argument as the whole command line, not a single argument.

As an illustration of `exec` use, consider the program `waitfile`. The command

```
$ waitfile filename [command]
```

Periodically checks the file name. If it is unchanged since the last time it was checked, the `command` is executed. If no `command` was specified, the file is copied to the standard output. `waitfile` can be used to monitor the progress of an application that generates its output in a single file, and that takes a significant amount of time to complete its processing. For example, if `slowprog` generates its output in `slowprog.out`, then `waitfile` can be used as in

```
$ waitfile slowprog.out echo slowprog has finished &
```

The implementation of `waitfile` uses `fstat` to extract the time when the file was last changed.

```
/* waitfile: wait until file stops changing */

#include <stdio.h>     /* for fprintf() */
#include <sys/stat.h>  /* for struct stat, fstat() */
#include <errno.h>     /* for errno, sys_nerr, sys_errlist[] */
#include <unistd.h>    /* for sleep(), execlp(), execvp() */
#include <stdlib.h>    /* for EXIT_SUCCESS, exit() */
#include <fcntl.h>     /* for open(), creat() */

#define DELTA_T 60     /* number of seconds between checks */

char *progname;

void error(char *s1, char *s2);

int main(int argc, char *argv[]) {
    int fd;
    struct stat stbuf, oldbuf;

    progname = argv[0];
    if (argc < 2)
        error("Usage: %s filename [command]", progname);
    if ((fd = open(argv[1], 0)) == -1)
        error("can't open %s", argv[1]);
    oldbuf.st_mtime = 0;
    fstat(fd, &stbuf);
    while (stbuf.st_mtime != oldbuf.st_mtime) {
        oldbuf.st_mtime = stbuf.st_mtime;
        sleep(DELTA_T);
        fstat(fd, &stbuf);
    }
    if (argc == 2) {          /* copy file to standard output */
        execlp("cat", "cat", argv[1], NULL);
        error("can't execute \"cat %s\"", argv[1]);
    } else {
        execvp(argv[2], &argv[2]);
        error("can't execute %s", argv[2]);
    }
    return EXIT_SUCCESS;
}
```

This illustrates both `execlp` and `execvp`.

We picked this design because it's useful, but other variations are plausible. For example, `waitfile` could simply return after the file has stopped changing.

Exercise 1-12. Modify `watchfile` (Exercise 1-8) so it has the same property as `waitfile` - if there is no `command`, it copies the file; otherwise it does the command. Could `watchfile` and `waitfile` share source code? Hint: `argv[0]`. □

1.4.2 Control of processes - `fork` and `wait`

The next step is to regain control after running a program with `execlp` or `execvp`. Since these routines simply overlay the new program on the old one, to save the old one requires that it first be split into two copies; one of these can be overlaid, while the other waits for the new, overlaying program to finish. The splitting is done by a system call named `fork`:

```
#include <unistd.h>
pid_t proc_id;

proc_id = fork();
```

splits the program into two copies, both of which continue to run. The only different between the two is the value returned by `fork`, the *process-id*. In one of these processes (the *child*), `proc_id` is zero. In the other (the *parent*), `proc_id` is non-zero; it is the process-id of the child. Thus, the basic way to call, and return from, another program is

```
if (fork() == 0)
    execlp("/usr/bin/sh", "sh", "-c", commandline, NULL);
```

And in fact, except for handling errors, this is sufficient. The `fork` makes two copies of the program. In the child, the value returned by `fork` is zero, so it calls `execlp` which does the `commandline` and then dies. In the parent, `fork` returns non-zero so it skips the `execlp`. (If there is any error, `fork` returns `-1`.)

