There is no conceptual difference between the original bc/dc model and the World Wide Web. It is probably no coincidence that the Web first grew up on Unix systems.[2]

1.6 CAN I TRY TO DO ONE?

We have now looked at the first two questions. We looked at several aspects of Unix and asked, "What does that do?" We have asked, "How does that work?" In a few cases, such as bc/dc, we saw at least a partial answer.

The third question in our approach is "Can I try to do one?" In this section, we write a version of the Unix more program.

First, "What does more do?"

more displays a file one screenful at a time. Most Unix systems come with a large text file called /etc/termcap used by some editors and video games. If you want to page through this file, you can type

```
$ more /etc/termcap
```

You will see the first screenful of the file. At the bottom of the screen, the more program will print, in reverse video, the percentage of the file you have seen. You may press the space bar to see the next page, you may press the Enter key to see the next line, you may press the letter "q" to quit, and you may press "h" to see the help screen.

Notice that you do not have to press the Enter key after pressing the space bar or "q" or "h." The program responds at once.

The three ways to invoke more from the command line are as follows:

```
$ more filename
$ command | more
$ more < filename
```

In the first case, more displays the contents of the named file. In the second case, the program specified by *command* runs, and its output is displayed page by page. In the third case, more displays the contents of what it reads from standard input. Standard input happens to be attached to the specified file.

Second, "How does more work?"

After running more a few times, we can guess that the logic probably goes something like this:

```
+----> show 24 lines from input
| +--> print [more?] message
| |    Input Enter, SPACE, or q
| +--  if Enter, advance one line
+----  if SPACE
       if q --> exit
```

[2]By the way, it was Amy Chused, a student in my course, who pointed out the connection between bc/dc and all TCP/IP client-server programming.

Our program should be flexible about input, just like the real more. That is, if the user specifies a filename on the command line, we read from that file. If no filename is specified on the command line, the program reads from standard input.

Here is a first draft of our own version of more:

```
/* more01.c - version 0.1 of more
 *      read and print 24 lines then pause for a few special commands
 */
#include         <stdio.h>

#define PAGELEN 24
#define LINELEN 512

void do_more(FILE *);
int see_more();

int main( int ac , char *av[] )
{
        FILE *fp;
        if ( ac == 1 )
                do_more( stdin );
        else
                while ( --ac )
                        if ( (fp = fopen( *++av , "r" )) != NULL )
                        {
                                do_more( fp ) ;
                                fclose( fp );
                        }
                        else
                                exit(1);
        return 0;
}
void do_more( FILE *fp )
/*
 * read PAGELEN lines, then call see_more() for further instructions
 */
{
       char    line[LINELEN];
       int     num_of_lines = 0;
       int     see_more(), reply;

       while ( fgets( line, LINELEN, fp ) ){          /* more input    */
              if ( num_of_lines == PAGELEN ) {        /* full screen? */
                      reply = see_more();             /* y: ask user  */
                      if ( reply == 0 )               /*    n: done   */
                              break;
                      num_of_lines -= reply;          /* reset count  */
              }
              if ( fputs( line, stdout ) == EOF )     /* show line    */
                      exit(1);                        /* or die       */
              num_of_lines++;                         /* count it     */
       }
```

```
}
int see_more()
/*
 *     print message, wait for response, return # of lines to advance
 *     q means no, space means yes, CR means one line
 */
{
        int      c;

        printf("\033[7m more? \033[m");          /* reverse on a vt100  */
        while( (c=getchar()) != EOF )                  /* get response */
        {
                if ( c == 'q' )                  /* q -> N              */
                        return 0;
                if ( c == ' ' )                  /* ' ' => next page    */
                        return PAGELEN;          /* how many to show    */
                if ( c == '\n' )                 /* Enter key => 1 line */
                        return 1;
        }
        return 0;
}
```

The code consists of three functions. The `main` function decides whether to read data from a file or from standard input. Having settled on an input stream, `main` passes that input stream to the function called `do_more` to display that stream screenful by screenful. The `do_more` function, in turn, displays a screenful of text and then calls the function `see_more` to ask the user what to do next.

Compile and run it as follows:

```
$ cc more01.c -o more01
$ more01 more01.c
```

This program works pretty well. The program displays 24 lines of the file and prints, in eye-catching reverse video, the prompt more?. Press the Enter key to advance the display one line. This program needs more work.

In particular, the more? message stays on the screen and scrolls up with the text. Also, if you press the space bar or the "q" key, nothing happens until you press Enter. That's not so good. Also, the little more? message is still there.
Writing a version of `more` illustrates the basic fact about Unix programming:

> **Unix programming is not as difficult as you think it is, but it is not as easy as you first imagine.**

The program performs a clearly defined task. The logic behind that task is pretty clear from its action. Devising an algorithm that performs that action is not all that tricky.

