Chapter 1 Introduction to System Programming

"UNIX is basically a simple operating system, but you have to be a genius to understand the simplicity." - Dennis Ritchie, 1941 - 2011.

Concepts Covered

The kernel and kernel API,
System calls and libraries,
Processes, logins and shells,
Environments, man pages,
Users, the root, and groups,
Authentication,
File system, file hierarchy,
Files and directories,
Device special files,
UNIX standards, POSIX,
System programming,
Terminals and ANSI escape sequences,
History of UNIX,
syscall, getpid, ioctl

1.1 Introduction

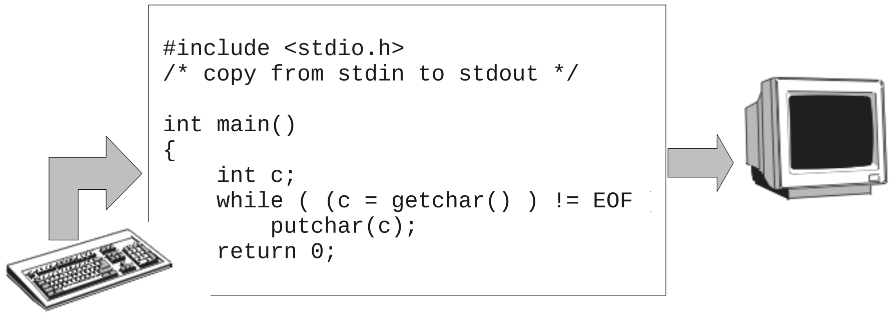
A modern software application typically needs to manage both private and system resources. Private resources are its own data, such as the values of its internal data structures. System resources are things such as files, screen displays, and network connections. An application may also be written as a collection of cooperating threads or sub-processes that coordinate their actions with respect to shared data. These threads and sub-processes are also system resources.

Modern operating systems prevent application software from managing system resources directly, instead providing interfaces that these applications can use for managing such resources. For example, when running on modern operating systems, applications cannot draw to the screen directly or read or write files directly. To perform screen operations or file I/O they must use the interface that the operating system defines. Although it may seem that functions from the C standard library such as `getc()` or `fprintf()` access files directly, they do not; they make calls to system routines that do the work on their behalf.

The interface provided by an operating system for applications to use when accessing system resources is called the operating system's *application programming interface* (*API*). An API typically consists of a collection of function, type, and constant definitions, and sometimes variable definitions as well. The API of an operating system in effect defines the means by which an application can utilize the services provided by that operating system.

It follows that developing a software application for any *platform*[1] requires mastery of that platform's API. Therefore, aside from designing the application itself, the most important task for the application developer is to master the system level services defined in the operating system's API. A program that uses these system level services directly is called a *system program*, and the type of programming that uses these services is called *system programming*. System programs make requests for resources and services directly from the operating system and may even access the system

[1]We use the term platform to mean a specific operating system running on a specific machine architecture.



```
#include <stdio.h>
/* copy from stdin to stdout */

int main()
{
    int c;
    while ( (c = getchar() ) != EOF
        putchar(c);
    return 0;
```

Figure 1.1: Simple I/O model used by beginning programmer.

resources directly. System programs can sometimes be written to extend the functionality of the operating system itself and provide functions that higher level applications can use.

These lecture notes specifically concern system programming using the API of the UNIX operating system. They do not require any prior programming experience with UNIX. They also include tutorial information for those readers who have little experience with UNIX as a user, but this material can be skipped by the experienced UNIX users.

In the remainder of these notes, a distinction will be made between the user's view of UNIX and the *programmer's view* of UNIX. The user's view of UNIX is limited to a subset of commands that can be entered at the command-line and parts of the file system. Some commands and files are not available to all users, as will be explained later. The programmer's view includes the programming language features of the kernel API, the functions, types, and constants in all of the libraries, the various header files, and the various files used by the system. Familiarity with basic C programming is assumed.

1.2 A Programming Illusion

A beginning programmer typically writes programs that follow the simple I/O model depicted in Figure 1.1: the program gets its input from the keyboard or a disk file, and writes its output to the display screen or to a file on disk. Such programs are called *console applications.* because the keyboard and display screen are part of the console device. Listings 1.1 and 1.2 contain examples of such a program, one using the C Standard I/O Library, and the other, the C++ stream library. Both get input from the keyboard and send output to the display device, which is some sort of a console window on a monitor.

The comment in Listing1.1 states that the program copies from stdin to stdout. In UNIX[2], every process has access to abstractions called the standard input device and the standard output device. When a process is created and loaded into memory, UNIX automatically creates the standard input and standard output devices for it, opens them, and makes them ready for reading and writing respectively[3]. In C (and C++), `stdin` and `stdout` are variables defined in the `<stdio.h>`

[2]In fact, every POSIX-compliant operating system must provide both a standard input and standard output stream.

[3]It also creates a standard error device that defaults to the same device as standard output.

header file, that refer to the standard input and standard output device[4] respectively. By default, the keyboard and display of the associated terminal are the standard input and output devices respectively.

Listing 1.1: C program using simple I/O model.

```
#include <stdio.h>
/* copy from stdin to stdout */
int main()
{
    int c;
    while ( (c = getchar() ) != EOF )
        putchar(c);
    return 0;
}
```

Listing 1.2: Simple C++ program using simple I/O model.

```
#include <iostream>
using namespace std;
/* copy from stdin to stdout using C++ */
int main()
{
    char c;
    while ( (c = cin.get() ) && !cin.eof() )
        cout.put(c);
    return 0;
}
```

These programs give us the illusion that they are directly connected to the keyboard and the display device via C library functions `getchar()` and `putchar()` and the C++ `iostream` member functions `get()` and `put()`. Either of them can be run on a single-user desktop computer or on a multi-user, time-shared workstation in a terminal window, and the results will be the same. If you build and run them as console applications in Windows, they will have the same behavior as if you built and ran them from the command-line in a UNIX system.

On a personal computer running in single-user mode, this illusion is not far from reality in the sense that the keyboard is indirectly connected to the input stream of the program, and the monitor is indirectly connected to the output stream. This is not the case in a multi-user system.

In a multi-user operating system, several users may be logged in simultaneously, and programs belonging to different users might be running at the same time, each receiving input from a different keyboard and sending output to a different display. For example, on a UNIX computer on a network into which you can login, many people may be connected to a single computer via a network program such as SSH, and several of them will be able to run the above program on the same computer at the same time, sending their output to different terminal windows on physically different computers, and each will see the same output as if they had run the program on a single-user machine.

As depicted in Figure 1.2, UNIX ensures, in a remarkably elegant manner, that each user's *processes* have a logical connection to their keyboard and their display. (The process concept will be explained

[4]In C and C++, stderr is the variable associated with the standard error device.

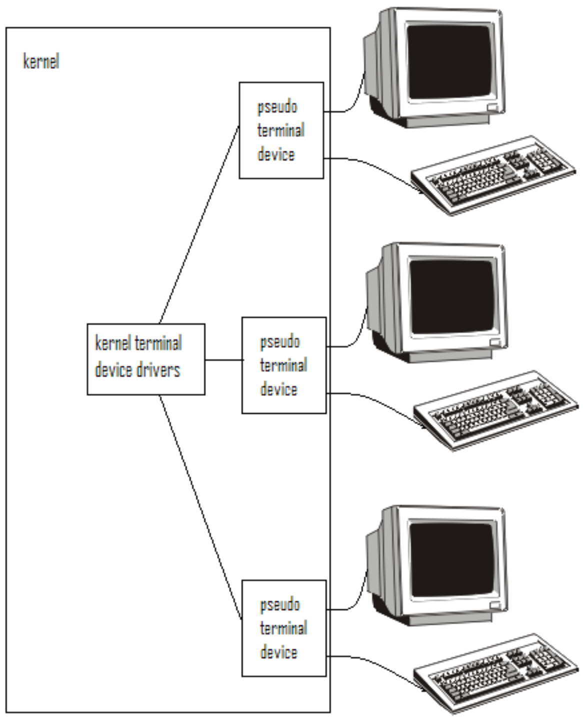


Figure 1.2: Connecting multiple users to a UNIX system.

shortly.) Programs that use the model of I/O described above do not have to be concerned with the complexities of connecting to monitors and keyboards, because the operating system hides that complexity, presenting a simplified interface for dealing with I/O. To understand how the operating system achieves this, one must first understand several cornerstone concepts of the UNIX operating system: files, processes, users and groups, privileges and protections, and environments.

1.3 Cornerstones of UNIX

From its beginning, UNIX was designed around a small set of clever ideas, as Ritchie and Thompson [2] put it:

> “The success of UNIX lies not so much in new inventions but rather in the full exploitation of a carefully selected set of fertile ideas, and especially in showing that they can be keys to the implementation of a small yet powerful operating system.”

Those “fertile ideas” included the design of its file system, its process concept, the concept of privileged and unprivileged programs, the concepts of user and groups, a programmable shell, environments, and device independent input and output. In this section we describe each of these briefly.

1.3.1 Files and the File Hierarchy

Most people who have used computers know what a file is, but as an exercise, try explaining what a file is to your oldest living relative. You may know what it is, but knowing how to define it is another matter. In UNIX, the traditional definition of a file was that "it is the *smallest unit of external storage.*" "External storage" has always meant non-volatile storage, not in primary memory, but on media such as magnetic, optical, and electronic disks, tapes and so on. (Internal storage is on memory chips.) The contemporary definition of a file in UNIX is that it is an object that can be written to, or read from, or both. There is no requirement that it must reside on external storage. We will use this definition of a file in the remainder of these notes.

UNIX organizes files into a tree-like hierarchy that most people erroneously call the *file system*. It is more accurately called the *file hierarchy*, because a file system is something slightly different. The internal nodes of the hierarchy are called *directories*. Directories are special types of files that, from the user perspective, appear to contain other files, although they do not contain files any more than a table of contents in a book contains the chapters of the book themselves. To be precise, a directory is a file that contains *directory entries*. A directory entry is an object that associates a *filename* to a file[5]. Filenames are not the same things as files. The root of the UNIX file system is a directory known in the UNIX world as the *root directory*, however it is not named "`root`" in the file system; it is named "`/`". When you need to refer to this directory, you call it "root", not "slash". More will be said about files, filenames, and the file hierarchy in Section 1.8.

1.3.2 Processes

A *program* is an executable file, and a *process* is an instance of a running program. When a program is run on a computer, it is given various resources such as a primary memory space, both physical and logical, secondary storage space, mappings of various kinds[6], and privileges, such as the right to read or write certain files or devices. As a result, at any instant of time, associated to a process is the collection of all resources allocated to the running program, as well as any other properties and settings that characterize that process, such as the values of the processor's registers. Thus, although the idea of a process sounds like an abstract idea, it is, in fact, a very concrete thing.

UNIX assigns to each process a unique number called its *process-id* or *pid*. For example, at a given instant of time, several people might all be running the Gnu C compiler, `gcc`. Each separate execution instance of `gcc` is a process with its own unique pid. The `ps` command can be used to display which processes are running, and various options to it control what it outputs.

At the programming level, the function `getpid()` returns the process-id of the process that invokes it. The program in Listing 1.3 does nothing other than printing its own process-id, but it illustrates how to use it. Shortly we will see that `getpid()` is an example of a *system call*.

Listing 1.3: A program using `getpid()`.

```
#include <stdio.h>
#include <unistd.h>
int main()
```

[5]In practice a directory entry is an object with two components: the name of a file and a pointer to a structure that contains the attributes of that file.

[6]For example, a map of how its logical addresses map to physical addresses, and a map of where the pieces of its logical address space reside on secondary storage.

```
{
    printf("I am the process with process-id %d\n",
            getpid());

    return 0;
}
```

1.3.3 Users and Groups

Part of the security of UNIX rests on the principle that every user of the system must be *authenticated*. Authentication is a form of security clearance, like passing through a metal detector at an airport.

In UNIX, a *user* is a person[7] who is authorized to use the system. The only way to use a UNIX system is to log into it[8]. UNIX maintains a list of names of users who are allowed to login[9]. These names are called *user-names*. Associated with each user-name is a unique non-negative number called the *user-id*, or *uid* for short. Each user also has an associated password. UNIX uses the user-name/password pair to authenticate a user attempting to login. If that pair is not in its list, the user is rejected. Passwords are stored in an encrypted form in the system's files.

A *group* is a set of users. Just as each user has a user-id, each group has unique integer *group-id*, or *gid* for short. UNIX uses groups to provide a form of resource sharing. For example, a file can be associated with a group, and all users in that group would have the same access rights to that file. Since a program is just an executable file, the same is true of programs; an executable program can be associated with a group so that all members of that group will have the same right to run that program. Every user belongs to at least one group, called the primary group of that user. The `id` command can be used to print the user's user-id and user-name, and the group-id and group-name of all groups to which the user belongs. The `groups` command prints the list of groups to which the user belongs.

In UNIX, there is a distinguished user called the *superuser*, whose user-name is root, one of a few predefined user-names in all UNIX systems. The superuser has the ability to do things that ordinary users cannot do, such as changing a person's user-name or modifying the operating system's configuration. Being able to login as `root` in UNIX confers absolute power to a person over that system. For this reason, all UNIX systems record every attempt to login as `root`, so that the system administrator can monitor and catch break-in attempts.

Every process has an associated (real) user-id and, as we will see later, an effective user-id that might be different from the real user-id. In simplest case, when a user starts up a program, the resulting process has that user's uid as both its real and effective uid. The privileges of the process are the same as those of the user[10]. When the superuser (root) runs a process, that process runs

[7] A user may not be an actual person. It can also be an abstraction of a person. For example, `mail`, `lp`, and `ftp` are each users in a UNIX system, but they are actually programs.

[8] To "login" to a system is to "log into" it. Remember that logging means recording something in a logbook, as a sea captain does. The term "login" conveys the idea that the act is being recorded in a logbook. In UNIX, logins are recorded in a special file that acts like a logbook.

[9] We take this word for granted. We use "login" as a single word only because it has become a single word on millions of "login screens" around the world. To login, as a verb, really means "to log into" something; it requires an indirect object.

[10] To be precise, the privileges are those of user with the process's effective user-id.

with the superuser's privileges. Processes running with user privileges are called user processes. At the programming level, the function `getuid()` returns the real user-id of the process that calls it, and the `getgid()` function returns the real group-id of the process that calls it.

1.3.4 Privileged and Non-Privileged Instructions

In order to prevent ordinary user processes from accessing hardware and performing other operations that may corrupt the state of the computer system, UNIX requires that the processor support two modes of operation, known as *privileged* and *unprivileged* mode[11]. Privileged instructions are those that can alter system resources, directly or indirectly. Examples of privileged instructions include:

- acquiring more memory;
- changing the system time;
- raising the priority of the running process;
- reading from or writing to the disk;
- entering privileged mode.

Only the operating system is allowed to execute privileged instructions. User processes can execute only the unprivileged instructions. The security and reliability of the operating system depend upon this separation of powers.