More often, the parent waits for the child to terminate before continuing itself. This is done with the system call `wait`:

```
#include <sys/types.h>
#include <sys/wait.h>
int status;

if (fork() == 0)
    execlp(...);      /* child */
wait(&status);        /* parent */
```

This still doesn't handle any abnormal conditions, such as a failure of `execlp` or `fork`, or the possibility that there might be more than one child running simultaneously. (`wait` returns the process-id of the terminated child, if you want to check it against the value returned by `fork`.) Finally, this fragment does not deal with any funny behavior on the

Table 1.2: Macros to process wait() status

`WIFEXITED(status)`	returns true if the child terminated normally, that is, by calling `exit(3)` or `_exit(2)`, or by returning from `main()`
`WEXITSTATUS(status)`	returns the exit status of the child. This consists of the least significant 8 bits of the `status` argument that the child specified in a call to `exit(3)` or `_exit(2)` or as the argument for a return statement in `main()`. This macro should only be employed if `WIFEXITED` returned true.
`WIFSIGNALED(status)`	returns true if the child process was terminated by a signal.
`WTERMSIG(status)`	returns the number of the signal that caused the child process to terminate. This macro should only be employed if `WIFSIGNALED` returned true.
`WIFSTOPPED(status)`	returns true if the child process was stopped by delivery of a signal; this is only possible if the call was done using `WUNTRACED` or when the child is being traced (see `ptrace(2)`).
`WSTOPSIG(status)`	returns the number of the signal which caused the child to stop. This macro should only be employed if `WIFSTOPPED` returned true.
`WIFCONTINUED(status)`	returns true if the child process was resumed by delivery of `SIGCONT`.

part of the child. Still, these three lines are the heart of the standard `system` function.[5]

The `status` returned by `wait` encodes in its low-order eight bits the system's idea of the child's exit status; it is 0 for normal termination and non-zero to indicate various kinds of problems. The next higher eight bits are taken from the argument of the call to `exit` or return from `main` that caused termination of the child process. Macros are defined to enable your program to inspect the status value returned; these macros are provided in Table 1.2.

When a program is invoked by the shell, the three file descriptors `0`, `1`, and `2` are set up pointing at the correct files, and all other file descriptors are available for use. When this program invokes another one, correct etiquette suggests making sure that the same conditions hold. Neither `fork` nor `exec` calls affect open files in any way; both parent and child have the same open files. If the parent is buffering output that must come out before output from the child, the parent must flush its buffers before the `execlp` or `execvp` call. Conversely, if the parent buffers an input stream, the child will lose any information that has been read by the parent. Output can be flushed, but input cannot be put back. Both of these considerations arise if the input or output is being done with the standard I/O library, since it normally buffers both input and output.

The system call `dup(fd)` duplicates the file descriptor `fd` on the lowest numbered

[5] See the man page for `waitpid(2)` if you desire more sophisticated ways to wait for child processes to terminate.

unallocated file descriptor, returning a new descriptor that refers to the same open file. [6] This code connects the standard input of a program to a file:

```
int fd;

fd = open("file", 0);
close(0);
dup(fd);
close(fd);
```

The `close(0)` deallocates file descriptor `0`, the standard input, but as usual doesn't affect the parent.

1.4.3 Signals and interrupts

This section is concerned with how to deal gracefully with signals (like interrupts) from the outside world, and with program faults. Program faults arise mainly from illegal memory references, execution of peculiar instructions, or floating point errors. The most common outside-world signals are *interrupt*, which is sent when the *ctl-c* character is typed; *quit*, generated when the *ctl-* is typed; and *terminate*, generated by the `kill` command. When one of these events occurs, the signal is sent to all processes that were started from the same terminal; unless other arrangements have been made, the signal terminates the process. For many signals, a core image file is written for potential debugging.[7]

The system call `signal` alters the default action. It has two arguments. The first is a number that specifies the signal. The second is either the address of a function, or a code which requests that the signal be ignored or be given the default action. The file `<signal.h>` contains definitions for the various arguments. Thus

```
#include <signal.h>
* * *
signal(SIGINT, SIG_IGN);
```

causes interrupts to be ignored, while

```
signal(SIGINT, SIG_DFL);
```

restores the default action of process termination. In all cases, `signal` returns the previous value of the signal. If the second argument to `signal` is the name of a function (which must have been declared already in the same source file), the function will be called when the signal occurs. Most commonly this facility is used to allow the program to clean up unfinished business before terminating, for example to delete a temporary file:

---

[6] `dup2(old, new)` creates a copy of `old` using `new`. If the file descriptor `new` was previously open, it is silently closed before being reused.