Then we begin to see subtle problems. How do we get the program to respond to keystrokes without having to press Enter? How do we figure out the percent of the file we have displayed? How do we erase the more? prompt after the user presses a key?

That cannot be too tricky, but first we need to finish comparing other features. How well does our program handle input streams? The main function checks the number of command-line arguments. If no filenames are specified on the command line, the program reads from standard input. That makes it possible to put `more` at the end of a pipeline, as in

```
$ who | more
```

This *pipeline* runs the `who` command to list all users on the system and sends that list of users into the `more` command. The `more` program displays 24 lines at a time, a useful thing if the number of users exceeds 24. Let us test our program, not with `who`, but with `ls`:

```
$ ls /bin | more01
```

We expect to see the contents of the `/bin` directory, 24 lines at a time.

When you run this, you will see that `more01` does *not* pause after 24 lines. What could have gone wrong? Here is the reason. Our `more01` program reads and prints 24 lines of input from the `ls` command. When `more01` reads the 25th line, it prints the more? prompt and waits for user input. Our program waits for the user to press the space bar, the Enter key or the letter "q."

Where does that user input come from? The program uses `getchar`, which reads from standard input. But the notation

```
$ ls /bin | more01
```

attaches the standard input of `more01` to the output of `ls`. Our version of `more` tries to read user commands from the same stream as the data. The following picture shows the situation:

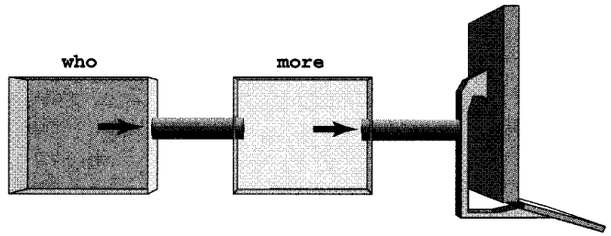

FIGURE 1.14

`more` reads `stdin`.

How does the real `more` solve this problem? That is, how can the program read data from standard input and still get user input from the keyboard? The answer is to read from the keyboard directly. Figure1.15 illustrates what the real version of `more` does.

There is a special file in every Unix system called `/dev/tty`. This file is actually a connection to the keyboard and screen. Even if the user changes the standard input or standard output of a program by using the < or > characters, the program can still communicate with the terminal by reading and writing data to the `/dev/tty` file.

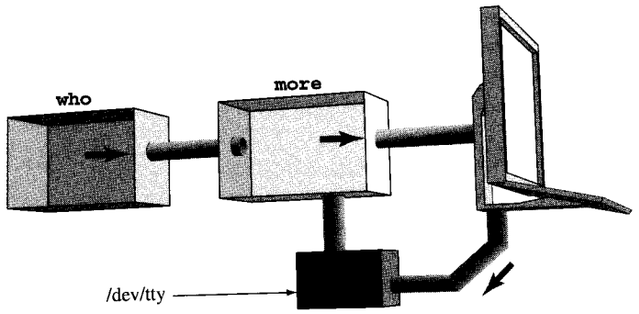

FIGURE 1.15

who reads user input from a terminal.

The diagram shows more has two sources of input. The standard input of the program is attached to the output of who. But more also reads data from /dev/tty. more reads lines from the file and displays them on the screen. When it needs to ask the user whether to display one more line, one more page, or to quit, it reads the input from /dev/tty.

With this new knowledge, we can enhance more01.c to more02.c:

```
/* more02.c - version 0.2 of more
 *      read and print 24 lines then pause for a few special commands
 *      feature of version 0.2: reads from /dev/tty for commands
*/
#include        <stdio.h>

#define PAGELEN 24
#define LINELEN 512

void do_more(FILE *);
int see_more(FILE *);

int main( int ac , char *av[] )
{
        FILE *fp;

        if ( ac == 1 )
                do_more( stdin );
        else
                while ( --ac )
                        if ( (fp = fopen( *++av , "r" )) != NULL )
                        {
                                do_more( fp ) ;
                                fclose( fp );
                        }
                        else
                                exit(1);

        return 0;
}
void do_more( FILE *fp )
```



```
/*
 * read PAGELEN lines, then call see_more() for further instructions
 */
{
        char    line[LINELEN];
        int     num_of_lines = 0;
        int     see_more(FILE *), reply;
        FILE    *fp_tty;

        fp_tty = fopen( "/dev/tty", "r" );          /* NEW: cmd stream  */
        if ( fp_tty == NULL )                       /* if open fails    */
                exit(1);                            /* no use in running*/

        while ( fgets( line, LINELEN, fp ) ){       /* more input   */
                if ( num_of_lines == PAGELEN ) {    /* full screen? */
                        reply = see_more(fp_tty);   /* NEW: pass FILE * */
                        if ( reply == 0 )           /*    n: done   */
                                break;
                        num_of_lines -= reply;      /* reset count */
                }
                if ( fputs( line, stdout ) == EOF ) /* show line    */
                        exit(1);                    /* or die       */
                num_of_lines++;                     /* count it     */
        }
}
int see_more(FILE *cmd)                             /* NEW: accepts arg */
/*
 * print message, wait for response, return # of lines to advance
 * q means no, space means yes, CR means one line
 */
{
        int     c;
        printf("\033[7m more? \033[m");             /* reverse on a vt100  */
        while( (c=getc(cmd)) != EOF )               /* NEW: reads from tty */
        {
                if ( c == 'q' )                     /* q -> N              */
                        return 0;
                if ( c == ' ' )                     /* ' ' => next page    */
                        return PAGELEN;             /* how many to show    */
                if ( c == '\n' )                    /* Enter key => 1 line */
                        return 1;
        }
        return 0;
}
```

