6

USING MINIX

By now you should have installed MINIX and gotten to the point where you can use it. In this chapter we will discuss some basic and some less basic points about using it. Once again, if you are not already reasonably familiar with using UNIX, you should first read one of the many books about it.

As a general rule, most aspects of MINIX work the same way as they do in UNIX. When you log in, you get a shell, which is functionally similar to the standard V7 shell (Bourne shell). Most programs are called the same way as in UNIX, have the same flags, and perform the same functions as their UNIX counterparts. The (default) keyboard editing conventions are also similar to V7 UNIX: the backspace key (CTRL-H) is used to correct typing errors, the @ symbol is used to erase the current input line, CTRL-S is used to stop the screen from scrolling out of view, CTRL-Q is used to start the screen moving again, and CTRL-D is used to indicate end-of-file from the keyboard., for example, to log out. These key bindings can be changed using the IOCTL system call and *stty* program, the same way as in UNIX.

6.1. MAJOR COMPONENTS OF MINIX

Although MINIX consists of hundreds of files, programs, and procedures, from the user's perspective, a few of them stand out as being especially important. In this section we will take a quick look at a few of the most important ones.

6.1.1. The Shell

The MINIX command interpreter is functionally identical to the Version 7 command interpreter, known as the **shell** (or the **Bourne shell** in honor of its inventor, S. R. Bourne). When a user logs in, the shell starts out by displaying the **prompt**, a character such as a dollar sign, which tells the user that the shell is waiting to accept a command. If the user now types:

```
date
```

for example, the shell sees to it that the *date* program is run. When *date* finishes, the shell types the prompt again and tries to read the next input line.

The user can specify that standard output be redirected to a file by typing, for example:

```
date >file
```

Similarly, standard input can be redirected, as in:

```
sort <file1 >file2
```

which invokes the sort program with input taken from *file1* and output sent to *file2*.

The output of one program can be used as the input for another program by connecting them with a pipe. Thus

```
cat file1 file2 file3 | sort >outfile
```

invokes the *cat* program to con*cat*enate three files and send the output to *sort* to arrange all the lines in alphabetical order. The output of *sort* is redirected to the file *outfile*.

If a user puts an ampersand after a command, the shell does not wait for it to complete. Instead it just gives a prompt immediately. Consequently:

```
cat file1 file2 file3 | sort >outfile &
```

starts up the sort as a background job, allowing the user to continue working normally while the sort is going on.

It is possible to collect several commands together in a file called a **shell script** and have them executed by just typing the name of the shell script. The shell also recognizes some programming constructs, such as *if*, *for*, *while*, and *case*, so it is possible to write shell scripts that act like programs. For more information about the MINIX shell, consult any book about the UNIX system because the MINIX and Bourne shells are practically indistinguishable to the user (although they are very different internally).

6.1.2. Editors

MINIX comes with several editors, among them a line-oriented editor called *ed*, a simple full-screen editor called *mined*, a powerful multifile, multiwindow editor called *elle*, and a clone of the well-known Berkeley *vi* editor, called *elvis*. People often have very strong, almost emotional, attachments to particular editors, so we have provided a wide choice. Try them all and see which you like best.

The *ed* editor is based on the V7 editor used on old mechanical teletypes. Although it still useful under certain circumstances, for daily use, it is rarely used nowadays.

In contrast, *mined* is a simple, but modern full-screen editor. Its greatest virtue is that it can be learned in about 10 minutes. When you type an ordinary ASCII character, that character is inserted on the screen (and in the file being edited) at the position of the cursor. This may sound obvious, but many editors require you to first enter a special "insert mode," enter the text, and then leave insert mode.

Commands to *mined*, such as moving the cursor or terminating the edit session, are handled by control characters, such as CTRL-F (go forward one word) or by the keys such as the four arrows on the numeric keypad at the right-hand side of the IBM PC keyboard. There are about three dozen commands in all, mostly chosen for their mnemonic value (e.g., CTRL-A moves the cursor to the start of the current line; CTRL-Z moves it to the end of the line).