[7] See `signal(7)` for those signals that generate a core file. Core files can be used with `gdb(1)`.

```
#include <signal.h>
#include <stdlib.h>
#include <unistd.h>

#define UNUSED __attribute__((unused))

char tempfile[] = "onintr.XXXXXX";

void onintr(UNUSED int sig) {
    unlink(tempfile);
    exit(EXIT_FAILURE);
}

int main(UNUSED int argc, UNUSED char *argv[]) {
    if (signal(SIGINT, SIG_IGN) != SIG_IGN)
        signal(SIGINT, onintr);
    mkstemp(tempfile);

    /* logic using tempfile */

    return EXIT_SUCCESS;
}
```

Why the test and the double call to `signal` in `main`? Recall that `interrupt` and `quit` signals are sent to all processes started from a particular terminal. Accordingly, when a program is to be run non-interactively (started by &), the shell arranges that the program will ignore interrupts, so it won't be stopped by interrupts intended for foreground processes. If this program began by announcing that all interrupts were to be sent to the `onintr` routine regardless, that would undo the shell's effort to protect it when run in the background.

The solution, shown above, is to test the state of interrupt handling, and to continue to ignore interrupts if they are already being ignored. The code as written depends on the fact that `signal` returns the previous state of a particular signal. If signals were already being ignored, the process should continue to ignore them; otherwise, they should be caught.

You will also note the introduction in the code above of the definition of `UNUSED`. The standard signature for a signal handling function is

```
void function_name(int signal)
```

`onintr` does not access its integer argument; the `gcc` compiler will issue a warning about this lack of use - it is often an indication of a programming error. The definition of `UNUSED` as a `gcc` attribute and its use as a prefix to the declaration of the argument to `onintr` indicates to the compiler (and to future readers of your code) that you are explicitly not using the `signal` parameter. We have also used `UNUSED` to indicate that we do not use `argc` or `argc` in `main`. If you are not using `gcc`, there will likely be some other compiler-specific way to indicate that you are explicitly not using one or more arguments.

A more sophisticated program may wish to intercept an interrupt and interpret it as a request to stop what it is doing and return to its own command-processing loop. Think of a text editor – interrupting a long printout should not cause it to exit and lose the work already done. The code for this case can be written like this:

```
#include <signal.h>
#include <stdlib.h>
#include <stdio.h>
#include <setjmp.h>

#define UNUSED __attribute__((unused))

jmp_buf sjbuf;

void onintr(UNUSED int sig) {
    signal(SIGINT, onintr); /* reset for next interrupt */
    fprintf(stderr, "\nInterrupt received\n");
    longjmp(sjbuf, 0); /* return to saved state */
}

int main(UNUSED int argc, UNUSED char *argv[]) {
    if (signal(SIGINT, SIG_IGN) != SIG_IGN)
        signal(SIGINT, onintr);
    setjmp(sjbuf); /* save current stack position */

    for (;;) {
        /* main processing loop */
    }
    return EXIT_SUCCESS;
}
```

The file `<setjmp.h>` declares the type `jmp_buf` as an object in which the current stack position can be saved; `sjbuf` is declared to be such an object. The function `setjmp(3)` saves a record of where the program was executing. The values of variables are *not* saved. When an interrupt occurs, a call is forced to the `onintr` routine, which can print a message, set flags, or whatever. `longjmp` takes as an argument an object stored into by `setjmp`, and restores control to the location after the call to `setjmp`. So control (and the stack level) will pop back to the place in the main routine where the main loop is entered.