1.3.5 Environments

When a program is run in UNIX, one of the steps that the operating system takes prior to running the program is to make available to that program an array of name-value pairs called the *environment*. Each name-value pair is a string of the form

```
name=value
```

where `value` is a `NULL`-terminated C string. The `name` is called an *environment variable* and the pair `name=value` is called an *environment string*. The variables by convention contain only uppercase letters, digits, and underscores, but this is not required[12]. The only requirement is that the name does not contain the "=" character. For example, `LOGNAME` is an environment variable that stores the user-name of the current user, and `COLUMNS` is a variable that stores the number of columns in the current console window[13]. Even though it is a number, it is stored as a string.

When a user logs into a UNIX system, the operating system creates the environment for the user, based on various files in the system. From that point forward, whenever a new program runs, it is given a copy of that environment. This will be explained in greater depth later.

[11]These modes are also known as *supervisor mode* and *user mode*.

[12]Environment variable names used by the utilities in the Shell and Utilities volume of POSIX.1-2008 consist solely of uppercase letters, digits, and the underscore ('_') and do not begin with a digit.

[13]If the user defines a value for `COLUMNS` in a start-up script, then terminal windows will have that many columns. If the user does not define it, or sets it to the `NULL` string, the size of terminal windows is determined by the operating system software.

The `printenv` command displays the values of all environment variables as does the `env` command. Within a program the `getenv()` function can be used to retrieve a particular environment string, as in

```
    char* username = getenv("LOGNAME");
    printf("The user's user-name is %s\n, username);
```

The operating system also makes available to every running program an external global variable

```
    extern char **environ;
```

which is a pointer to the start of the array of the name-value pairs in the running program's environment. Programs can read and modify these variables if they choose. For example, a program that needs to know how many columns are in the current terminal window will query the `COLUMNS` variable, whereas other programs may just ignore it.

1.3.6 Shells

The kernel provides services to processes, not to users; users interact with UNIX through a command-line interface called a *shell*. The word "shell" is the UNIX term for a particular type of *command-line-interpreter*. Command-line interpreters have been in operating systems since they were first created. *DOS* uses a command-line-interpreter, as is the *Command* window of Microsoft Windows, which is simply a DOS emulator. The way that DOS and the Command window are used is similar to the way that UNIX is used[14]: you type a command and press the *Enter* key, and the command is executed, after which the prompt reappears. The program that displays the prompt, reads the input you type, runs the appropriate programs in response and re-displays the prompt is the command-line-interpreter, which in UNIX is called a shell.

In UNIX, a shell is much more than a command-line-interpreter. It can do more than just read simple commands and execute them. A shell is also programming language interpreter; it allows the user to define variables, evaluate expressions, use conditional control-of-flow statements such as `while-` and `if`-statements, and make calls to other programs. A sequence of shell commands can be saved into a file and executed as a program by typing the name of the file. Such a sequence of shell commands is called a *shell script*. When a shell script is run, the operating system starts up a shell process to read the instructions and execute them.

1.3.7 Online Documentation: The Man Pages

Shortly after Ritchie and Thompson wrote the first version of UNIX, at the insistence of their manager, Doug McIlroy, in 1971, they wrote the *UNIX Programmer's Manual*. This manual was initially a single volume, but in short course it was extended into a set of seven volumes, organized by topic. It existed in both printed form and as formatted files for display on an ordinary character display device. Over time it grew in size. Every UNIX distribution comes with this set of manual pages, called "manpages" for short. Appendix B.4 contains a brief description of the structure of the manpages, and Chapter 2 provides more detail about how to use them.

[14]This is not a coincidence. Long before Microsoft wrote MS-DOS, they wrote a version of UNIX for the PC called *Xenix*, whose rights they sold to Santa Cruz Operations in 1987

1.4 The UNIX Kernel API

A multi-user operating system such as UNIX must manage and protect all of the system's resources and provide an operating environment that allows all users to work efficiently, safely, and happily. It must prevent users and the processes that they invoke from accessing any hardware resources directly. In other words, if a user's process wants to read from or write to a disk, it must ask the operating system to do this on its behalf, rather than doing it on its own. The operating system will perform the task and transfer any data to or from the user's process. To see why this is necessary, consider what would happen if a user's process could access the hard disk directly. A user could write a program that tried to acquire all disk space, or even worse, tried to erase the disk.

A program such as the ones in Listings 1.1 and 1.2, may look like it does not ask the operating system to read or write any data, but that is not true. Both `getchar()` and `putchar()`, are library functions in the C Standard I/O Library (whose header file is `<stdio.h>`), and they do, in fact, "ask" the operating system to do the work for the calling program. The details will be explained later, but take it on faith that one way or another, the operating system has intervened in this task.

The operating system must protect users from each other and protect itself from users. However, while providing an operating environment for all users, a multi-user operating system gives each user the impression that he or she has the computer entirely to him or herself. This is precisely the illusion underlying the execution of the program in Figure 1.1. Somehow everyone is able to write programs that look like they have the computer all to themselves, and run them as if no one else is using the machine. The operating system creates this illusion by creating data paths between user processes and devices and files. The data paths connect user processes and devices in the part of memory reserved for the operating system itself. And that is the first clue – physical memory is divided into two regions, one in which ordinary user programs are loaded, called *user space*, and one where the operating system itself is stored, called *system space*.

How does UNIX create this illusion? We begin with a superficial answer, and gradually add details in later chapters.

The UNIX operating system is called the *kernel*. The kernel defines the application programming interface and provides all of UNIX's services, whether directly or indirectly. The kernel is a program, or a collection of interacting programs, depending on the particular implementation of UNIX, with many *entry points*[15]. Each of these entry points provides a service that the kernel performs. If you are used to thinking of programs as always starting at their first line, this may be disconcerting. The UNIX kernel, like many other programs, can be entered at other points. You can think of these entry points as functions that can be called by other programs. These functions do things such as opening, reading, and writing files, creating new processes, allocating memory, and so on. Each of these functions expects a certain number of arguments of certain types, and produces well-defined results. The collection of kernel entry points makes up a large part of UNIX's API. You can think of the kernel as a collection of separate functions, bundled together into a large package, and its API as the collection of signatures or prototypes of these functions.

When UNIX boots, the kernel is loaded into the portion of memory called system space and stays there until the machine is shut down. User processes are not allowed to access system space. If they do, they are terminated by the kernel.

[15]An entry point is an instruction in a program at which execution can begin. In the programs that you have probably written, there has been a single entry point – `main()` –, but in other programs, you can specify that the code can be entered at any of several entry points. Software libraries are code modules with multiple entries points. In the Windows world, dynamically linked libraries (DLLs) are examples of code modules with multiple entry points.

The kernel has full access to all of the hardware attached to the computer. User programs do not; they interact with the hardware indirectly through the kernel. The kernel maintains various system resources in order to provide these services to user programs. These system resources include many different data structures that keep track of I/O, memory, and device usage for example. In Section 1.4.1 this is explained in more detail.

Summarizing, if a user process needs data from the disk for example, it has to "ask" the kernel to get it. If a user process needs to write to the display, it has to "ask" the kernel to do this too. All processes gain access to devices and resources through the kernel. The kernel uses its resources to provide these services to user processes.

1.4.1 System Resources

The kernel provides many services to user programs, including

- process scheduling and management,
- I/O handling,
- physical and virtual memory management,
- device management,
- file management,
- signaling and inter-process communication,
- multi-threading,
- multi-tasking,
- real-time signaling and scheduling, and
- networking services.

Network services include protocols such as HTTP, NIS, NFS, X.25, SSH, SFTP, TCP/IP, and Java. Exactly which protocols are supported is not important; what is important is for you to understand that the kernel provides the means by which a user program can make requests for these services.

There are two different methods by which a program can make requests for services from the kernel:

- by making a *system call* to a function (i.e., entry point) built directly into the kernel, or
- by calling a higher-level *library routine* that makes use of this call.

Do not confuse either of these with a *system program*. The term "system program" refers to a separate program that is bundled with the kernel, that interfaces to it to achieve its functionality, and that provides higher level services to users. We can browse through the `/bin` or `/usr/bin` directories of a UNIX installation to find many different system programs. Many UNIX commands are implemented by system programs.

1.4.2 System Calls

An ordinary function call is a jump to and return from a subroutine that is part of the code linked into the program making the call, regardless of whether the subroutine is *statically* or *dynamically* linked into the code. A system call is like a conventional function call in that it causes a jump to a subroutine followed by a return to the caller. But it is significantly different because it is a call to a function that is a part of the UNIX kernel.

The code that is executed during the call is actually kernel code. Since the kernel code accesses hardware and contains privileged instructions, kernel code must be run in privileged mode. Because the kernel alone runs in privileged mode, it is also commonly called *kernel mode* or *superuser mode*. Therefore, during a system call, the process that made the call is run in kernel mode. Unlike an ordinary function call, a system call requires a change in the execution mode of the processor; this is usually implemented by a *trap instruction*. The trap is typically invoked with special parameters that specify which system call to run. The method of implementing this is system dependent. In all cases, the point at which a user process begins to execute kernel code is a perilous point, and great care must be taken by operating system designers and the programmers who code the system call interfaces to make sure that it cannot be booby-trapped by malicious programmers trying to get their programs to run in kernel mode.

Programs do not usually invoke system calls directly. The C library provides *wrappers* for almost all system calls, and these usually have the same name as the call itself. A wrapper for a function `f` is a function that does little more than invoking `f`, usually rearranging or pre-processing its arguments, checking error conditions, and collecting its return value and possibly supplying it in a different form to the caller. Wrappers for system calls also have to trap into kernel mode before the call and restore user mode after the call. A wrapper is *thin* if it does almost nothing but pass through the arguments and the return values. Often, for example, the GNU C library wrapper function is very thin, doing little work other than copying arguments to the right registers before invoking the system call, and then setting the value of a global error variable[16] appropriately after the system call has returned.

Sometimes a wrapper is not so thin, as when the library function has to decide which of several alternative functions to invoke, depending upon what is available in the kernel. For example. there is a system call named `truncate()`, which can "crop" a file to a specified length, discarding the data after that length. The original `truncate()` function could only handle lengths that could fit into a 32-bit integer, and when file systems were able to support very large files, a newer version named `truncate64()` was developed. The latter function can handle lengths representable in 64 bits in size. The wrapper for `truncate()` decides which one is provided by the kernel and calls it appropriately.

There may be system calls that do not have wrappers in the library, and for these, the programmer has no other choice but to invoke the system call with a special function named `syscall()`, passing the system call's identification number and arguments. Every system call has a unique number associated to it. Generally speaking, for a system call named `foo`, its number is defined by a macro named either `__NR_foo` or `SYS_foo`. The macro definitions are included by including the header file `<sys/syscall.h>` in the code. They may not be in that file itself; they may be another file, such as `<asm/unistd_32.h>` or `<asm/unistd_64.h>`. An example of a system call without a wrapper is

[16]To be precise, the variable is named `errno` and it has *thread local storage*, which means each thread has its own unique copy and the lifetime of this variable is the entire lifetime of the thread.

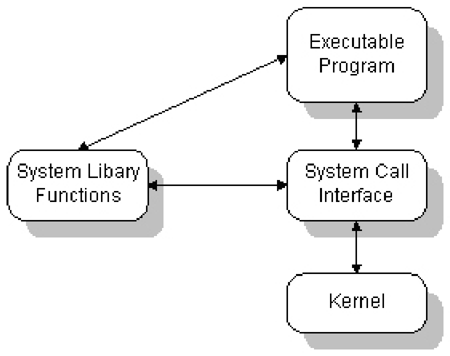


Figure 1.3: System calls versus system libraries.

`gettid()`, which returns the calling thread's thread id. It is the same as `getpid()` for a process with a single thread. The following program calls `gettid()` and prints the returned id on the screen:

```
#define _GNU_SOURCE
#include <unistd.h>
#include <sys/syscall.h>
#include <sys/types.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
    printf("Thread id %ld\n", syscall(SYS_gettid));
    /* could also pass __NR_gettid */
    return 0;
}
```

Because the function has no arguments, it is not necessary to pass anything other than the system call number to `syscall()`.

The complete set of system calls is usually available in the `syscalls` manpage in Section 2.

1.4.3 System Libraries

Many system calls are very low-level primitives; they do very simple tasks. This is because the UNIX operating system was designed to keep the kernel as small as possible. Also for this reason, the kernel typically does not provide many different kinds of routines to do similar things. For example, there is only a single kernel function to perform a read operation, and it reads large blocks of data from a specified device to specified system buffers. There is no system call to read a character at a time, which is a useful function to have. In short, there is a single kernel function that performs input operations!

To compensate for the austerity of the kernel, UNIX designers augmented the programming interface with an extensive set of higher-level routines that are kept in system libraries. These routines provide a much richer set of primitives for programming the kernel. Of course, they ultimately make calls to the kernel, as depicted in Figure 1.3. UNIX also contains libraries for various specialized tasks, such as asynchronous input and output, shared memory, terminal control, login and logout management, and so on. Using any of these libraries requires that the library's header file be included in the code with the appropriate `#include` directive (e.g. `#include <termios.h>`), and sometimes, that the library be linked explicitly because it is not in a standard place. Manpages for functions that are part of system libraries are contained in Volume 3 of the UNIX Manual Pages.

1.5 UNIX and Related Standards

1.5.1 The Problem

The very first version of UNIX was written by Ken Thompson and Dennis Ritchie in 1969 while they were working for AT&T Bell Labs. Their fledgling operating system turned out to be full of very novel ideas, and they presented these ideas in a seminal paper at the ACM Symposium on Operating Systems at IBM Yorktown Heights in 1973. In January 1974, the University of California at Berkeley (UCB) acquired Version 4 from Bell Labs and embarked on a mission to add modern features to UNIX. Later that year, AT&T began licensing UNIX to universities. From 1974 to 1979, UCB and AT&T worked on independent copies of UNIX. By 1978, the various versions of UNIX had most of the features that are found in it today, but not all in one system. But in 1979, AT&T did something that changed the playing field; they staked proprietary rights to their own brand of UNIX, selling it commercially. In essence, they trademarked UNIX and made it expensive to own it.

BSD code was released under a much more generous license than AT&T's source and did not require a license fee or a requirement to be distributed with source unlike the GPL that the GNU Project and Linux use today. The result was that much BSD source code was incorporated into various commercial UNIX variants. By the time that 4.3BSD was written, almost none of the original AT&T source code was left in it. FreeBSD/NetBSD/OpenBSD were all forks of 4.3BSD having none of the original AT&T source code, and no right to the UNIX trademark, but much of their code found its way into commercial UNIX operating systems. In short, two major versions of UNIX came into existence – those based on BSD and those based on the AT&T version.

In 1991, the picture was further complicated by the creation of Linux. Linux was developed from scratch unlike BSD and it used the existing GNU Project which was a clean-room implementation of much of the UNIX user-space. It is a lot less like the AT&T UNIX than BSD is. In 1993, AT&T divested itself of UNIX, selling it to Novell, which one year later sold the trademark to an industry consortium known as X/Open.

There are now dozens of different UNIX distributions, each with its own different behavior. There are systems such as Solaris and UnixWare that are based on SVR4, the AT&T version released in 1989, and FreeBSD and OpenBSD based on the UC Berkeley distributions. Systems such as Linux are hybrids, as are AIX, IRIX, and HP-UX. It is natural to ask what makes a system UNIX. The answer is that over the course of the past thirty years or so, standards have been developed in order to define UNIX. Operating systems can be branded as conforming to one standard or another.