Some of the commands move the cursor around the screen, scroll the screen forward or backward, or position it at the beginning or end of the file. Other commands delete text around the cursor (e.g., delete the word to the left or right of the cursor, or delete the tail of the current line). There are also commands available to manipulate blocks of text, such as deleting a block of text or saving it in a buffer to be copied to another part of the file. Finally, there are commands for searching forward or backward for a given text pattern, where the text pattern may contain a mixture of ordinary ASCII characters and "wild card" characters for matching sets of characters, end-of-line, and so on.

Another editor is *elle*, which can be thought of as a fast, simplified version of the famous *emacs* editor. It has about 100 commands, and can edit multiple files in full-screen or split-screen mode. Many sophisticated users regard *emacs* as the last word in editing.

Finally, there is *elvis*, an editor that has nearly all the features of the Berkeley *vi* and *ex* editors. All four editors have different properties. If you do not already have a preference, try them all until you find one you like.

6.1.3. The C Compiler

MINIX comes with a C compiler that accepts programs written in C as described in the Kernighan and Ritchie book. It also accepts many nonstandard features that are commonly used, but gives a warning message about each of them when asked

to. It also provides the standard header files normally provided with C compilers. The command:

```
cc prog.c
```

compiles the program on the file *prog.c* and leaves the executable binary program on a file called *a.out*.

The compiler knows about most of the standard C compiler flags, including **–c** (compile but do not link), **–o** (put the compiler output on a specific file instead of *a.out*), **–D** (define a macro), and **–I** (search a given directory for include files). Like the Version 7 compiler, this one also has a preprocessor for *#define*, *#include*, and *#ifdef* statements.

One minor difference between the MINIX compiler for the IBM PC and most other C compilers is that this one produces *.s* rather than *.o* files as a result of the **–c** flag. Furthermore, the assembler and linker are combined into a single program, *asld*, that reads a list of *.s* files and possibly some library archives, and produces an executable file. The members of the library archives are also *.s* files, although both they and the compiler output are compacted to save time and space. The programs *libpack* and *libupack* are provided to convert assembly language files from ASCII to compact format and back. The C compilers for the 68000 machines produce normal *.o* files.

6.1.4. The Utility Programs

MINIX comes with more than 175 utility programs. One rough grouping is to classify them into five categories as follows:

1. Compiler utilities.
2. File and directory manipulation.
3. Text file processing.
4. System administration.
5. Miscellaneous.

The compiler utilities are programs such as *make*, for keeping track of interdependent source and object files; *ar*, for maintaining libraries; and *size*, for determining the size of the various segments in a binary program.

The file and directory manipulation programs include *cat*, *cp*, *dd*, *mv*, and *pr*, for moving files around; *mkdir*, *rmdir*, and *ls*, for managing directories; and *chmod*, and *chown*, for dealing with protection.

A variety of programs are present for working with text files in addition to the editors, including the well-known filters *grep*, *rev*, *sort*, *tr*, *uniq*, and *wc*. The program *gres* searches a set of files for a pattern, and replaces occurrences with a given

pattern. The MINIX text justifiers are *roff*, and *nroff*, which have a wide variety of commands for controlling page layout.

Some utility programs deal with system administration. These include *df*, for determining how much space a file system has, *mkfs*, for making new file systems, *mount* and *umount* for attaching and detaching file systems to the main file tree, *passwd* for changing passwords, and *su* for becoming superuser.

The last category is for programs that do not fit in anywhere else. Among these are *date*, for setting and displaying the date and time, *pwd*, for printing the working directory, and *stty*, for setting the terminal parameters. There are many more.

6.1.5. The Library Procedures

MINIX also comes with over 200 library procedures that can be called from C programs. Like the utilities, these can also be divided into several rough groups:

1. System calls.
2. ANSI C procedures.
3. Miscellaneous.

The system call procedures allow C programs to issue system calls. There are more than 40 system calls available, including OPEN, READ, WRITE, CLOSE, LSEEK, PIPE, FORK, and EXEC. For almost every system call, there is a library procedure with exactly the same parameters and results as in Version 7. It should be possible to take almost any portable C program that runs under Version 7 and compile and run it on MINIX. Furthermore, most reasonable C programs written for other versions of UNIX should also work on MINIX, provided that they do not use any of the more bizarre system calls available in other versions and do not make any implicit assumptions about the sizes of integers and pointers (which are not the same on the 68000 versions of MINIX).