Notice that the signal is set again in `onintr` after an interrupt occurs. This is necessary – signals are automatically reset to their default action when they occur.[8]

Some programs that want to detect signals simply can't be stopped at an arbitrary point, for example in the middle of updating a complicated data structure. The solution is to

[8] Setting the signal again within the signal handler is *not* required in Linux if the `_DEFAULT_SOURCE` macro is defined before one includes `<signal.h>`. There is a race condition between when the signal is delivered to the handler and the default handler is overridden, during which time the signal could be delivered again. By defining `_DEFAULT_SOURCE` in your code, this race condition is eliminated, with delivery of another instance of that signal blocked while you are executing your handler, and upon return from the handler your handler is still associated with that signal.

have the interrupt handler routine set a flag and return instead of calling `exit` or `longjmp`. Execution will continue at the exact point it was interrupted, and the interrupt flag can be tested later.

```
#define _DEFAULT_SOURCE

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

#define UNUSED __attribute__((unused))

int interrupted = 0;

void onintr(UNUSED int sig) {
    interrupted++; /* indicate that interrupt was received */
}

int main(UNUSED int argc, UNUSED char *argv[]) {
    if (signal(SIGINT, SIG_IGN) != SIG_IGN)
        signal(SIGINT, onintr);

    for (; !interrupted;) {
        /* main processing loop */
    }
    return EXIT_SUCCESS;
}
```

There is one difficulty associated with this approach. Suppose the program is reading the terminal when the interrupt is sent. The specified routine is duly called; it sets its flag and returns. If it were really true, as we said above, that execution resumes "at the exact point it was interrupted," the program would continue reading the terminal until the user typed another line. This behavior might well be confusing, since the user might not know that the program is reading, and presumably would prefer to have the signal take effect instantly. To resolve this difficulty, the system terminates the `read`, but with an error status that indicates what happened – `errno` is set to `EINTR`, defined in `<errno.h>`, to indicate an interrupted system call.

Thus programs that catch and resume execution after signals should be prepared for "errors" caused by interrupted system calls. (The system calls to watch out for are `read`s from a terminal, `wait`, and `pause`.) [9] Such a program could use code like the following when it reads the standard input:

[9] `pause(2)` causes the calling process (or thread) to sleep until a signal is delivered that either terminates the process or causes the invocation of a signal handler. It is defined in `<unistd.h>`.

```
#include <errno.h>
* * *
if (read(0, &c, 1) <= 0)     /* EOF or interrupted */
    if (errno == EINTR) {    /* interrupted */
        errno = 0;           /* reset for next time */
        * * *
    } else {                 /* EOF */
        * * *
    }
```

There is a final subtlety to keep in mind when signal-catching is combined with execution of other programs. Suppose a program catches interrupts, and also includes a method whereby other programs can be executed. Then the code would look something like this:

```
if (fork() == 0)
    execlp(...)
signal(SIGINT, SIG_IGN);   /* parent ignores interrupts */
wait(&status);             /* until child has finished */
signal(SIGINT, onintr);    /* restore interrupt handling */
```

Why is this? Signals are sent to all your processes. Suppose the program you call catches its own interrupts, as an editor does. If you interrupt the subprogram, it will get the signal and return to its main loop, and probably read your terminal. But the calling program will also pop out of its `wait` for the subprogram and read your terminal. Having two processes reading your terminal is very confusing, since in effect the system flips a coin to decide who should get each line of input. The solution is to have the parent program ignore interrupts until the child is done.