1.5.2 The Solution: Standards

One widely accepted UNIX standard is the *POSIX* standard. Technically, POSIX does not define UNIX in particular; it is more general than that. POSIX, which stands for *Portable Operating*

System Interface, is a family of standards known formally as *IEEE 1003*. It is also published by the *International Standards Organization* (*ISO*) as *ISO/IEC 9945:2003*; these are one and the same document. The most recent version of POSIX is *IEEE Std 1003.1-2008*, also known as *POSIX.1-2008*. The POSIX.1-2008 standard consolidates the major standards preceding it, including POSIX.1, and the *Single UNIX Specification* (*SUS*). The spirit of POSIX is to define a UNIX system, as is stated in the Introduction to the specification (http://pubs.opengroup.org/onlinepubs/9699919799/):

> The intended audience for POSIX.1-2008 is all persons concerned with an industry-wide standard operating system based on the UNIX system. This includes at least four groups of people:
>
> Persons buying hardware and software systems
>
> Persons managing companies that are deciding on future corporate computing directions
>
> Persons implementing operating systems, and especially
>
> Persons developing applications where portability is an objective

The Single UNIX Specification was derived from an earlier standard written in 1994 known as the *X/Open System Interface* which itself was developed around a UNIX portability guide called the *Spec 1170 Initiative*, so called because it contained a description of exactly 1,170 distinct system calls, headers, commands, and utilities covered in the spec. The number of standardized elements has grown since then, as UNIX has grown.

The Single UNIX Specification was revised in 1997, 2001, and 2003 by *The Open Group*, which was formed in 1996 as a merger of *X/Open* and the *Open Software Foundation* (*OSF*), both industry consortia. The Open Group owns the UNIX trademark. It uses the Single UNIX Specification to define the interfaces an implementation must support to call itself a UNIX system.

What, then, does this standard standardize? It standardizes a number of things, including the collection of all system calls, the system libraries, and those utility programs such as `grep`, `awk`, and `sed` that make UNIX feel like UNIX. The collection of system calls is what defines the UNIX kernel. The system calls and system libraries together constitute the UNIX application programming interface. They are the programmer's view of the kernel. The utility programs are the part of the interface that the UNIX user sees.

From the Introduction again:

> POSIX.1-2008 is simultaneously IEEE Std 1003.1™-2008 and The Open Group Technical Standard Base Specifications, Issue 7.
>
> POSIX.1-2008 defines a standard operating system interface and environment, including a command interpreter (or "shell"), and common utility programs to support applications portability at the source code level. POSIX.1-2008 is intended to be used by both application developers and system implementers [sic] and comprises four major components (each in an associated volume):
>
> - General terms, concepts, and interfaces common to all volumes of this standard, including utility conventions and C-language header definitions, are included in the Base Definitions volume.

- Definitions for system service functions and subroutines, language-specific system services for the C programming language, function issues, including portability, error handling, and error recovery, are included in the System Interfaces volume.
- Definitions for a standard source code-level interface to command interpretation services (a "shell") and common utility programs for application programs are included in the Shell and Utilities volume.
- Extended rationale that did not fit well into the rest of the document structure, which contains historical information concerning the contents of POSIX.1-2008 and why features were included or discarded by the standard developers, is included in the Rationale (Informative) volume.

POSIX.1-2008 specifically defines certain areas as being outside of its scope:

- Graphics interfaces
- Database management system interfaces
- Record I/O considerations
- Object or binary code portability
- System configuration and resource availability

In summary, the Single UNIX Specification, Version 4, known as SUSv4, also known as The Open Group Specification Issue 7, consists of four parts: a base definition, detailed system interfaces, shell and utilities, and rationale, which describes reasons for everything else.

The fact that there are standards does not imply that all UNIX implementations adhere to them. Although there are systems such as AIX, Solaris, and Mac OS X that are fully POSIX-compliant, most are "mostly" compliant. Systems such as FreeBSD and various versions of Linux fall into this category.

Any single UNIX system may have features and interfaces that do not comply with a standard. The challenge in system programming is being able to write programs that will run across a broad range of systems in spite of this. Later we will see how the use of *feature test macros* in programs provides a means to compile a single program on a variety of different UNIX systems.

1.6 The C Library and C Standards

The interfaces described in the POSIX standard are written in C[17], mostly because C is the major language in which most systems programs are written, and because much of UNIX was originally developed in C. Because of this, POSIX depends upon a standard definition of C, and it uses the ISO standard, the most recent version of which is officially known as ISO/IEC 9899:2011, and informally known as C11. C11 incorporated the earlier ANSI C and augmented it. This version of C is known as ISO C, but people also continue to call it ANSI C, even though it is not the same thing. You can download the last free draft of this standard from C11 Standard (pdf)

In short, POSIX specifies not just what UNIX must do, but what the various parts of the C Standard Library must do as well. It specifies, in effect, a superset of the C language, including additional

[17]There are language bindings of the kernel API in *Fortran*, *Java*, *Python*, *Pascal*, *C++*, and other languages.

functions to those introduced in standard C. Therefore, a UNIX system that is POSIX compliant contains all of the library functions of the ISO C language. For example, every UNIX distribution includes libraries such as the C Standard I/O Library, the C math library, and the C string library.

The C Standard Library provided for Linux as well as several other UNIX distributions is the GNU C library, called GNU `libc`, or `glibc`. GNU often extends the C library, and not everything in it conforms to the ISO standard, nor to POSIX. What all of this amounts to is that the version of the C library on one system is not necessarily the same as that found on another system.

This is one reason why it is important to know the standard and know what it defines and what it does not define. In general, the C standard describes what is required, what is prohibited, and what is allowed within certain limits. Specifically, it describes

- the representation of C programs
- the syntax and constraints of the C language
- the semantic rules for interpreting C programs
- the representation of input data to be processed by C programs
- the representation of output data produced by C programs
- the restrictions and limits imposed by a conforming implementation of C

Not all compilers and C runtime libraries comply with the standard, and this complicates programming in C. The GNU compiler has command line options that let you compile according to various standards. For example, if you want your program to be compiled against the ANSI standard, you would use the command

```
$ gcc -ansi
```

or

```
$ gcc -std=c90
```

To use the current ISO C11 standard, either of these works:

```
$ gcc -std=c11
$ gcc -std=iso9899:2011
```

Understanding how to write programs for UNIX requires knowing which features are part of C and which are there because they are part of UNIX. In other words, you will need to understand what the C libraries do and what the underlying UNIX system defines. Having a good grasp of the C standard will make this easier.

1.7 Learning System Programming by Example

The number of system calls and library functions is so large that mere mortals cannot remember them all. Trying to learn all of the intricacies and details of the UNIX API by reading through reference manuals and user documentation would be a painstaking task. Fortunately, people often learn well by example. Rather than studying the reference manuals and documentation, we can learn the API little by little by writing programs that use it. One starts out simple, and over time adds complexity.

Bruce Molay [1] uses an excellent strategy for learning how to write system programs and discover the UNIX API:

1. Pick an existing program that uses the API, such as a shell command;

2. Using the system man pages and other information available online, investigate the system calls and kernel data structures that this program most likely uses in its implementation; and

3. Write a new version of the program, iteratively improving it until it behaves just like the actual command.

By repeating this procedure over and over, one can familiarize oneself with the relevant portions of the API as well as the resources needed to learn about it. When it is time to write a full-fledged application, the portions of it that must communicate with the kernel should be relatively easy to write. We will partly follow this same paradigm in this sequence of notes on UNIX.

1.8 The UNIX File Hierarchy

The typical UNIX file hierarchy has several directories just under the root. Figure 1.4 shows part of a typical, but hypothetical, file hierarchy. The following directories should be present in most UNIX systems, but they are not all required. The only required directories are `/dev` and `/tmp`.

Directory	**Purpose**
`bin`	The repository for all essential binary executables including those shell commands that must be available when the computer is running in "single-user mode" (something like safe mode in Windows.)
`boot`	Static files of the boot loader
`dev`	The directory containing essential device files, which will be explained later.
`etc`	Where almost all host configuration files are stored. It is something like the registry file of Windows.
`home`	The directory where all user home directories are located, but not always.
`lib`	Essential shared libraries and kernel modules
`media`	Mount point for removable media
`mnt`	Mount point for mounting a file system temporarily
`opt`	Add-on application software packages
`sbin`	Essential system binaries
`srv`	Data for services provided by this system
`tmp`	Temporary files

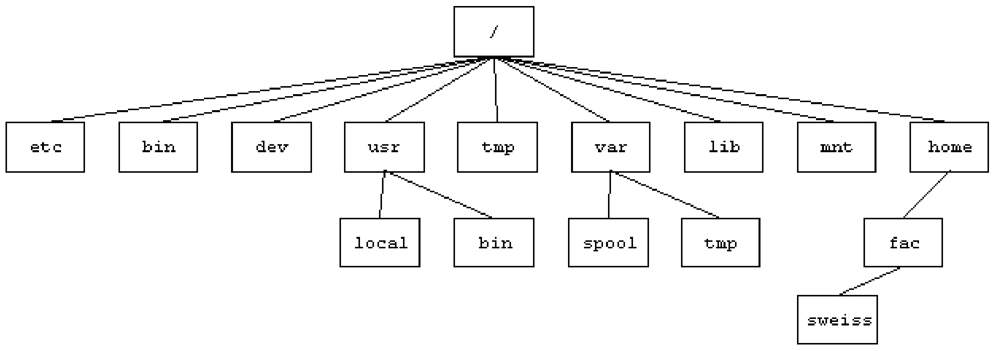


Figure 1.4: The top of a typical UNIX directory hierarchy.

Directory	**Purpose**
`usr`	Originally, `/usr` was the top of the hierarchy of user "data" files, but now it serves as the top of a hierarchy in which non-essential binaries, libraries, and sources are stored. Below `/usr` are directories such as `/usr/bin` and `/usr/sbin`, containing binaries, `/usr/lib`, containing library files, and `/usr/local`, the top of a third level of "local" programs and data.
`var`	Variable files (files containing data that can change)

Files have two independent binary properties: *shareable* vs. *unshareable* and *variable* vs. *static*. In general, modern UNIX systems are encouraged to put files that differ in either of these respects into different directories. This makes it easy to store files with different usage characteristics on different file systems.

Shareable files are those that can be stored on one host and used on others. Unshareable files are those that are not shareable. For example, the files in user home directories are shareable whereas boot loader files are not.

Static files include binaries, libraries, documentation files and other files that do not change without system administrator intervention. "Variable" files are files that are not static. The `/etc` directory should be unshareable and static. The `/var` directory is variable but parts of it, such as `/var/mail` may be shareable while others such as `/var/log` may be unshareable. `/usr` is shareable and static.

1.8.1 About Pathnames and Directories

A *pathname* is a character string that is used to identify a file. POSIX.1-2008, puts a system-dependent limit on the number of bytes in a pathname, including the terminating null byte[18].

There are two types of pathnames: *absolute* and *relative*. An absolute pathname is a pathname that starts at the root. It begins with a "/" and is followed by zero or more filenames separated by "/" characters. All filenames except the last must be directory names[19]. For exam-

[18] The variable `PATH_MAX` contains the maximum length of a string representing a pathname. The library function `pathconf()` can be used to obtain its value. On many Linux systems it is 4096 bytes.

[19] This is not exactly true. Filenames that are not the last in the pathname may be symbolic links to directories.

ple, `/home/fac/sweiss` is the absolute pathname to the directory `sweiss` in Figure 1.4, as is `/home//fac///sweiss`. The extra slashes are ignored. Observe that UNIX (actually POSIX) uses slashes, not backslashes, in pathnames: `/usr/local`, not `\usr\local`.

If a pathname does not begin with "/" it is called a *relative pathname*. When a process must resolve a relative pathname so that it can access the file, the pathname is assumed to start in the *current working directory*. In fact, the definition of the current working directory, also called the *present working directory*, is that it is the directory that a process uses to resolve pathnames that do not begin with a "/". For example, if the current working directory is `/home/fac`, then the pathname `sweiss/testdata` refers to a file whose absolute pathname is `/home/fac/sweiss/testdata`. The convention is to use *pwd* as a shorthand for the current working directory.

The environment variable `PWD` contains the absolute pathname of the current working directory. The command `pwd` prints the value of the `PWD`.

1.8.1.1 Working with Directories

This section may be skipped if you have experience working with directories at the user level. You do not need to know many commands in order to do most directory-related tasks in UNIX. This is a list of the basic commands. The principal tasks are navigation and displaying and altering their contents. The tables that follow give the command name and the simplest usage of it. They do not describe the various options or details of the command's usage.

Command	**Explanation**
`pwd`	print the path of the current working directory (pwd)
`ls [<dir1>] [<dir2>] ...`	list the contents of the *pwd*, or `<dir1>, <dir2> ...` if supplied
`cd [<dir>]`	change *pwd* to `HOME` directory, or `<dir>` if it is supplied
`mkdir <dir> [<dir2>] ...`	create new directories `<dir>` (and `<dir2>` ...) in the pwd;
`rmdir <dir> [<dir2>] ...`	remove the EMPTY directory `<dir>` (and `<dir2>` ...)
`rm -r <dir>`	remove all contents of `<dir>` and `<dir>` itself. Dangerous!!
`mv`	see the explanation in Section 1.8.2

Notes

1. The "`p`" in "`pwd`" stands for print, but it does not print on a printer. In UNIX, "printing" means displaying on the screen[20].
2. `mkdir` is the only way to create a directory
3. You cannot use `rmdir` to delete a directory if it is not empty.
4. You can delete multiple directories and their contents with `rm -r`, but this is not reversible, so be careful.
5. Commands that create and delete files are technically modifying directories, but these will be covered separately.

[20]That is why the C instruction `printf` sends output to the display device, not the printer. In FORTRAN, by contrast, the `print` instruction sent output to the printer.

"Changing directories" and "being in a directory" are imprecise phrases. When you `cd` to a directory named `dir`, you may think of yourself as being “in `dir`”, but this is not true. What is true is that the `dir` directory is now your current working directory and that every process that you run from the shell process in which you changed directory, including the shell process, will use this `dir` directory by default when it is trying to resolve relative pathnames.

There are two special entries that are defined in every directory

`.`	The directory itself
`..`	The parent directory of the directory, or itself if it is /

Thus, "`cd ..`" changes the *pwd* to the parent of the current *pwd*, "`cd ../..`" changes it to the grandparent and "`cd .`" has no effect. Before reading further, you should experiment with these commands and make sure that you understand how they work.

1.8.2 Files and Filenames

Unlike other operating systems, UNIX distinguishes between only five types of non-directory files:

- regular files
- device files (character or block)
- FIFOs
- sockets
- symbolic links

Device files and FIFOs will be described in depth in Chapter 4, sockets in Chapter 9, and symbolic links below. That leaves regular files, which are defined quite simply: a regular file is a sequence of bytes with no particular structure.

