Chapter 1

1.1 For this exercise, we use the following two arguments for the `ls`(1) command: `-i` prints the i-node number of the file or directory (we say more about i-nodes in Section 4.14), and `-d` prints information about a directory instead of information on all the files in the directory.

Execute the following:

```
$ ls -ldi /etc/. /etc/..          -i says print i-node number
 162561 drwxr-xr-x  66 root    4096 Feb  5 03:59 /etc/./
      2 drwxr-xr-x  19 root    4096 Jan 15 07:25 /etc/../
$ ls -ldi /. /..                  both . and .. have i-node number 2
      2 drwxr-xr-x  19 root    4096 Jan 15 07:25 /./
      2 drwxr-xr-x  19 root    4096 Jan 15 07:25 /../
```

1.2 The UNIX System is a multiprogramming, or multitasking, system. Other processes were running at the time this program was run.

1.3 Since the *msg* argument to `perror` is a pointer, `perror` could modify the string that *msg* points to. The qualifier `const`, however, says that `perror` does not modify what the pointer points to. On the other hand, the error number argument to `strerror` is an integer, and since C passes all arguments by value, the `strerror` function couldn't modify this value even if it wanted to. (If the handling of function arguments in C is not clear, you should review Section 5.2 of Kernighan and Ritchie [1988].)

1.4 During the year 2038. We can solve the problem by making the `time_t` data type a 64-bit integer. If it is currently a 32-bit integer, applications will have to be recompiled to work properly. But the problem is worse. Some file systems and backup media store times in 32-bit integers. These would need to be updated as well, but we still need to be able to read the old format.

1.5 Approximately 248 days.

Chapter 1 Supplemental Exercises and Answers

1.6 **EXERCISE:** Most UNIX system programs are relatively small and designed to do one thing well. List at least three advantages to this approach.

SOLUTION:

1. Smaller programs are easier to maintain.
2. Smaller programs are easier to test.
3. It is easier to have confidence that a smaller program does what it is supposed to do than a larger program.

4. Small programs can be combined in interesting ways to solve new problems. For example, to find all of the misspelled words in a document, you might type

```
spell filename | sort | uniq
```

This is superior to building a spelling checker into every document editor.

1.7 **EXERCISE:** Usually the CPU time of a program doesn't exceed the wall clock time (the running time) of the program. Explain the circumstances under which this can be false.

SOLUTION: On a multiprocessor, if an application is multithreaded, more than one processor can accumulate CPU time at the same time, so it is possible for the CPU time to exceed the running time of the program.

Chapter 2

2.1 The following technique is used by FreeBSD. The primitive data types that can appear in multiple headers are defined in the header `<machine/_types.h>`. For example:

```
#ifndef _MACHINE__TYPES_H_
#define _MACHINE__TYPES_H_

typedef int             __int32_t;
typedef unsigned int    __uint32_t;
    ⋮

typedef __uint32_t      __size_t;
    ⋮

#endif  /* _MACHINE__TYPES_H_ */
```

In each of the headers that can define the `size_t` primitive system data type, we have the sequence

```
#ifndef  _SIZE_T_DECLARED
typedef __size_t       size_t;
#define _SIZE_T_DECLARED
#endif
```

This way, the `typedef` for `size_t` is executed only once.

2.2 This is an activity for the reader, intended to increase familiarity with the system header files.

2.3 If OPEN_MAX is indeterminate or ridiculously large (i.e., equal to LONG_MAX), we can use getrlimit to get the per-process maximum for open file descriptors. Since the per-process limit can be modified, we can't cache the value obtained from the previous call (it might have changed). See Figure 1.

```
#include "apue.h"
#include <limits.h>
#include <sys/resource.h>

#define OPEN_MAX_GUESS  256

long
open_max(void)
{
    long openmax;
    struct rlimit rl;

    if ((openmax = sysconf(_SC_OPEN_MAX)) < 0 ||
      openmax == LONG_MAX) {
        if (getrlimit(RLIMIT_NOFILE, &rl) < 0)
            err_sys("can't get file limit");
        if (rl.rlim_max == RLIM_INFINITY)
            openmax = OPEN_MAX_GUESS;
        else
            openmax = rl.rlim_max;
    }
    return(openmax);
}
```

Figure 1 Alternative method for identifying the largest possible file descriptor

Chapter 2 Supplemental Exercises and Answers

2.4 **EXERCISE:** Use sysconf to determine the maximum number of standard I/O streams a process can open at a time. Then try to open this number of streams. How many can you open? Does it match what sysconf reports?