As an aside on declarations, the function `signal` obviously has a rather strange second argument. It is in fact a pointer to a function returning `void` that takes a single integer argument, and this is also the return type of the `signal` routine itself. If you need to define variables into which you store the return value from `signal()`, I recommend that you declare the following typedef:

```
typedef void (*sighandler_t)(int);
```

and declare those variables as follows:

```
sighandler_t int_variable, quit_variable;
```

This is much simpler than declaring them directly, as in:

```
void (*int_variable)(int), (*quit_variable)(int);
```

and much clearer to someone reading your code.

1.4.4 Alarms

The system call `alarm(n)` causes a signal `SIGALRM` to be sent to your process `n` seconds later. The alarm signal can be used for making sure that something happens within the proper amount of time; if the something happens, the alarm signal can be turned off, but if it does not, the process can regain control by catching the alarm signal.

To illustrate, here is a program called `timeout` that runs another command; if that command has not finished by the specified time, it will be aborted when the alarm goes off.

The code in `timeout` illustrates almost everything we have talked about in the past two sections. The child is created; the parent sets an alarm and then waits for the child to finish. If the alarm arrives first, the child is killed. An attempt is made to return the child's exit status.

```
/* timeout: set time limit on a process */
#include <signal.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <errno.h>

#define UNUSED __attribute__((unused))   /* compiler-dependent */

int pid; /* child process id */
char *progname;

void error(char *s1, char *s2);

void onalarm(UNUSED int sig) {
    kill(pid, SIGKILL);
}

int main(int argc, char *argv[]) {
    int sec = 10, status;

    progname = argv[0];
    if (argc > 1 && argv[1][0] == '-') {
        sec = atoi(&argv[1][1]);
        argc--;
        argv++;
    }
    if (argc < 2)
        error("Usage: %s [-seconds] command", progname);
    if ((pid = fork()) == 0) {
        execvp(argv[1], &argv[1]);
        error("couldn't start %s", argv[1]);
    }
```

```
    signal(SIGALRM, onalarm);
    alarm(sec);
    if (wait(&status) == -1 || WIFSIGNALED(status))
        error("%s killed", argv[1]);
    return WEXITSTATUS(status);
}
```

Exercise 1-13. Can you infer how `sleep(3)` is implemented? Hint: `pause(2)`. Under what circumstances, if any, could `sleep` and `alarm` interfere with each other. □

1.4.5 More sophisticated timing features

Linux provides each process with three interval timers, each decrementing in a distinct time domain. When any timer expires, a signal is sent to the process, and the timer restarts.

The three interval timers are:

1. `ITIMER_REAL`: decrements in real time, and delivers `SIGALRM` upon expiration.
2. `ITIMER_VIRTUAL`: decrements only when the process is executing, and delivers `SIGVTALRM` upon expiration.
3. `ITIMER_PROF`: decrements both when the process executes and when the system is executing on behalf of the process, and delivers `SIGPROF` upon expiration. Coupled with `ITIMER_VIRTUAL`, this timer is usually used to profile the time spent by the application in user and kernel space.

If your application needs to periodically take some action, this can be easily done by using one of these interval timers. If the action requires managing things external to the current process (e.g., child processes, network communications), they you will use `ITIMER_REAL`.

The following program provides a simple example of the interval timer use:

```
/* setitimer: simple use of the interval timer */

#include <sys/time.h>   /* for setitimer */
#include <unistd.h>     /* for pause */
#include <signal.h>     /* for signal */
#include <stdio.h>

#define UNUSED __attribute__((unused))

#define INTERVAL 500 /* number of milliseconds */
#define NALARMS 10 /* number of alarms to receive */

int alarms_left = NALARMS;
```

```
static void onalarm(UNUSED int sig) {
    printf("Timer went off.\n");
    alarms_left--;
}

int main(UNUSED int argc, UNUSED char *argv[]) {
    struct itimerval it_val; /* for setting itimer */

    /* Upon SIGALRM, call onalarm().
     * Set interval timer.  We want frequency in ms,
     * but the setitimer call needs seconds and useconds. */
    if (signal(SIGALRM, onalarm) == SIG_ERR) {
        perror("Unable to catch SIGALRM");
        return 1;
    }
    it_val.it_value.tv_sec = INTERVAL/1000;
    it_val.it_value.tv_usec = (INTERVAL*1000) % 1000000;
    it_val.it_interval = it_val.it_value;
    if (setitimer(ITIMER_REAL, &it_val, NULL) == -1) {
        perror("error calling setitimer()");
        return 1;
    }
    while (alarms_left)
        pause();
    return 0;
}
```