1.8.2.1 Filenames

Files and filenames, as noted earlier, are different things. A filename, which is technically called a *file link*, or just a *link*, is just a string that names a file[21]. A file may have many filenames. Filenames are practically unlimited in size, unless you think 255 characters is not big enough. The maximum number of bytes in a filename is contained in the system-dependent constant `NAME_MAX`. Filenames can contain any characters except "`/`" and the null character. They can have spaces and new lines, but if they do, you will usually need to put quotes around the name to use it as an argument to commands. UNIX is *case-sensitive*, so "References" and "references" are two different filenames.

Unlike DOS, Windows, and Apple operating systems, filename extensions are not used by the operating system for any purpose, although application-level software such as compilers and word

[21] A filename is sometimes referred to as a "pathname component".

processors use them as guides, and desktop environments such as Gnome and KDE create associations based on filename extensions in much the same way that Windows and Apple do. But again, UNIX itself does not have a notion of file type based on content, and it provides the same set of operations for all files, regardless of their type.

Remember that a directory entry consists of two parts. One part is a filename and the other part is a means by which a file is associated with this name. You may think of this second part as an index into a table of pointers to the actual files[22].

One file can have many links, like a spy traveling with several forged passports. In one directory, the file may be known by one link, and in a different directory, it may have another link. It is still the same file though. In Figure 1.5, the file is known by three names, each a link in a different directory. There is two restrictions concerning multiple links. One is that directories cannot have multiple names. Another is that two names for the same file cannot exist on different file systems. The simplest way to understand this last point, without going into a discussion of mounting and mount points, is that different parts of the file hierarchy can reside on different physical devices, e.g., different partitions of a disk or different disks, and that a file on one physical device cannot have a name on a different physical device. This will be clarified in a later chapter.

1.8.2.2 What is a File?

All files, regardless of their type, have *attributes*. Almost all have *data*, or *content*.

- Attributes include properties such as the time the file was created, the time it was last modified, the file size expressed as a number of bytes, the number of disk blocks allocated to the file, and so on. The attributes that describe restrictions on access to the file are called the *file mode*. The attributes of a file collectively are called the *file status* in UNIX. The word "status" may sound misleading, but it is the word that was used by the original designers of UNIX.
- The data are the actual file contents. In fact, UNIX uses the term *content* to identify this part of a file. Some files do not have any content because they are merely interfaces that the operating system uses. That will be discussed further in Chapter 4.

These two items, the status and the content, are not stored together[23].

1.8.2.3 Working with Files

This section may be skipped if you have user level experience in UNIX. Commands for working with files can be classified as those that view file contents but do not modify them, those that view file attributes, and editors – those that modify the files in one way or another. Listed below are the basic commands for viewing the contents of files, not editing or modifying them. Editors are a separate topic. In all cases, if the file arguments are omitted, the command reads from standard input. There is also a separate class of programs called filters that provide sophisticated filtering of ordinary text files.

[22]Traditional UNIX system used an index number, called an *inumber*, to associate the file to the filename. This number was used to access a pointer to a structure called an *inode*, to be discussed later. The inode contains members that point to the file contents. Conceptually, the inode is like a proxy for the file.

[23]The status is stored in the inode.

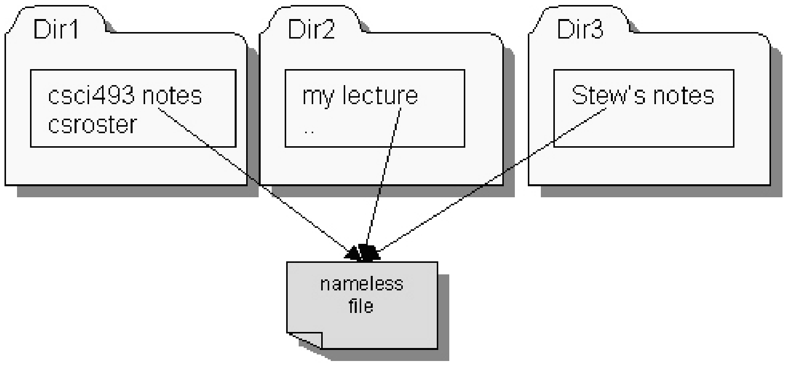


Figure 1.5: Example of multiple links to a file.

Viewing File Contents

Command	Explanation
`cat [<files>]`	display file contents
`more [<files>]`	display file contents a screen at a time
`less [<files>]`	display file contents a screen at a time with more options
`pg  [<files>]`	display file contents a screen at a time.
`head [-<n>] [<file>]`	display the first `<n>` lines of a file, default n = 10
`tail [-<n>] [<file>]`	display the last `<n>` lines of a file, default n = 10

Note that the `pg` command is not POSIX and may not be available on all UNIX systems.

A note about `more` and `less`: the `more` command existed long before there was a `less` command. `more` was created to allow you to read a page at a time, and also to jump ahead in the file with regular expression searches, using the same "/" operator from `vi`. However, it was not easy to go backwards in the file with `more`. You could use the single quote operator (') to go back to the beginning of the file, but that was about it. `less` was created to do more than `more`, naturally, which is why it is called `less`. Maybe the person who wrote it knew the famous quotation from Ludwig Mies Van der Rohe, the German-born architect, whose famous adage, "Less is more," is immortalized. Maybe not. But for whatever reason that they named `less`, "`less`", `less` does more than `more`. And this is not an Abbott and Costello skit. The `less` command is the most versatile means of viewing files, and my recommendation is that you learn how to use it first.

Creating, Removing, and Copying Files and Links Remember that a link is just a name pointing to a file.

Command	Explanation
`ln  <f1> <f2>`	create a new link for file `<f1>` named `<f2>`
`rm <files>`	delete a link or name for a file
`mv <file1> <file2>`	rename `<file1>` with the new name `<file2>`
`mv <files> <dir>`	move all files into the destination directory `<dir>`
`cp <file1> <file2>`	copy `<file1>` to the new name `<file2>`
`cp <files> <dir>`	copy all files into the destination directory `<dir>`

Notes.

- The `ln` command can only be used on files, not directories. There is a simplified version of this command named `link`.
- The `rm` command is irreversible; once a link is removed, it cannot be recovered.
- The `mv` command has two forms. If there are more than two arguments, the last must be an existing directory name, and all files named before the last are moved into that directory. Conversely, if the last argument is an existing directory name, then all arguments preceding it will be moved into it. The preceding arguments can be existing directories, as long as it does not create a circularity, such as trying to make a child into a parent. If the last argument is not an existing directory, the meaning of `mv` is simply to rename the first file with the new name. If the new name is an existing file, `mv` will silently overwrite it. For this reason, you should use "`mv -i`", which prompts you before overwriting files.
- The way you use the `cp` command is almost the same as how you use the `mv` command except that it replicates files instead of moving them. The main difference is that `cp` does not accept directories as arguments, except as the last argument, unless you give it the option "`-r`", i.e., `cp -r`, in which case it recursively copies the directories. Note that `cp` makes copies of files, not just their names, so changes to one file are not reflected in the other.

Examples

```
$ ls
hwk1_6.c
$ mv hwk1_6.c hwk1_finalversion.c
$ ls
hwk1_finalversion.c
```

This command changed the link `hwk1_6.c` to `hwk1_finalversion.c`.

```
$ rm hwk1.o hwk1.old.o main.o
```

This removes these three file links from the current working directory. If these files have no other links in other directories, the attributes and contents are also removed from the disk. I will explain more about this in a later chapter.

```
$ ln hwk1.1.c ../mysourcefiles/proj1.c
```

creates a new link `proj1.c` in the directory `../mysourcefiles` for the file whose name is `hwk1.1.c` in the current working directory.

```
$ cp -r main.c utils.c utils.h images ../version2
```

copies the three files `main.c`, `utils.c`, and `utils.h`, and the directory `images`, into the directory `../version2`.

1.8.2.4 File Attributes

In this section, the word "file" refers to all file types, including directories. UNIX has a very simple, but useful, file protection method. To provide a way for users to control access to their files, the inventors of UNIX devised a rather elegant and simple access control system. Every file has an owner, called its *user*. The file is also associated with one of the groups to which the owner belongs, called its *group*. The owner of a file can make the file's group any of the groups to which the owner belongs. Lastly, everyone who is neither the user nor a member of the file's group is in the class known as *others*. Thus, the set of all users is partitioned into three disjoint subsets: user, group, others, which you can remember with the acronym *ugo*.

There are three modes of access to any file: read, write, and execute. *Read access* is the ability to view file contents. For directories, this is the ability to view the contents using the ls command. *Write access* is the ability to change file contents or certain file attributes. For directories, this implies the ability to create new links in the directory, to rename files in the directory, or remove links in the directory. This will be counterintuitive – the ability to delete a file from a directory does not depend on whether one has write privileges for the file, but on whether one has write privileges for the directory. *Execute access* is the ability to run the file. For a directory, execute access is the ability to `cd` into the directory and as a result, the ability to run programs contained in the the directory and run programs that need to access the attributes or contents of files within that directory. In short, without execute access on a directory, there is little you can do with it.

For each of the three classes of users, there are three protection bits that define the read, write, and execute privileges afforded to members of the class. For each class, if a specific protection bit is set, then for anyone in that class, the particular type of access is permitted. Thus, there are three bits called the **r**ead, **w**rite, and e**x**ecute bits for the user (`u`), for the group (`g`), and for others (`o`), or nine in total. These bits, which are called *mode* or *permission bits*, are usually expressed in one of two forms: as an octal number, or as a string of nine characters.

The nine-character permission bit string is:

```
r w x      r w x       r w x
user bits  group bits  others bits
```

Various commands use a dash is used to indicate that the bit is turned off.

Examples

- The string `rwxrw-r--`, gives the user (owner) read, write, and execute permission, the group, read and write but no execute, and others, only read permission.
- The string `r-xr-xr-x` gives everyone only read and execute permission.
- The string `rwxr-xr--` gives the user read, write, and execute permission, the group, read and execute permission, and others, only read access.

The mode string can be represented as a 3-digit octal number, by treating each group of three bits as a single octal digit. Using the C ternary operator, `?:`, I would write this as

```
value = r?4:0 + w?2:0 + x?1:0
```

which results in the following table of values for each group of three bits.

rwx	rw-	r-x	r--	-wx	-w-	--x	---
7	6	5	4	3	2	1	0

For example, the octal number for `rwxrw-r--` is 764 because the user digit is 7, the group is 6 and the others is 4.

In addition to the mode bits, a file's permission string is usually displayed with a single character file attribute that characterizes the file type. The character, which appears to the left of the mode bits, can be one of `-` (regular file), `d` (directory), `b` (buffered special file), `c` (character special file), `l` (symbolic link), `p` (pipe), or `s` (socket).

1.8.2.5 Viewing and Modifying File Attributes

To see the attributes of a file, use the `-l` option to the `ls` command: The "`-l`" means "long listing" and it prints the permission string, the number of links to the file, the user-name of the owner, the group name of the group, the number of bytes in the file, the last modification time, and the file link. For example, to look at the attributes of the file named .bashrc in the current working directory, I would type

```
$ ls -l ~/.bashrc
-rw-r--r-- 1 sweiss faculty 3304 Sep 22 13:05 .bashrc
```

This file is a regular file and can be read and modified by me, its owner (`-rw-`). It can be read by anyone in the faculty group (`r--`), and it can be read by anyone else (`r--`). `ls` has many other options, for displaying additional information. Read its man page for details. You can also use the `stat` command, which will provide additional information:

```
$ stat .bashrc
  File: '.bashrc'
  Size: 3304     Blocks: 8   IO Block: 4096   regular file
Device: 18h/24d Inode: 1318        Links: 1
Access:(0644/-rw-r--r--)Uid:(1220/ sweiss)Gid:(400/ faculty)
Access: 2010-12-20 13:20:04.582733000 -0500
Modify: 2010-09-22 13:05:11.271251000 -0400
Change: 2010-09-22 13:05:11.278893000 -0400
```

You can get the count of bytes, words, and lines with `wc`:

```
$ wc .bashrc
156 387 3304 .bashrc
```

There are 156 lines, 387 words, and 3304 bytes in the file. Other options provide different information.

Commands that alter the attributes of a file are:

Command	**Explanation**
`chmod  <mode> <files>`	change the file permissions
`chown <owner> <files>`	change the file ownership
`chgrp <group> <files>`	change the group ownership
`touch <files>`	update timestamps of files; create empty files (a file of size 0)

1.8.2.6 Symbolic Links

A *symbolic link*, also called a *soft link*, is a file that stores a string containing the pathname of another file. The stored string has a length of `SYMLINK_MAX` bytes or fewer. This string is not part of the content of the file, but can be used by programs to access the file whose pathname it contains. Symbolic links are like shortcuts in the Windows operating system; they have no data in themselves, but instead point to files. They are useful for overcoming the limitations of hard links that they cannot link to directories and cannot cross file systems. For example, suppose that one frequently accesses a directory whose absolute path is

```
/data/research/biochem/proteins
```

He or she could create a link in his or her home directory for easier access to this directory, as follows:

```
$ ln -s /data/research/biochem/proteins ~/proteins
```

The `ln` command, with the `-s` option, creates symbolic links. The pathname `/data/research/biochem/proteins` would be stored in the file `~/proteins` in such a way that the commands that use the filename proteins would replace it by the pathname stored there.

Symbolic links pose hazards for the operating system because of the possibility of circular references and infinite loops. More will be said about them in Chapter 3.

1.9 Under the Hood: How Logging In Works

When you log in to a UNIX system, what actually happens? What does it mean to be logged in?

There is no single answer to this question, as it depends upon (1) which version of UNIX is running and (2) whether the login is at the console, or within a terminal window, or across a network using a protocol such as SSH. Fortunately, the way that logging in works at the console or in a terminal window in the predominant UNIX systems – Linux, Mac OS, Solaris, and BSD[24] variants – is pretty much the same, with minor variations, and the way that it works over a network, while very different from how it works at a console or terminal window, is similar across the major UNIX systems.

[24]BSD stands for *Berkeley Software Distribution*. See the historical notes at the end of this chapter.

Terminal or console logins in the predominant UNIX systems are usually based upon the method used by the early BSD UNIX system. We will examine how the BSD method worked. Modern UNIX systems have the option to use a very different method known as PAM, discussed below.

When a UNIX system is started, after the kernel initializes the data structures that it needs and enables interrupts, it creates a new process with process-id 1, named `init`. `init` is the first process created at start-up, and it is also the ancestor of all user-level processes in a UNIX system, even though it runs with root's privileges. `init` monitors the activities of all processes in the outer layers of the operating system, and also manages what takes place when the computer is shutdown.

The `init` process does a number of interesting things, but of interest to us now is that `init` uses information about available terminal devices on the system (i.e., consoles, modems, etc.) to create, for each available terminal, a process to listen for activity on that terminal. These processes are the `getty` processes[25]. The name `getty` stands for "`get tty`". The `getty` process configures the terminal device, displays a prompt such as "`login:`" in the terminal, and waits for the user to enter a user-name.

The "`tty`" in "`getty`" is short for *Teletype*. For those who do not know the history, a *Teletype* is the precursor to the modern computer terminal. Teletype machines came into existence as early as 1906, but it was not until around 1930 that their design stabilized. Teletype machines were essentially typewriters that converted the typed characters into electronic codes that could be transmitted across electrical wires. Modern computer terminals inherit many of their characteristics from Teletype machines.