SOLUTION: The results differ depending on which platform you use. Solaris 10 reports STREAM_MAX to be 256, but we can open only 253 standard I/O streams. This is because our program already has 3 open streams when it starts: standard input, standard output, and standard error. On Mac OS X 10.6.8, the behavior is similar to Solaris. FreeBSD 8.0 reports STREAM_MAX to be 20,000, but we can only open 15,098 standard I/O streams. In this case, we probably are running into the system-wide limit for open files. Linux 3.2.0 reports STREAM_MAX to be 16, but we can open 1021 streams. This means the limit is really 1024.

2.5 **EXERCISE:** Assume you want to use a feature that is optional for systems to support. Describe three potential strategies to dealing with the possibility that the feature might not be present on a system on which you want to run your program.

SOLUTION: Use `sysconf` to determine if the feature is supported. If it is, use it. Otherwise, potential approaches are:

1. Reduce your program to use the least common denominator—whatever feature that is supported universally that allows you to accomplish the task at hand.
2. If this is insufficient, consider providing your own implementation of the feature for the platforms missing it.
3. If all else fails, you could run your program in a virtual machine using a guest operating system that provides the desired feature. You can communicate with the processes running in the host (native) operating system with sockets, if necessary.

Chapter 3

3.1 All disk I/O goes through the kernel's block buffers (also called the kernel's buffer cache). The exception to this is I/O on a raw disk device, which we aren't considering. (Some systems also provide a *direct I/O* option to allow applications to bypass the kernel buffers, but we aren't considering this option either.) Chapter 3 of Bach [1986] describes the operation of this buffer cache. Since the data that we `read` or `write` is buffered by the kernel, the term *unbuffered I/O* refers to the lack of automatic buffering in the user process with these two functions. Each `read` or `write` invokes a single system call.

3.2 In this case, if we can't call `fcntl`, we need to use `dup`. The problem is that we have no way to control which file descriptor is duplicated to the original file descriptor. To work around this limitation, we continue duplicating file descriptors until we reach the desired file descriptor (see Figure 2). We ensure that we will eventually reach the desired file descriptor by closing that file descriptor beforehand. It doesn't matter if the descriptor wasn't open in the first place; we simply ignore the return value of `close`. Once we have the desired descriptor, we close all of the other duplicates that we no longer need.

```
#include "apue.h"
#include <errno.h>

int
dup2(int fd, int fd2)
{
    long openmax;
    int savederr, idx;
    int *tfd;
```

```
    if (fd == fd2)
        return(fd);
    if ((openmax = open_max()) < 0)
        return(-1);
    if (fd2 < 0 || fd2 >= openmax) {
        errno = EBADF;
        return(-1);
    }
    if ((tfd = malloc(openmax * sizeof(int))) == NULL)
        return(-1);
    close(fd2);
    savederr = 0;
    idx = 0;
    while (idx < openmax) {
        if ((tfd[idx] = dup(fd)) < 0) {
            savederr = errno;
            break;
        }
        if (tfd[idx] == fd2)
            break;
        idx++;
    }
    if (idx >= openmax)
        savederr = EMFILE;
    while (--idx >= 0)
        close(tfd[idx]);
    free(tfd);
    errno = savederr;
    if (errno != 0)
        return(-1);
    else
        return(fd2);
}
```

Figure 2 Implementation of `dup2` without using `fcntl`

3.3 Each call to `open` gives us a new file table entry. However, since both `open`s reference the same file, both file table entries point to the same v-node table entry. The call to `dup` references the existing file table entry. We show this in Figure 3.

An `F_SETFD` on `fd1` affects only the file descriptor flags for `fd1`, but an `F_SETFL` on `fd1` affects the file table entry that both `fd1` and `fd2` point to.

3.4 If `fd` is 1, then the `dup2(fd, 1)` returns 1 without closing file descriptor 1. (Remember our discussion of this in Section 3.12.) After the three calls to `dup2`, all three descriptors point to the same file table entry. Nothing needs to be closed.

If `fd` is 3, however, after the three calls to `dup2`, four descriptors are pointing to the same file table entry. In this case, we need to close descriptor 3.