1.4.6 Using signals to handle child processes

You may encounter situations in which your program creates many child processes to solve some problem. Management of your application probably entails harvesting each process as it terminates, and then taking any appropriate action (e.g. if you have two child processes, one which is a data producer, and the other is a data consumer, and the data consumer terminates for any reason, you probably need to kill the producer.) It should be clear from the discussion in Section 1.4.2 that `wait` permits you to wait for the next child process to terminate. Therefore, your code could use `wait` to implement your process management requirements. This would be problematic if the parent process also had to perform application functions while managing the child processes.

Every time a major event occurs in a child process, a `SIGCHLD` signal is delivered to its parent. By handling these signals, you can implement your process management functionality in your signal handler, thus leaving the mainline of your parent process to perform other activities.

```
/* Simplest dead child cleanup in a SIGCHLD handler. Prevents zombie processes
 * but doesn't actually do anything with the information that a child died.
 */
```

```c
#include <unistd.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <signal.h>
#include <stdio.h>

#define NUM_OF_CHILDREN 10
#define UNUSED __attribute__ ((unused))

int active_processes = NUM_OF_CHILDREN;

/* SIGCHLD handler. */
static void CHLD_handler(UNUSED int sig) {
    pid_t pid;
    int status;

    /* Wait for all dead processes.
     * We use a non-blocking call to be sure this signal handler will not
     * block if a child was cleaned up in another part of the program. */
    while ((pid = waitpid(-1, &status, WNOHANG)) > 0) {
        if (WIFEXITED(status) || WIFSIGNALED(status)) {
            active_processes--;
            fprintf(stderr, "%d: exited\n", pid);
        }
    }
}

int main (UNUSED int argc, UNUSED char *argv[])
{
    int i;

    if (signal(SIGCHLD, CHLD_handler) == SIG_ERR) {
        fprintf(stderr, "Can't establish SIGCHLD handler\n");
        return 1;
    }
    /* Make some children. */
    for (i = 0; i < NUM_OF_CHILDREN; i++) {
        switch (fork()) {
            case -1:
                fprintf(stderr, "Unable to fork child %d\n", i);
                return 1;
            case 0: /* child sleeps then exits */
                sleep(5);
                return 0;
        }
    }
    /* Wait while there are still active processes. */
        while (active_processes)
            pause();
    return EXIT_SUCCESS;
```

```
}
```

Particular things to note in this code:

1. We use `waitpid` in the `SIGCHLD` handler. Specifying the first argument as `-1` indicates that we should wait for any event in any of our children. The third argument, `WNOHANG`, indicates that `waitpid` should return immediately if there are no more events associated with child processes; in such a situation, `waitpid` returns a value of `0`; thus, each invocation of the `SIGCHLD` handler harvests all child process events that may have occurred since the last invocation until there are no more left.
2. Events not only correspond to child process termination. Thus, we must check to see if the event corresponds to a child process exit (`WIFEXITED`) or a child process terminated by a signal (`WIFSIGNALED`). If either of these are true, this code simply decrements the global variable `active_processes` and prints a message on `stderr` indicating that the process has terminated.
3. `main()` is particularly simple. First we establish a `SIGCHLD` signal handler. Then we create `NUM_OF_CHILDREN` child processes; each child process invokes `sleep(5)` and then returns successfully. After creating the child processes, the mainline code simply waits until all of the child processes have exited.