When the user enters a user-name on the terminal, the `getty` process runs the `login` program[26], passing it the entered user-name. `login` performs a number of tasks and prompts the user for the password and tries to validate it. If it is valid, `login` sets the current working directory (`PWD`[27]) to the user's home directory, sets the process's user-id to that of the user, initializes the user's environment, adjusts permissions and ownership of various files, and then starts up the user's login shell. If the password is invalid, the `login` program will exit, and `init` will notice this and start up a new `getty` for that terminal, which will repeat the above procedure.

Systems that use PAM do not work this way. PAM, which is short for *Pluggable Authentication Modules*, is a library of dynamically configurable authentication routines that can be selected at runtime to do various authentication tasks, not just logins. We will not cover PAM here.

Network logins, which are usually based upon the BSD network login mechanism, must work differently. For one, there are no physical terminals, and so there is no way to know in advance how many terminals must be initialized. For another, the connection between the terminal and the computer is not point-to-point, but is a network service, such as SSH or SFTP.

BSD took the approach of trying to make the login code independent of the source of the login. The result is that it uses the idea of a *pseudo-terminal*. These will be covered in depth later. With network logins, rather than creating a `getty` process for each terminal, `init` creates the process

[25]This is not quite accurate but it is good enough for now. If you want to know the actual steps, it is best to wait until you understand how processes are created and what the difference is between creating a process and running a program.

[26]Actually, the `getty` process replaces itself with the `login` program using a system call named `execve()`. This topic is covered in a later chapter.

[27]In `bash`, the environment variable `PWD` stores the absolute pathname of the current working directory. `Bash` inherits this name from the Bourne shell, in which it stood for "present working directory." In the C-shell, it is still `cwd`. Even though it is `PWD` in `bash`, no one calls it the present working directory anymore; it is the current working directory.

that will listen for the incoming network requests for logins. For example, if the system supports logging in through SSH, then `init` will create a process named `sshd`, the SSH daemon, which will in turn create a new process for each remote login. These new processes will, in turn, create what is called a *pseudo-terminal driver* (pts driver), which will then spawn the `login` program, which does everything described above. The picture is quite different and much more complicated than a simple terminal login, because the management of the pseudo-terminal is complex. This will be covered in depth later.

1.10 UNIX From The System Programmer's Perspective

The way that a systems programmer sees UNIX is very different from the way that a user sees it. Whereas the user perceives UNIX by the functionality of the shell, the systems programmer looks "beneath" the shell at the functionality of the kernel inside, as suggested by Figure 1.6.

However, not only does the systems programmer need to understand the kernel API, he or she also has to understand how programs interact with users, how they interact with the operating system, and how they interact with other programs. Interaction encompasses three distinct actions:

- Acquiring data
- Delivering data
- Coordinating execution in time

Simple programs acquire data from an external source such as a file or the keyboard, and deliver data to external sources such as files or the console. But real applications may also have to acquire data from other programs, and the operating system in particular, or acquire it from a system resource or a shared resource such as a file already opened by a different process. They may also have to write to a shared resource, such as a window into which other processes are writing, or a file that may be shared by other processes.

Figure 1.6: The system programmer's view of UNIX.

Even more complex is the possibility that data may be delivered to the process asynchronously, meaning, not when the process asked for it, but at some later, unpredictable time. And if that is not enough, consider the possibility that the process should not continue execution until some other process has performed some other task. For example, consider a program that is playing chess against another program. Neither program is allowed to make a move until the other has finished its turn. Similarly, imagine multiple processes that are cooperating to compute the structure of a complex protein. Certain parts of the structure cannot be constructed until other parts have been calculated, so the processes must coordinate their activities with respect to the work accomplished by the others. Both of these situations require the use of process synchronization. The challenge is to master these concepts.

1.11 A First System Program

We will begin by writing a stripped down version of the `more` program, which displays a file one "screen[28]" at a time. When `more` runs, it displays one screen's many lines of a file and then displays information and a prompt on the very bottom line of the screen:

–More–(0%)

It then waits for the user to press a key such as the space-bar to advance to the next screen, or the *Enter* key to advance one line, or the "`q`" key to exit. The prompt is in *reverse video*, meaning the foreground and background terminal colors are reversed. While there are other `more` advanced options, we will limit our version to these alone. Run `more` and observe that when you press the space-bar, it responds immediately; you do not have to press the *Enter* key after space-bar.

1.11.1 A First Attempt at the more Program

The `more` command can be run in several ways:

```
$ more file1 file2 ... fileN
$ ls -l | more
$ more < myfile
```

The first line causes `more` to display the files named `file1`, `file2`, and so on until `fileN`, one after the other. This proves that the `more` program has to look at the command line arguments and use them as its input if they exist. In the second example, the output of the command "`ls -l`" becomes the input of the `more` program, and the result is that `more` displays the directory listing a screen at a time. This implies that `more` also has to work when its input comes from standard input through a pipe. In the last line, the file `myfile` is used as input to the `more` command, but the `more` command is simply getting input from the standard input stream.

These three examples show that `more` has to be able to read from a file specified on the command line as well as read from standard input, and in either case it has to output a screen at a time. Suppose that we let `P` represent the number of lines in a terminal window (`P` stands for "page", which is what a screenful of data is called.) A crude outline of the `more` program, ignoring the issue of where it gets its input, is:

[28]The standard screen has 24 lines, but this is user-adjustable and also dependent on both hardware and system settings. Until we know how to find the actual number, we will assume a screen has 24 lines, and we write 23 new lines at a time.

```
Show P-1 lines from standard input (save the last line for the prompt)
Show the [more?] message after the lines.
Wait for an input of Enter, Space, or 'q'
If input is Enter, advance one line; go to 2
If input is Space, go to 1
If input is 'q', exit.
```

We have to add to this the ability to extract the filenames, if they exist, from the command line. Listing 1.4 contains a C main program for a version of `more` using the above logic, that also checks if there are one or more command line arguments, and if there are, uses the arguments as the names of files to open instead of standard input. If there are no command line arguments, it uses standard input. We will assume for now that `P`=24.

Listing 1.4: A first version of the main program for `more`.

```c
#include <stdio.h>
#define SCREEN_ROWS      23 /* assume 24 lines per screen */
#define LINELEN         512
#define SPACEBAR          1
#define RETURN            2
#define QUIT              3
#define INVALID           4

/** do_more_of()
 *  Given a FILE* argument fp, display up to a page of the
 *  file fp, and then display a prompt and wait for user input.
 *  If user inputs SPACEBAR, display next page.
 *  If user inputs RETURN, display one more line.
 *  If user inputs QUIT, terminate program.
 */
void do_more_of (FILE * filep);

/** get_user_input()
 *  Displays more's status and prompt and waits for user response,
 *  Requires that user press return key to receive input
 *  Returns one of SPACEBAR, RETURN, or QUIT on valid keypresses
 *  and INVALID for invalid keypresses.
 */
int  get_user_input();

int main( int argc , char *argv[] )
{
    FILE     *fp;
    int      i = 0;
    if ( 1 == argc )
        do_more_of( stdin );    // no args, read from standard input
    else
        while ( ++i < argc ) {
            fp = fopen( argv[i] , "r" );
            if ( NULL != fp ) {
                do_more_of( fp ) ;
                fclose( fp );
            }
            else
                printf ( "Skipping %s\n", argv[i] );
        }
    return 0;
}
```

If you are not familiar with some of the C functions (also in C++) used in this program, then read about them, either in the man pages, or in any introductory C/C++ textbook. In particular you need to know about the following functions and types, all part of ANSI standard C[29] and defined in `<stdio.h>`:

`FILE`	a file stream
`fopen`	opens a file and returns a `FILE*`
`fclose`	closes a `FILE` stream
`fgets`	reads a string from a `FILE` stream
`fputs`	writes a string to a `FILE` stream

One thing to observe in the code above is that it checks whether or not the return value of the `fopen()` function is `NULL`. Every time a program makes a call to a library function or a system function, it should check the possible error conditions. A program that fails to do this is a program that will have frustrated users.

If the argument list in the main program signature is also new to you, here is a brief explanation. In the signature

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

`argc` is an integer parameter that specifies the number of words on the command line. Since the program name itself is one word, `argc` is always at least 1. If it is exactly 1, there are no command line arguments. The second parameter, `argv`, is an array of `char` pointers. In C, a `char` pointer is a pointer to a `NULL`-terminated string, i.e., a string whose last character is `'\0'`. The array `argv` is an array whose strings are the words on the command line. `argv[0]` is the program name itself, `argv[1]`, the first command argument, and so on. The name of the parameter is arbitrary. It will work whether `argc` is named *rosebud*, *numargs*, or *ac*. The shell is responsible for putting the command arguments into the memory locations where the first and second parameters are expected (which it does by making a system call and passing these arguments to the call).

Listing 1.4 does not include the definitions of the two functions used by `main()`: `do_more_of()` and `get_user_input()`, which are contained in Listing 1.5. The first of these is easy to figure out, except that you might not have used `fgets()` and `fputs()` before. The function keeps track of how many lines can be written to the screen before the screen is full in the variable `num_of_lines`, which is initially 23. The second function is also pretty straightforward. The only part that might require explanation is the `printf()` instruction.

[29]These are in all versions of C.

1.11.2 A Bit About Terminals

Our program requires that we display the prompt in reverse video. The question is how we can display text in reverse video on a terminal. Terminals are a complex subject, about which we devote almost an entire chapter later on (Chapter 4). Now we offer just a brief introduction to them.

A terminal normally performs two roles: it is an input device and an output device. As an output device, there are special codes that can be delivered to it that it will treat, not as actual text to be displayed, but as control sequences, i.e., sequences of bytes that tell it where to position the cursor, how to display text, how to scroll, how to wrap or not, what colors to use, and so on. When terminals were first developed, there were many different types of them and many vendors. Each different type had a different set of control sequences. In 1976, the set of sequences that can be delivered to a terminal was standardized by the *European Computer Manufacturers Association (ECMA)*. The standard was updated several times and ultimately adopted by the *International Organization for Standardization (ISO)* and the *International Electrotechnical Commission (IEC)* and was named *ISO/IEC 6429*. It was also adopted by the *American National Standards Institute* (*ANSI*) and known as *ANSI X3.64*. The set of these sequences are now commonly called the *ANSI escape sequences* even though ANSI withdrew the standard in 1997.

An ANSI escape sequence is a sequence of ASCII characters, the first two of which are normally the ASCII *Escape* character, whose decimal code is 27, and the left-bracket character " [". The Escape character can be written as '`\033`' using octal notation. The string "`\033[`" is known as the *Control Sequence Introducer*, or *CSI*. The character or characters following the CSI specify an alphanumeric code that controls a keyboard or display function. For example, the sequence "`\033[7m`" is the CSI followed by the control code "`7m`". The code "`7m`" is a code that reverses the video display. The escape sequence "`\033[m`" turns off all preceding character codes.

If we want to display a portion of text, such as a prompt, in reverse video, we need to send the escape sequences and text to the terminal device. Any output function that can write to the terminal will suffice. We use the `printf()` function because it is the easiest. To write the prompt " `more?` " in reverse video, we send the first escape sequence, then the prompt, then the sequence to restore the terminal:

```
printf("\033[7m more? \033[m");
```

Although the preceding discussion centered on terminals, it is not likely that you are using an actual terminal when you are working in a shell. Most likely, you are using a *terminal emulation package*, such as *Gnome Terminal* or *Konsole* on Linux, or if connecting remotely, a package such as *PuTTY*. Almost all terminal emulators running on Unix systems interpret some subset of the ANSI escape sequences. They use software to emulate the behavior of hardware terminals. One of the first terminals to support ANSI escape sequences was the VT100. To this day, most terminal emulators support the VT100. Some also support more advanced terminals such as the VT102 or the VT220. In principle, if the terminal emulator supports the full set of ANSI escape sequences, a program that uses these sequences should work regardless of which terminal is being emulated.

This preceding code to reverse the video is just one of many escape sequences that can control the terminal. We will explore a few more of them a little later.

Listing 1.5: The supporting functions for Version 1 of more.

```
void do_more_of( FILE *fp )
```

```
{
    char    line[LINELEN];        // buffer to store line of input
    int     num_of_lines = SCREEN_ROWS;  // # of lines left on screen
    int     getmore      = 1;    // boolean to signal when to stop
    int     reply;               // input from user

    while ( getmore && fgets( line, LINELEN, fp ) ){
        // fgets() returns pointer to string read or NULL
        if ( num_of_lines == 0 ) {
            // reached screen capacity so display prompt
            reply = get_user_input();
            switch ( reply ) {
                case SPACEBAR:
                    // allow full screen
                    num_of_lines = SCREEN_ROWS;
                    break;
                case RETURN:
                    // allow one more line
                    num_of_lines++;
                    break;
                case QUIT:
                    getmore = 0;
                    break;
                default:    // in case of invalid input
                    break;
            }
        }
        if ( fputs( line, stdout )  == EOF )
            exit(1);
        num_of_lines--;
    }
}

int get_user_input()
/*
 *    display message, wait for response, return key entered as int
 *    Returns SPACEBAR, RETURN, QUIT, or INVALID
 */
{
    int     c;

    printf("\033[7m more? \033[m");    /* reverse on a VT100     */
    while( (c = getchar()) != EOF )    /* wait for response      */
        switch ( c ) {
            case 'q' :                 /* 'q' pressed */
                return QUIT;
            case ' ' :                 /* ' ' pressed */
                return SPACEBAR;
            case '\n' :                 /* Enter key pressed */
                return RETURN;
            default :                   /* invalid if anything else */
                return INVALID;
        }
}
```

Compile and run this program. To compile and run, if all of the code is in the single file named `more_v1.c`, use the commands

```
$ gcc more_v1.c -o more_v1
$ more_v1 more_v1.c
```

My code is in three files, `more_v1.c`, `more_utils_v1.c`, and `more_utils_v1.h`. I compile using

```
$ gcc -o more_v1 more_utils_v1.c more_v1.c
```

When you run it you will find that

- `more_v1` does display the first 23 lines, and
- It does produce reverse video over the `more?` prompt.
- But pressing space-bar or 'q' has no effect until you press the *Enter* key.
- If you press *Enter*, the `more?` prompt is replicated and the old one scrolls up with the displayed text. In other words, the `more?` prompt is not erased.

Next, run the command

```
$ ls /bin | more_v1
```

and observe what happens. It is not what you expected. Why not? The function `get_user_input()` calls `getchar()` to get the user's response to determine what `more_v1` should do next. If you recall the discussion in the beginning of this chapter, `getchar()` reads from the standard input stream. Since standard input is the output end of the pipeline from the `ls -l` command, `getchar()` gets characters from the `ls` listing instead of from the user's typing at the keyboard. As soon as the output of `ls -l` contains a space character or a `q` or a newline, the program will treat that as what the user typed. This first version of `more` fails to work correctly because it uses `getchar()` to get the user's input in the `get_user_input()` function. Somehow, we have to get the user's input from the keyboard regardless of the source of the standard input stream. In other words, `get_user_input()` has to read from the actual keyboard device. Well, not exactly. It has to read from the terminal that the user was given when the user logged into the UNIX system.