Compile and test this version:

```
$ cc -o more02 more02.c
$ ls /bin | more02
```

This version, `more02.c` can read data from standard input and still read commands from the keyboard. Notice how trying to write a standard Unix program led us to learn about the file called `/dev/tty` and its role as a direct connection to the user's terminal.

Our program still needs work. We still have to press the Enter key to get the program to respond. Also, the "q" and space characters show up on the screen. Somehow, the real version of `more` changes input so characters are delivered immediately to the program; the Enter key is not needed. The real `more` also arranged it so characters you type do not show up. If you press "q", the program quits, but you do not see the letter "q" on the screen.

Instant Input: How Does That Work?

It turns out that connections to terminals have settings. You can adjust the settings of the connection so characters are delivered as they are typed instead of only after the user presses Enter. You can adjust the settings so the characters the user types do not show up on the screen. You can adjust all sorts of settings that control the way the terminal delivers data to your program.

As we delve into this problem, more details emerge in our picture. Now, it looks like the following:

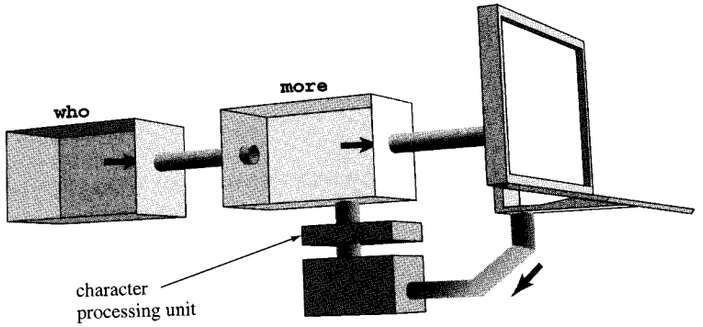

FIGURE 1.16

The connection to the terminal has settings.

The new item in this picture is the control unit added to the connection to `/dev/tty`. That control unit lets a programmer adjust how the connection between the program and the terminal works, the way dials and switches on a radio let you adjust how the connection between the tuner and the speaker works.

To write a complete, smoothly operating version of `more`, we need to study that connection control unit and how to program it.

We also need to answer some other questions. How do we find the percent of the file shown? The real version of `more` displays the percent of the file the user has seen. How can we add that feature? The operating system knows how large the file is. We just need to learn how to ask the operating system for that information.

What about the reverse video? What about the number of lines? Some displays use different methods to create reverse-video text. Different displays have different numbers of lines. A fixed size of 24 lines and the vt100-specific reverse-video codes lack flexibility. How can we write a version of `more` that works for any terminal type and any number of lines? We need to learn about terminal screen control and attributes.

1.7 A LOT OF QUESTIONS AND A MAP

1.7.1 What Now?

We have defined the project for this book. Unix is an operating system that allows several people to use the system at the same time. Users can run programs and work with files and directories. These programs can communicate with one another, within one computer, and across networks. Users run programs to manage their files, process data, transfer and transform data, and communicate with other users.

How do all these programs work? What do the programs do? What does the operating system do? As we explored the main features of the system, we asked many questions.

We now begin to answer these questions. Our case study of the `more` command demonstrates the approach we shall take. We look at a real program, study what it does, and then try to write our own version of it. In the process, we discover more details of how Unix works and learn to use those principles.

1.7.2 And Now for the Map

We need a map for our journey. Here it is:

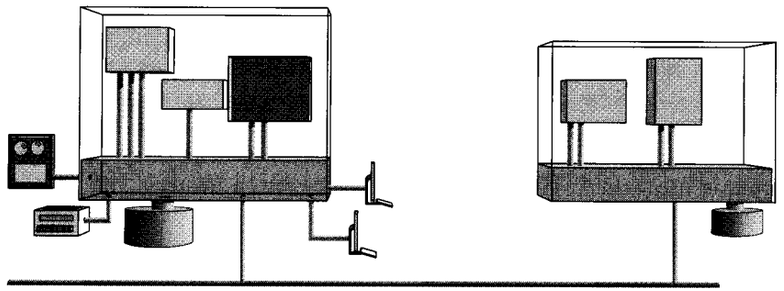

FIGURE 1.17

A diagram of the main structure of a Unix system.

This diagram of a Unix system depicts the main structure of any Unix system. The memory is divided into system space and user space. The kernel and its data structures occupy system space. User processes live in user space. Some users connect to the system through terminals; the lines to these terminals connect to the kernel. Files are stored in a file system on a disk. Devices of various sorts are attached to the kernel and made available to user processes. Finally, there is a network connection. Some users connect to the system through a network connection.

In each section of the book, we focus on parts of this diagram. We zoom in on each component to explore services the kernel offers and to explain the logic and data structures the kernel uses to provide those services.

By the end of the book, you will have studied each part of this picture and will have seen all the ideas and techniques needed to write complete Unix system programs—Internet bridge, for example.

1.7.3 What Is Unix? History and Dialects

This book explains basic ideas and structures of Unix and shows how to write programs that work in a Unix system. But what is Unix? Where did it come from? What exactly can you expect from this book?

First, where did Unix come from? Unix started as a kernel and set of tools created around 1969 at Bell Laboratories by a few computer scientists to solve specific technical problems. Unix was not a commercial product. In fact, during the 1970s, Bell Labs distributed Unix software, including complete source code, to schools and research centers for nominal fees. Researchers at Bell Labs and many other computer scientists spent years learning from, improving, and adding to the original programs. During the 1980s, several companies licensed Unix source code and built customized versions of the system. The two main centers of Unix development were AT&T and the University of California at Berkeley. AT&T developed a version called System V, and UCB developed versions collectively called BSD. Most varieties of Unix derived from one or both of these main versions. Over the years, ownership passed, in a sequence of sales, from AT&T through a sequence of companies, UCB stopped working on Unix, and various groups tried to reconcile and standardize the system.

Independent of deals and standards, the basic design and principles of Unix spread through academic and commercial computing. Different dialects and models of the system evolved. Some versions include specialized features, like real-time processing. Through all these adaptations and changes, Unix always retained a core architecture and consistent set of functions. Although the exact internal structure and set of tools in a version of Unix from AT&T in 1980 differ from those in a version of Unix written in Helsinki in 1991, system programs written for that 1980 version can, with minimal changes, compile and run on the Finnish version.

What, then, is Unix? The term a *Unix system*, increasingly is used to refer to systems that follow the core structure and provide the functions common to all these variations. Some systems look like and work like Unix, but are not derived from AT&T or UCB code. The GNU/Linux combination of tools and kernel is a well-known Unix-like system. One formal description of the system interface is called POSIX. To understand, read, and write Unix programs, you need to know more than a single standard, though.

Unix has a long and varied history; how much of Unix can you expect to learn about from one book? We concentrate on structure, principles, and techniques common to all Unix systems. Some details are left out, some operations are duplicated, and all ideas are explained in practical contexts.

I do not include every detail, and sometimes I suggest you check local documentation. This book is not a comprehensive reference on every aspect of every version of Unix. In fact, exploring and using on-line documentation to learn about your system is an essential part of knowing about Unix.

Sometimes I describe different functions that do the same thing. One cause of duplication of functions is the decentralized growth of Unix. Different groups, like the ones at AT&T and UCB, sometimes devised different solutions to the same problem. Another cause of duplicate functions is normal growth. When people replace a Unix service, such as alarm timers, with a more flexible version, they do not want to break existing programs, so they rarely remove the older, simpler interface. Sometimes I mention one solution, sometimes more than one. As you study Unix programs, you will encounter these various solutions. Studying various approaches can clarify the fundamental ideas and help you adapt to local variations.

Finally, I present Unix in the context of actual software projects. Unix is a system of ideas and techniques created by people building solutions to real problems. We start with real problems and see how the ideas provide solutions. Unix makes sense when you see how the pieces work as a system.

SUMMARY

- A computer system includes several types of resources such as disk storage, memory, peripheral devices, and network connections. Programs use these resources to store, transfer, and process data.
- Computer systems that run several programs for several users at the same time require a central management program. The Unix kernel is a program that schedules programs and controls access to resources.
- User programs ask the kernel for access to resources.
- Some Unix programs consist of separate programs that share or exchange data.
- Writing systems programs requires an understanding of the structure and use of kernel services.