A process can read directly from a terminal in UNIX by reading from the file `/dev/tty`. `/dev/tty` is an example of a *device special file*.

1.11.3 Device Special Files

The UNIX system deviated from the design of all other operating systems of the time by simplifying the way in which programs handled I/O. Every I/O device (disk, printer, modem, etc.) is associated with a *device special file*, which is one kind of *special file*. Special files can be accessed using the same system calls as regular files, but the way the kernel handles the system call is different when the file argument is a device special file; the system call activates the device driver for that device rather than causing the direct transfer of data. This frees the programmer from having to write different code for different types of devices: he or she can write a program that performs output without having to know whether the output will go to a disk file, a display device, a printer, or any other device. The program just connects to a file variable, which may be associated at run time with any of these files or devices. This is the essence of *device-independent I/O*.

To illustrate, suppose a portion of C++ code writes to an output stream as follows:

```
ofstream  outfile;
cout << "Where should output be sent? ";
cin >> filename;
outfile.open(filename);
```

On a UNIX system, the user can enter the filename `"/dev/console"` at the prompt, and the output will go to the display device. If the user enters `"myfile"` the output will go to `myfile`. The kernel will take care of the details of transferring the data from the program to the device or the file.

There is another type of special file called a *named pipe*, also called a *FIFO special file*, or a *FIFO* for short. Named pipes are used for inter-process communication on a single host; they will be covered in Chapter 8.

The last type[30] of special file is a *socket*. Sockets were introduced into UNIX in 4.2BSD, and provide inter-process communication across networks. Sockets are special files in the sense that they are part of the special file system, but they are different than other special files because there are different system calls needed to manipulate them. Sockets will be covered in Chapter 9.

An entry for each device file resides in the directory `/dev`, although system administrators often create soft links to them in other parts of the file system. The advantages of device files are that

- Device I/O is treated uniformly, making it easier to write programs that are device independent;
- Changes to the hardware result only in changes to the drivers and not to the programs that access them;
- Programs can treat files and devices the same, using the same naming conventions, so that a change to a program to write to a device instead of a file is trivial;
- Devices are accorded the same protections as files.

Device files are described in greater depth in Chapter 4 of these notes.

[30] This is not completely true, since certain versions of UNIX have other special files, such as the door in Sun Solaris. The ones listed here are part of most standards.

1.11.3.1 Examples

The names of device files vary from one version of UNIX to another. The following are examples of device files that with very high certainty will be found on any system you use.

- `/dev/tty` is the name of the terminal that the process is using. Any characters sent to it are written to the screen.
- `/dev/mem` is a special file that is a character interface to memory, allowing a process to write individual characters to portions of memory to which it has access.
- `/dev/null` is a special file that acts like a black hole. All data sent to it is discarded. On Linux, `/dev/zero` is also used for this purpose, and will return null characters (`'\0'`) when read.
- The names of hard disk drive partitions vary from one system to another. Sometimes they have names like `/dev/rd0a` or `/dev/hda`. The cd-rom drive is almost always mapped to `/dev/cdrom`. You should browse the `/dev` directory on your machine to see which files exist.

Now try an experiment. Login to the UNIX system and enter the command below and observe the system response.

```
$ echo "hello" > /dev/tty
```

Now do the following. Type "`tty`" and use the output in the `echo` command as below.

```
$ tty
/dev/pts/4
$ echo "hello" > /dev/pts/4
hello
```

Now open a second window. If you are connected remotely, you can either login a second time, or use the SSH client to open a second terminal window. If you are logged in directly, just create a second terminal window. Type the `tty` command in the new window and record the name of the device file. Suppose it is `/dev/pts/2`. Now in the first window, type

```
  $ echo "hello" > /dev/pts/2
```

and observe what happens. You have just discovered that you can write to a terminal by writing to a device file. You will only be able to write to terminals that you own, not those of other people. Later we will see how you can write to other terminals, provided that the owner of that terminal has granted this type of access.

1.11.4 A Second Attempt at the more Program

We can use the device file idea to modify the more program to overcome the problem of redirected standard input. All that is necessary is to supply the `get_user_input()` function with a parameter that specifies the source of user input. This requires

1. That `do_more_of()` declare a `FILE*` variable and assign to it the result of a call to `fopen(/dev/tty)`.
2. That `do_more_of()` call `get_user_input()` with the `FILE*` variable assigned by `fopen()`.
3. That `get_user_input()` have a parameter `fp` of type `FILE*`.
4. That we replace the call in the `while` loop condition `(c = getchar())` by `(c=fgetc(fp))`.

The `getchar()` function always reads from the standard input stream. It does not take an argument. It may be, in fact, a macro. It is better to use `fgetc()`, which takes a `FILE*` argument. The modified functions are in Listing 1.6.

Listing 1.6: Corrected versions of `do_more_of()` and `get_user_input()`.

```
void do_more_of( FILE *fp )
{
    char line[LINELEN];                  // buffer to store line of input
    int  num_of_lines = SCREEN_ROWS;     // number of lines left on screen
    int  getmore      = 1;               // boolean to signal when to stop
    int  reply;                          // input from user
    FILE *fp_tty;

    fp_tty = fopen( "/dev/tty", "r" );  // NEW: FILE stream argument
    if ( fp_tty == NULL )                // if open fails
        exit(1);                         // exit

    while ( getmore && fgets( line, LINELEN, fp ) ){
        // fgets() returns pointer to string read
        if ( num_of_lines == 0 ) {       // reached screen capacity
            reply = get_user_input( fp_tty );   // NEW
            switch ( reply ) {
                case SPACEBAR:
                    // allow full screen
                    num_of_lines = SCREEN_ROWS;
                    break;
                case RETURN:
                    // allow one more line
                    num_of_lines++;
                    break;
                case QUIT:
                    getmore = 0;
                    break;
                default:    // in case of invalid input
                    break;
            }
        }
        if ( fputs( line , stdout )  == EOF )
            exit(1);
        num_of_lines--;
    }
}

int get_user_input( FILE *fp )
//  Display message, wait for response, return key entered as int
//  Read user input from stream fp
//  Returns SPACEBAR, RETURN, QUIT, or INVALID
{
    int    c;

    printf("\033[7m more? \033[m");    // reverse on a VT100
    // Now we use getc instead of getchar. It is the same except
    // that it requires a FILE* argument

    while( (c = getc( fp )) != EOF )  // wait for response
        switch ( c ) {
            case 'q' :                // 'q' pressed
                return QUIT;
            case ' ' :                // ' ' pressed
                return SPACEBAR;
            case '\n' :               // Enter key pressed
                return RETURN;
            default :                 // invalid if anything else
                return INVALID;
         }
}
```

1.11.5 What Is Still Wrong?

The second version does not display the percentage of the file displayed. It still requires the user to press the *Enter* key after the space-bar and the 'q', and the space characters and 'q' are echoed on the screen, which the real `more` prevents. It still keeps displaying the old `more?` prompt instead of erasing it. It has hard-coded the number of lines in the terminal, so it will not work if the terminal has a different number of lines. So how do we get our version of `more` to behave like the real thing? There is something we have yet to understand, which is how the kernel communicates with the terminal.

Displaying the percentage is mostly a matter of logic. The real `more` prints the percentage of bytes of the file displayed so far, not the number of lines displayed. If you know the size of the file, you can calculate the percentage of bytes printed. That problem is easy to solve once we know how to obtain the size of a file programmatically.

Remember that `/dev/tty` is not really a file; it is an interface that allows the device driver for the particular terminal to run. That device driver allows us to configure the terminal, to control how it behaves. We will see that we can use this interface to do things such as suppressing echoing of characters and transmitting characters without waiting for the *Enter* key to be pressed.

The problem of the `more?` prompt not disappearing is harder to solve, and also requires understanding how to control the terminal. One alternative is to learn more of the `VT100` escape sequences and use them to erase the `more?` prompt. Another alternative is to write copies of all lines to an off-screen buffer and clear the screen and then replace the screen contents whenever the user presses the *Enter* key. This is an easier solution than the first, but it is not how the real `more` command behaves.

1.11.6 A Third Attempt at the more Program

Our final version, not a correct version, will address some of the above problems.

The first problem is how we can determine the actual size of the terminal window. There are two solutions to this problem. If the user's environment contains the environment variables `COLUMNS` and `LINES`, then the program can use their values to determine the number of columns and lines in the terminal window, as suggested in the following code snippet that stores the `LINES` value in the variable `num_lines`.

```
int    num_lines;
char  *endptr;
char  *linestr = getenv("LINES");
if ( NULL != linestr ) {
    num_lines = strtol(linestr, &endptr, 0);
    if ( errno != 0) {
        /* handle error and exit */
    }
    if ( endptr == linestr ) {
        /* not a number so handle error and exit */
    }
}
```

The code is incomplete – one should fill in the error handling portions with the appropriate code. We will discuss error handling later. But the preceding code will not work if the `LINES` variable has not been put into the user's environment, and it will not work in the case that the user resizes the window after the program starts.

The better solution is to use a more advanced function, which will be covered in more depth in Chapter 4, named `ioctl()`. The function `ioctl()` is short for I/O Control. It is designed to do a great many things with peripheral devices, not just terminals. Its prototype is

```
#include <sys/ioctl.h>
int ioctl(int d, int request, ...);
```

Notice the "`...`" in the parameter list. This is C's notation for a variable number of parameters. The first parameter is a file descriptor. File descriptors are covered in Chapter 2. The second parameter, of integer type, is a command. The following parameters are arguments to the command, and they depend on the given command. The `ioctl()` call to get the number of rows and columns of the current terminal window has three arguments:

```
ioctl(tty_file_descriptor, TIOCGWINSZ, &window_structure))
```

where `tty_file_descriptor` is a file descriptor for our terminal, `TIOCGWINSZ` is a command that means "get the terminal window size structure" and `window_size_structure` is a variable of type `struct winsize`. The structure winsize is defined in a header file that is automatically included when you include the `<sys/ioctl.h>` header file in the code. The code that would use this `ioctl` is

```
struct winsize window_arg;
int    num_rows, num_cols;

fp_tty = fopen( "/dev/tty", "r" );
if ( fp_tty == NULL )
    exit(1);
if (-1 == ioctl(fileno(fp_tty), TIOCGWINSZ, &window_arg))
    exit(1);
num_rows =  window_arg.ws_row;
num_cols =  window_arg.ws_col;
```

The function `fileno()` used inside the `ioctl` is covered in Chapter 2; it converts a `FILE*` to a *file descriptor*. File descriptors are also explained in Chapter 2. We will use the following `get_tty_size()` function in our last version of the `do_more_of()` and `get_user_input()` functions:

```
void get_tty_size(FILE *tty_fp, int* numrows, int* numcols)
{
    struct winsize window_arg;
    if (-1 == ioctl(fileno(tty_fp), TIOCGWINSZ, &window_arg))
exit(1);
    *numrows =  window_arg.ws_row;
    *numcols =  window_arg.ws_col;
}
```

Although the `ioctl()` function is standard and should be available on any UNIX system, the particular commands that it implements may not be. In particular, it is possible that the `TIOCGWINSZ` command is not available. When you write code that depends on features that are not guaranteed to be present, it is best to conditionally compile it by enclosing it within preprocessor directives to conditionally compile. The preceding function is better written using

```
void get_tty_size(FILE *tty_fp, int* numrows, int* numcols)
{
#ifdef TIOCGWINSZ
    struct winsize window_arg;
    if (-1 == ioctl(fileno(tty_fp), TIOCGWINSZ, &window_arg))
exit(1);
    *numrows =  window_arg.ws_row;
    *numcols =  window_arg.ws_col;
#else
    /* some fallback code here */
#endif
}
```

We will omit the fallback code for now, since it requires knowing more about terminals.

The second problem is how to remove the old `more?` prompt so that it does not scroll up the screen. In short we need a way to clear portions of the screen. We also need a way to move the cursor to various places on the screen. There are various ANSI escape sequences that do these things. The table below lists a small fraction of the set of ANSI escape sequences. The ones in the table are the easiest to learn and use. In the table, the symbol "`ESC`" is short for '`\033`'. In almost all cases, the default value of *n* in the table is 1.

Key Sequence	**Meaning**
`ESC[M`	Move the cursor up in scrolling region.
`ESC[`*n*`A`	Move the cursor up *n* lines.
`ESC[`*n*`B`	Move the cursor down *n* lines
`ESC[`*n*`C`	Move the cursor right *n* columns
`ESC[`*n*`D`	Move the cursor left *n* columns
`ESC[`*n*`G`	Move the cursor to column *n*
`ESCE`	Move cursor to start of next line
`ESC[`*r*`;`*c*`H`	Move the cursor to row *r*, column *c*.
`ESC[0K`	Erase from the cursor to the end of the line
`ESC[1K`	Erase from the beginning of the line to the cursor.
`ESC[2K`	Erase the line
`ESC[0J`	Erase from the cursor to the end of the screen.
`ESC[1J`	Erase from the bottom of the screen to the cursor
`ESC[2J`	Erase the screen
`ESC[0m`	Normal characters
`ESC[1m`	Bold characters
`ESC[4m`	Underline characters.
`ESC[5m`	Blinking characters
`ESC[7m`	Reverse video characters
`ESC[g`	Clear tab stop at current column.
`ESC[3g`	Clear all tab stops.

Examples

Following are a few examples of sequences of commands. The last two are compound control sequences, and they are of signficance because we can use them in our program.

Sequence	**Meaning**
`\033[2A`	Move the cursor up two lines.
`\033[2J`	Clear the entire screen.
`\033[24;80H`	Move the cursor to row 24, column 80.
`\033[1A\033[2K\033[1G`	Move up one line, erase that line, and move the cursor to the leftmost column.
`\033[1A\033[2K\033[1B\033[7D`	Move the cursor up one line; erase that line; move the cursor back down one line, and move it to the left 7 positions.

We can combine the formatting features of the `printf()` function with our `get_tty_size()` function and these escape sequences to park the cursor in the lower left hand corner of the screen, regardless of how large the screen is, and display the prompt in reverse video:

```
    int   tty_rows;
    int   tty_cols;
    get_tty_size(fp, &tty_rows, &tty_cols);
    printf("\033[%d;1H", tty_rows);
    printf("\033[7m more? \033[m");
```

The first `printf()` moves the cursor to the last row of the screen in the first column. The second displays the prompt in reverse video and then reverts the terminal back to normal character display. The third version of `more`, in Listing 1.7 below, includes these two improvements. The main program does not change, so it is not listed.

Listing 1.7: A third version of the more program.

```
#include  "more_utils.h"
#include  <sys/ioctl.h>

#define   LINELEN 512

void get_tty_size(FILE *tty_fp, int* numrows, int* numcols)
{
    struct winsize window_arg;

    if (-1 == ioctl(fileno(tty_fp), TIOCGWINSZ, &window_arg))
        exit(1);
    *numrows =  window_arg.ws_row;
    *numcols =  window_arg.ws_col;
}

/** get_user_input(FILE *fp )
 *  Displays more's status and prompt and waits for user response,
 *  Requires that user press return key to receive input
 *  Returns one of SPACEBAR, RETURN, or QUIT on valid keypresses
 *  and INVALID for invalid keypresses.
 *  Reads from fp instead of from standard input.
 */
int get_user_input( FILE *fp )
{
    int   c;
    int    tty_rows;
    int    tty_cols;

    /*
     * Get the size of the terminal window dynamically, in case it changed.
     * Then use it to put the cursor in the bottom row, leftmost column
     * and print the prompt in "standout mode" i.e. reverse video.
     */
    get_tty_size(fp, &tty_rows, &tty_cols);
    printf("\033[%d;1H", tty_rows);
    printf("\033[7m more? \033[m");


    /* Use fgetc() instead of getc().  It is the same except
     * that it is always a function call, not a macro, and it is in general
     * safer to use.
     */
    while ( (c = fgetc( fp )) != EOF )  {
        /* There is no need to use a loop here, since all possible paths
         * lead to a return statement. It remains since there is no downside
         * to using it.
         */
        switch ( c ) {
            case 'q' :                  /* 'q' pressed */
                return QUIT;
            case ' ' :                  /* ' ' pressed */
                return SPACEBAR;
            case '\n' :                 /* Enter key pressed */
                return RETURN;
            default :                   /* invalid if anything else */
                return INVALID;
        }
    }
    return INVALID;
}

/** do_more_of ( FILE * fp )
 *  Given a FILE* argument fp, display up to a page of the
 *  file fp, and then display a prompt and wait for user input.
 *  If user inputs SPACEBAR, display next page.
 *  If user inputs RETURN, display one more line.
 *  If user inputs QUIT, terminate program.
 */
void do_more_of( FILE *fp )
{
    char   line[LINELEN];              // buffer to store line of input
    int    num_of_lines;               // number of lines left on screen
    int    reply;                      // input from user
    int    tty_rows;                   // number of rows in terminal
    int    tty_cols;                   // number of columns in terminal
    FILE   *fp_tty;                    // device file pointer

    fp_tty = fopen( "/dev/tty", "r" );  // NEW: FILE stream argument
    if ( fp_tty == NULL )               // if open fails
```

```
        exit(1);                        // exit

    /* Get the size of the terminal window */
    get_tty_size(fp_tty, &tty_rows, &tty_cols);
    num_of_lines = tty_rows;

    while ( fgets( line, LINELEN, fp ) ){
        if ( num_of_lines == 0 ) {
            // reached screen capacity so display prompt
            reply = get_user_input( fp_tty );  // note call here
            switch ( reply ) {
                case SPACEBAR:
                    // allow full screen
                    num_of_lines = tty_rows;
                    printf("\033[1A\033[2K\033[1G");
                    break;
                case RETURN:
                    // allow one more line
                    printf("\033[1A\033[2K\033[1G");
                    num_of_lines++;
                    break;
                case QUIT:
                    printf("\033[1A\033[2K\033[1B\033[7D");
                    return;
                default:    // in case of invalid input
                    break;
            }
        }
        if ( fputs( line, stdout )  == EOF )
            exit(1);
        num_of_lines--;
    }
}
```

This version still has a few deficiencies. One is that it checks whether the terminal has been resized in each iteration of the loop, whenever it retrieves the user input. Although this works, it wastes cycles. Later we will learn how to do this asynchronously, finding the size of the terminal only when it has actually been resized.

Another deficiency is that it still requires the user to press the Enter key. Remedying this requires further understanding of terminals, again in Chapter 4. One feature we can easily integrate into the program is calculating the percentage of the file displayed so far. This is left as an exercise.

1.12 Where We Go from Here

The real purpose of trying to write the `more` program is to show that, using only the usual high-level I/O libraries, we cannot write a program that does the kind of things that `more` does, such as ignoring keyboard input and suppressing display of typed characters. The objective of these notes is to give you the tools for solving this kind of problem, and to expose you to the major components of the kernel's API, while also explaining how the kernel looks "under the hood." We are going to look at each important component of the kernel. You will learn how to rummage around the file system and man pages for the resources that you need.

Appendix A Brief History of UNIX

In 1957, Bill Norris started Control Data Corporation (CDC). In 1959 Ken Olsen started DEC with $70,000 in venture capital money. The first PDP-1 (manufactured by DEC) was shipped in 1960. In 1963, Project MAC (Multiple Access Computers) was organized at MIT to do research on interactive computing and time-sharing systems. In 1965, AT&T, GE, and Project MAC at IBM joined together to develop the time-sharing system MULTICS (Multiplexed Information and Computing Service).

In 1966, Ken Thompson finished studies at University of California at Berkeley (UCB) and joined the technical staff at AT&T Bell Telephone Laboratories to work on MULTICS. Two years later, Dennis Ritchie completed work on his doctorate at Harvard and joined Bell Labs to work on MULTICS project.

Thompson developed the interpretive language B based upon BCPL. Ritchie improved on "B" and called it "C".

In 1969 while working for AT&T Bell Labs, Ken Thompson and Dennis Ritchie wrote the first version of UNICS for a PDP-7, manufactured by DEC. It ran on a machine with 4K of 18-bit words. UNICS is a pun on MULTICS and stood for Uniplexed Information and Computing Services. Three years later, it was rewritten in C so that they could port it to other architectures. By 1973, the first UNIX support group was formed within Bell Labs, and some fundamental UNIX design philosophies emerged. Ritchie and Thompson gave a presentation about the new operating system at the *ACM Symposium on Operating Systems* at IBM that year. This was the catalyst that sparked the spread of the UNIX system. The University of California at Berkeley (UC Berkeley) soon acquired Version 4 from Bell Labs and embarked on a mission to add modern features to UNIX.

Over the next four years, a number of events shaped the future of UNIX. AT&T started licensing it to universities. Boggs and Metcalfe invented the Ethernet at Xerox in Palo Alto. Bill Joy joined UC Berkeley and developed BSD Version 1. UNIX was ported to a non-DEC machine. (DEC had been unwilling to support UNIX.) P.J. Plauger wrote the first commercial C compiler, and Doug and Larry Michel formed a company called Santa Cruz Operations (SCO) to start selling a commercial version of UNIX for the PC.

In 1974, "The UNIX Time-Sharing System" was published in CACM by Ken Thompson and Dennis Ritchie. That same year, the University of California at Berkeley (UCB) got Version 4 of UNIX, and Keith Standiford converted UNIX to a PDP 11/45. *The Elements of Programming Style* by Kernighan and Plauger was published, and AT&T officially began licensing UNIX to universities.

In 1976, *Software Tools* by Kernighan and Plauger was published, and Boggs and Metcalfe invented the Ethernet at Xerox in Palo Alto.

By 1978, UNIX had essentially all of the major features that are found in it today. One year later, Microsoft licensed UNIX from AT&T and announced XENIX (even before producing MS-DOS).

UNIX spread rapidly because:

- It was easily ported from one architecture to another because it was written in a high level language (C), unlike other operating systems.

- It was distributed to universities and research laboratories for free by AT&T, and early commercial versions were sold at very low cost to academic institutions.
- The source code was freely available, making it easy for people to add new features and programs to their systems.
- It had several features (e.g., pipes) that had not been part of other operating systems.

In 1979, Bell Labs released UNIX™ Version 7, attempting to stake proprietary rights to UNIX, and Microsoft licensed UNIX from AT&T and announced XENIX (even before producing MS-DOS). In the early 1980s, a number of developments occurred that shaped the future of the UNIX evolutionary tree:

- AT&T and Berkeley (the Computer Systems Research Group, or CSRG) each started to develop their own independent versions of UNIX.
- Bill Joy left Berkeley to co-found a new company, SUN Microsystems, to produce UNIX workstations. Sun got its name from the Stanford University Network (SUN) board. The workstation was based on the Motorola 68000 chip running SunOS based on 4.2BSD. It included an optional local area network based on Ethernet. The commercial UNIX industry was in full gear.
- Microsoft shifted its focus on the development of MS-DOS, shelving marketing of XENIX.
- The X/Open standards group was formed.

Over the next fifteen years, many different versions of UNIX evolved. Although the differences between these versions are not major, care must be taken when trying to treat "UNIX" as a single operating system. Through the 1980s and early 1990s, the two major UNIX distributions were the Berkeley series known by names such as 4.2BSD and 4.3BSD and the AT&T series known by names such as System III and System V. SUN's UNIX was called SunOS, and later Solaris, and was a hybrid of the two strains.

Since its creation in 1969, UNIX has grown from a single, small operating system to a collection of distinct operating systems of varying complexity, produced by vendors all around the world. The essential features of most UNIX systems have remained the same, partly because of standardization efforts that began in the late 1980's and continued through the 1990's. Modern UNIX is a slippery concept. Several standards have been written, and they have a large common core. What is generally called UNIX is the common core of these standards, which include POSIX, and UNIX 98 from the Open Group. UNIX 98 is also called the Single UNIX Specification and it alone is UNIX as far as property rights are concerned, since the Open Group owns the UNIX trademark now. The Open Group is an international consortium of more than 200 members from government, academia, worldwide finance, health care, commerce and telecommunications. Of course, there is also Linux, which has overtaken BSD and System V on the desktop.

Appendix B UNIX at the User Level

This section is designed for people with little, if any, experience with UNIX. It covers:

- Logging in and out
- Shells
- The environment

B.1 Notation

In the description of a command, square brackets `[ ]` enclose optional arguments to the command, and angle brackets `< >` enclose placeholders. The brackets are not part of the command. A vertical bar `"|"` is a logical-or. An ellipsis `...` means more than one copy of the preceding token. For example

```
ls [<option>] ... [<directory_name>] ...
```

indicates that both the option specifiers and the argument to the `ls` command are optional, but that any options should precede any directory names, and that both option specifiers and directory names can occur multiple times, as in

```
ls -l -t mydir yourdir
```

Commands will be preceded by a dollar sign to indicate that they are typed after the shell prompt. When the input and output of a command are shown, the user's typing will be in **bold**, and output will be in regular text.

```
$ ls -l -t mydir yourdir
```

B.2 Logging In

A user uses a UNIX system by logging in, running programs, and logging out. There are two different ways to login to a UNIX system:

- When sitting at the console of the computer, i.e., the computer keyboard and monitor are physically in front of you, and no one is presently logged in to the machine, or
- Remotely, by using a remote login utility like SSH.

At the Console

If you login to the computer at its own console, the appearance of the screen will depend upon which version of UNIX you use. In all cases, there will be an option to enter a user or login name followed by a password. For RedHat 9 with the Gnome 2 desktop environment, the screen will appear as in Figure B.1. After logging in, you will see a desktop with a graphical user interface, and a menu somewhere on the screen with an option somewhere to open a terminal window. It is within that terminal window that the rest of the notes pertain.

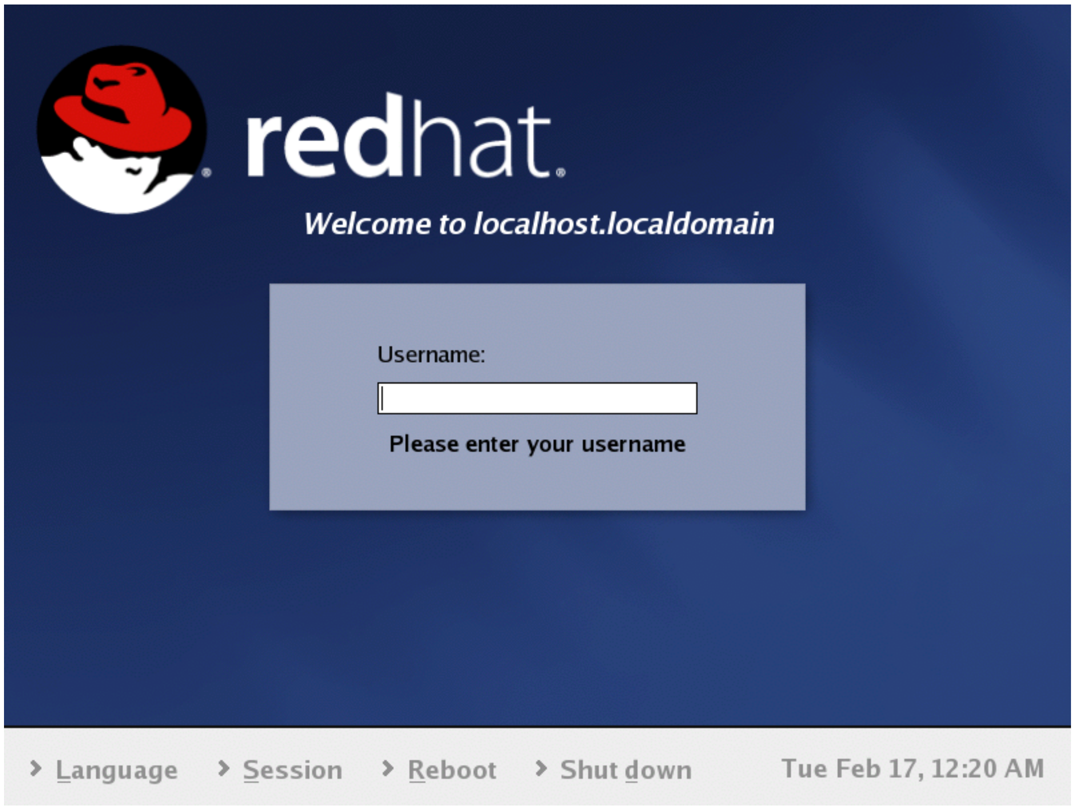


Figure B.1: RedHat 9 login screen.

Remotely via SSH

SSH is a protocol that provides encrypted communication to prevent passwords and other secure information from being captured in transit over insecure networks. SSH stands for *Secure SHell*. SSH is a client/server application. The server runs on the remote host, the one into which you want to login, and the client is on your local host. Once you have established a connection to the UNIX system from your local machine using SSH, everything that you type is encrypted by the SSH client on your local machine and decrypted by the remote host. The encryption algorithms are very strong and the keys are very secure.

These notes do not describe how to use any particular SSH client. All such clients will need from you the IP address of the remote host, the login or user name, and your password. Assuming that you have provided these, the SSH client will log you in and display a terminal window on your local machine.

You will see some messages, followed by the prompt displayed by your *login shell*. A login shell is the shell that runs when you log-in. It depends on the particular version of UNIX you are using and how the initial messages were configured by the system administrator. On the Hunter College Computer Science Department's network gateway machine, named `eniac`, which was running Solaris 9, a version of UNIX from SunSoft, it looked like:

```
Last login: Mon Aug 30 2004 19:51:19 -0500 from pool-68-161-100-
Sun Microsystems Inc.   SunOS 5.9       Generic October 1998
This workstation is running SunOS 5.x
eniac{sweiss} [42] $
```

When eniac was later converted to a Linux host, the login messages became:

```
Last login: Fri Jan 20 17:21:53 2006
from pool-68-161-21-160.ny325.east.verizon.net
eniac{sweiss} [339] $
```

In both cases, the last line begins with a *prompt string*, which is a sequence of characters followed by a waiting cursor. In each of these the prompt string looks like

```
eniac{sweiss} [n] $
```

where `n` is some number. The prompt is sometimes "`$`" alone. It depends on how it has been configured, which partly depends upon the shell. Some shells let you do more configuring than others. This prompt displays the name of the host computer, the user-name on that host, and the *history number* (42 or 339) of the current command. The history number is the ordinal number that indicates how many commands you have entered so far. It gets reset to 1 when it reaches the maximum number of saved commands. One can retrieve old commands using their numbers.

Your *login shell* is the special shell that is started up each time you login. On UNIX systems with a windowing environment, each window will have a separate instance of a running shell. Even UNIX systems with a GUI follow the shell paradigm – instead of typing commands, you use a pointing device and click on icons and controls, but the effect is still to construct calls to system programs, to a shell, or to shell scripts.

When you have logged-in, you will be "in" your *home directory*. You might feel comfortable with the idea of "being in a directory" but it makes no sense to say this. You are in your seat. You are not in the computer. Nonetheless, this language has taken hold. You probably know that "being in the home directory" means that when you view the contents of the current directory, you will see the files in your home directory. But this is an inadequate explanation. I will have more to say about the concept of "being in a directory" shortly. What you do need to understand now is that the home directory is the directory that you "are in" each time you log-in. It is the starting place for your current session. The `HOME` environment variable stores the "name" of your home directory. This will also be clarified shortly.

B.3 Logging Out

When you are finished using the system, you log out. There are usually a few ways to log out. POSIX specifies that a shell provide a built-in `exit` command, which can be used to exit any shell. There is also the `logout` command, which may or may not be built-into the shell:

```
eniac{sweiss} [342]$ logout
```

The difference is that logout can only be used to log out, not to terminate a shell that is not a login shell, such as a sub-shell of some other shell, whereas exit terminates all shells and logs you out if the shell is a login shell.

B.4 Online Help: The man Pages

Before going any further, you should understand how to get help online. Traditional UNIX provides only one source of online help – the manual pages. The single most important command to remember is the man command. The word "man" is short for "manual"; if you type man following by the name of a command, UNIX will display the manual page, called the man page for short, for that command. For example, to learn more about a command named echo, type

```
$ man echo
```

and you will see several screens of output, beginning with:

```
echo(1)                    User Commands                         echo(1)
NAME
       echo - display a line of text

SYNOPSIS
       echo [SHORT-OPTION]... [STRING]...
       echo LONG-OPTION

DESCRIPTION
       Echo the STRING(s) to standard output.
       -n     do not output the trailing newline
       -e     enable interpretation of backslash escapes
       -E     disable interpretation of backslash escapes (default)
       (remaining lines omitted )
: █
```

Every man page is in the same format, which may vary from one UNIX system to another. The first line shows the name of the section of the manual in which the page was found (*User Commands*) and the name of the man page followed by the section number (*echo, section 1*). Sometimes the name of the man page will be different from the name of the command. The sections of the man page, which may not all be present, are:

NAME	name of the command
SYNOPSIS	syntax for using the command
DESCRIPTION	brief textual summary of what the command does
OPTIONS	precise descriptions of command-line options
OPERANDS	precise descriptions of command-line arguments
USAGE	a more thorough description of the use of the command
ENVIRONMENT VARIABLES	list of environment variables that affect the command execution
EXIT STATUS	list of exit values returned by the command
FILES	list of files that affect the command execution
ATTRIBUTES	architectures on which it runs, availability, code independence, etc.
SEE ALSO	list of commands related to this command
NOTES	general comments that do not fit elsewhere
BUGS	known bugs

In man pages, the square bracket notation `[ ]` as in `[SHORT-OPTION]` above defines an optional command-line option or argument. The "`:`" is followed by your cursor because many UNIX systems now pipe the output of the man command through the `less` command for viewing, and the "`:`" is the `less` prompt to the user to type something on the keyboard. More accurately, the `less` command is the viewer for the man command itself[1].

For learning how to use a command, the man page by itself is usually sufficient. For learning how a command interacts with the operating system or how it might be implemented, one must do more research. In the next chapter, I will explain how to use the man pages in more detail for the latter purpose. In the meanwhile, you should look at the man pages of the various commands mentioned in this chapter as you read on.

It should also be mentioned that modern UNIX systems also provide other sources of help, such as the help and info commands. If the system you are using has these commands, typing help or info followed by a command or a topic will display information about that topic.

B.5 Shells

Your view and appreciation of UNIX is pretty much determined by the interface that the shell creates for you, as suggested in Figure B.2. The shell hides the inner workings of the kernel, presenting a set of high level functions that can make the system easy to use. Although you may have used a UNIX system with a graphical user interface, you must be aware that this GUI is an application separate and distinct from UNIX. The GUI provides an alternative to a shell for interacting with UNIX, but experienced users usually rely on using a shell for many tasks because it is much faster to type than it is to move a mouse around pointing and clicking.

There are many different shells, including the *Bourne* shell (`sh`), the *C* shell (`csh`), the *Korn* shell (`ksh`), the *Bourne-again* shell (`bash`), the *Z* shell (`zsh`), and the *TC* shell (`tcsh`). Although a few shells may be bundled with the operating system, they are not actually a part of it.

Unlike most operating systems, UNIX allows the user to replace the original login shell with one of his or her own choosing. The particular shell that will be invoked for a given user upon login is

[1]You can change which viewer the man command uses by changing the value of the PAGER environment variable.

Figure B.2: The shell: A user's view of UNIX.

specified in the user's entry in a file called the *password file*. The system administrator can change this entry, but very often the `chsh` command available on some systems can also be used by the user to change shells. In some versions of UNIX, the user can also use the command `passwd -s` to change the login shell.

B.5.0.1 Shell Features

In all shells, a simple command is of the form

```
$ commandname command-options arg1 arg2 ... argn
```

The shell waits for a newline character to be typed before it attempts to interpret (parse) a command. A newline signals the end of the command. Once it receives the entered line, it checks to see if `commandname` is a *built-in shell command* (A built-in command is one that is hard-coded into the shell itself.) If it is, it executes the command. If not, it searches for a file whose name, either relative or absolute, is `commandname`. (How that search takes place is explained in a later chapter.) If it finds one, this file is loaded into memory and the shell creates a child process to execute the command, using the arguments from the command line. When the command is finished, the child process terminates and the shell resumes its execution.

In addition to command interpretation, all shells provide

- redirection of the input and output of commands
- pipes – a method of channeling the output of one command to the input of another
- scripting – a method of writing shell programs that can be executed as files
- file name substitution using metacharacters
- control flow constructs such as loops and conditional execution

Shells written after the Bourne shell also provide

- history mechanism – a method of saving and reissuing commands in whole or in part

- backgrounding and job control – a method of controlling the sequencing and timing of commands
- aliases for frequently used commands

The `tcsh` shell added a number of features to the C shell, such as interactive editing of the command line and interactive file and command name completion, but the Korn shell introduced many significant additions, including:

- general interactive command-line editing
- coprocesses
- more general redirection of input and output of commands
- array variables within the shell
- autoloading of function definitions from files
- restricted shells
- menu selection

`bash` borrowed many of the features of its predecessors and has almost all of the above capabilities.

B.5.0.2 Standard I/O and Redirection

UNIX uses a clever method of handling I/O. Every program is automatically given three open files when it begins execution, called *standard input*, *standard output*, and *standard error*. (POSIX does not require standard error, but it is present in all systems that I know.) Standard input is by default the keyboard and standard output and error are to the terminal window.

Commands usually read from standard input and write to standard output. The shell, however, can "trick" a command into reading from a different source or writing to a different source. This is called *I/O redirection*. For example, the command

```
$ ls mydir
```

ordinarily will list the files in the given directory on the terminal. The command

```
$ ls mydir > myfiles
```

creates a file called `myfiles` and redirects the output of the `ls` command to `myfiles` provided `myfiles` did not already exist, in which case it will display a message such as

```
bash: myfiles: cannot overwrite existing file
```

The notation "`> outfile`" means "put the output of the command in a file named `outfile` instead of on the terminal." The *output redirection operator*, "`>`", replaces the standard output of a program by the the file whose name follows the operator.

Analogously, the notation "`< infile`" means read the input from the file `infile` instead of from the keyboard. Technically, the "`<`", known as the *input redirection operator*, replaces the standard input of a program by the file whose name follows the operator.

A command can have both input and output redirected:

```
$ command < infile > outfile
```

which causes the command to read its input from `infile` and send its output to `outfile`. Th order of the command, and the input and output redirection does not matter. One can also write any of the following semantically equivalent lines:

```
$ command > outfile < infile
$ > outfile command < infile
$ < infile > outfile command
```

The concept of redirection is carried one step further to allow the output of one command to be the input of another. This is known as a *pipe* or *pipeline*, and the operator is a vertical bar "`|`" :

```
$ ls mydir | sort | lpr
```

means list the contents of the given directory, and instead of writing them on the terminal or in a file, pass them as input to the `sort` command, which then sorts them, and sends the sorted list to the `lpr` command, which is a command to send files to the default printer attached to the system. It could have been done using temporary files as follows:

```
$ ls mydir > temp1
$ sort < temp1 > temp2
$ lpr < temp2
$ rm temp1 temp2
```

but in fact this is not semantically equivalent because when a pipeline is established, the commands run simultaneously. In the above example, the `ls`, `sort`, and `lpr` commands will be started up together. The shell uses a kernel mechanism called *pipes* to implement this communication. There are other kinds of I/O redirection, including appending to a file and redirecting the system error messages; for details consult the shell's man page.

There are two other I/O redirection operators: `<<` and `>>`. If you are curious and unfamiliar with these, read about them in the `bash` man page.

B.5.0.3 Command Separators and Multitasking

Commands can be combined on single lines by using command separators. The semicolon ";" acts like a newline character to the shell – it terminates the preceding command, as in:

```
$ ls mydir ; date
```

which lists the contents of the given directory and then displays the current time and date. The `date` command displays the date and time in various formats, but by default its output is in the form

```
Mon Aug 15 22:53:54 EDT 2011
```

The semicolon ";" is used to sequentially execute the commands. In contrast,

```
$ ls mydir > outfile &
```

causes the `ls` command to work "in the background." The ampersand "&" at the end of the line tells the shell not to wait for the command to terminate but to instead resume immediately; it is a *backgrounding* operator. The `ls` command runs in the background, sending its output to the file `outfile` while the shell does other things. This is a form of *multitasking*. The "&" can be used to run separate commands as well:

```
$ ls mydir > outfile1 & date > outfile2 &
```

tells the shell to run the `ls` and `date` commands *concurrently* and in the background, putting their respective outputs in files `outfile1` and `outfile2` respectively. Note that the ">" binds more closely, i.e., has higher priority, than "&". Lastly, parentheses can be used for grouping:

```
$ (ls mydir; date) > outfile &
```

means "execute `ls mydir`, then `date`, and direct their combined output to `outfile`," all in the background.

B.5.0.4 The Shell as a Command

Shells can be run as commands. You can type the name of a shell, e.g., `bash`, `sh`, `csh`, etc., at the command prompt within any other shell, to start another shell:

```
$ sh
```

in this case the Bourne shell. If you do this, you will have two instances of the `bash` shell running, but the first will be dormant, waiting for the second to exit. When one shell is created as a result of a command given in another shell, the created shell is called the sub-shell of the original shell, which is called the parent shell.

If you need to know what shell you are currently running (because you forgot, for example), there are two simple commands that can tell you:

```
$ echo $0
```

and

```
$ ps -p $$
```

The first uses the `echo` command. The `echo` command may sound at first rather useless, but it is very convenient. All it does is evaluate its arguments and display them. So writing

```
$ echo hello out there
hello out there
```

But the key is that `echo` evaluates its arguments. If you give it the name of a variable preceded by a dollar sign "$", it displays the value of the variable, not the name. As an example, you can use it to see the value of a particular environment variable, such as

```
$ echo $SHELL
/bin/bash
```

This does not print your current shell however. It prints the name of your login shell, which may not be the current shell. The value of `$0` is the name of the currently running program, which is the shell you are using, so `echo $0` is one way to see its name. The value of `$$` is the process-id of the currently running process, and `ps -p` is a command that will display information about the process whose process-id is given to it.

B.5.0.5 Scripts

Suppose the file named `myscript` contains the following lines:

```
ls mydir
date | lpr
```

Then the command

```
$ bash < myscript
```

causes the commands in the file to be executed as if they were typed directly to the `bash` shell, and when they have terminated, the `bash` shell exits. The file `myscript` is an example of a *shell script*. A script is nothing more than a program written in an interpreted language. It is not compiled, but executed line by line. Shell scripts can have command line arguments just like ordinary shell commands, making them very general. For example, the above script can be rewritten as

```
#!/bin/bash
ls $1
date | lpr
```

and executed by typing a command such as

```
$ myscript ~/bin
```

whereupon the "$1" will be replaced by ~/bin before the ls command is executed. The very first line indicates that the bash shell must be run to execute the remaining lines of the file. For more details of specific shell languages, consult the shell's man page (or read any of a multitude of on-line tutorials.)

Each shell has properties that determine how it behaves, which are stored in the shell's environment. As noted in Section 1.3.5, the operating system passes a pointer to a copy of the user's environment when it starts up a new process. Since a running shell is a process, each time a new shell is started from within another, it is given a copy of the previous shell's environment.

More generally, when any process is created from within a shell, that process inherits a copy of the values of the environment variables of the shell. When you run a program from the command line, the program inherits the environment of the shell. This is how programs "know" what their current working directories are, for example, or which users are their owners. The environment is a key element in making things work.

You are free to customize the environment of a shell by defining new variables, or redefining the values of existing variables. The syntax is shell-dependent. In the Bourne shell and bash, variables that you define, called *locals*, are not automatically placed into the environment; to place then in the environment, you have to export them. For example,

```
$ export MYVAR=12
```

will put MYVAR into the environment with the value 12, so that it can be inherited by future commands and processes, or equivalently:

```
$ MYVAR=12; export MYVAR
```

The convention is to use uppercase names for environment variables and lowercase names for locals.

B.5.0.6 Some Historical Remarks About Shells

The C shell was written by Bill Joy at UC Berkeley and made its appearance in Sixth Edition UNIX (1975), the first widely distributed release from UC Berkeley. It was an enhancement of the original Thompson shell with C-like syntax. It is part of all BSD distributions.

The Bourne shell, written by Stephen Bourne, was introduced into UNIX in System 7 (1979), which was the last release of UNIX by AT&T Bell Labs prior to the commercialization of UNIX by AT&T. Its syntax derives in part from the programming language Algol 68 and is a part of all UNIX releases.

The Korn shell was developed by David Korn at Bell Labs and introduced into the SVR4 commercial release of UNIX by AT&T. Its syntax is based on the Bourne shell but it had many more features.

The TENEX C shell, or TC shell, extended the C shell with command line editing and completion, as well as other features which were found in the TENEX operating system. It was written by Ken Greer and others in 1983.

The Bourne-again shell (Bash) is an extension of the Bourne shell with many of the sophisticated features of the Korn shell and the TC shell. It is the default user shell in Linux and has become popular because of this.

Bibliography

[1] Bruce Molay. *Understanding UNIX/LINUX Programming: A Guide to Theory and Practice.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2002.

[2] Dennis M. Ritchie and Ken Thompson. The unix time-sharing system. *Commun. ACM*, 17(7):365–375, July 1974.