The second category is the set of procedures defined ANSI C. The collection is not yet complete, but most of the more common procedures are present, including standard I/O and string handling. Calls such as *fopen*, *fread*, *fwrite*, *fclose*, and *fprintf* are all present, as are *strcat*, *strcmp*, *strcpy*, and *strlen*, to name just a few.

The last category consists of a mixture of other procedures, which span a wide range, from encryption (*crypt*) to temporary file creation (*mktemp*).

6.1.6. Relation with Other Operating Systems

Like UNIX, MINIX is a complete operating system. It does not require any other operating system to help it. On the IBM PC, Atari, and Amiga, when MINIX is running it takes over the entire computer and runs on the bare hardware. For the Macintosh version, this is not true. There MINIX runs as a user program on top on

the Macintosh operating system. Since MINIX has different system calls than MS-DOS, TOS, and AMIGA-DOS, it is not possible to run programs written for other operating systems on MINIX.

Nevertheless, it is possible to partition your hard disk with one or more partitions for MINIX and one or more partitions for other operating systems. Furthermore, you can transport files back and forth between MINIX and MS-DOS, TOS, or AMIGA-DOS, using utility programs have been provided for this purpose. The utilities reside in */usr/bin* and are invoked in the usual way, by just typing their names and arguments. The first program for the IBM PC, *dosdir*, reads an MS-DOS diskette and tells what is on it. The program can also be told to list a specific directory on the diskette.

The second program, *dosread*, reads a file from an MS-DOS diskette and copies it to standard output, which, of course, can be redirected to a file. When the **–a** flag is given, the MS-DOS conventions for ASCII files are converted to the MINIX conventions, so the resulting file appears to be a normal text file.

The third program, *doswrite*, copies its standard input to a diskette containing an MS-DOS file system, again doing format conversion if requested. It does not create directories, however, so all the necessary directories must be in place on the diskette when it is inserted into the drive. As an aside, these three programs are all links to the same file, which checks to see how it was called to see what it should do.

For the 68000-based systems, analogous programs are provided to get files back and forth.

6.2. PROCESSES AND FILES IN MINIX

Two key concepts in MINIX are processes and files. Since these may not be immediately familiar to people who are accustomed to other operating systems, below we give a brief introduction to them. Processes and files are accessed by **system calls**, services provided by the operating system. Some of the more important process and file system calls will be discussed below.

6.2.1. Processes

A **process** is basically a program in execution. It consists of the executable program, the program's data and stack, its program counter, stack pointer, and other registers, and all the other information needed to run the program.

The easiest way to get a good intuitive feel for a process is to think about a timesharing system. Periodically, the operating system decides to stop running one process and start running another, for example, because the first one has had more than its share of CPU time in the past second.

When a process is temporarily suspended like this, it must later be restarted in

exactly the same state it had when it was stopped. This means that all information about the process must be explicitly saved somewhere during the suspension. For example, if the process has several files open, the exact position in the files where the process was must be recorded somewhere, so that a subsequent READ given after the process is restarted will read the proper data. In many operating systems, all the information about each process, other than the contents of its own address space, is stored in an operating system table called the **process table**, which is an array (or linked list) of structures, one for each process currently in existence.

Thus, a (suspended) process consists of its address space, usually called the **core image** (in honor of the magnetic core memories used in days of yore), and its process table entry, which contains its registers, among other things.

The key process management system calls are those dealing with the creation and termination of processes. Consider a typical example. The shell reads commands from a terminal. The user has just typed a command requesting that a program be compiled. The shell must now create a new process that will run the compiler. When that process has finished the compilation, it executes a system call to terminate itself.

If a process can create one or more other processes (referred to as **child processes**) and these processes in turn can create child processes, we quickly arrive at the process tree structure of Fig. 6-1.

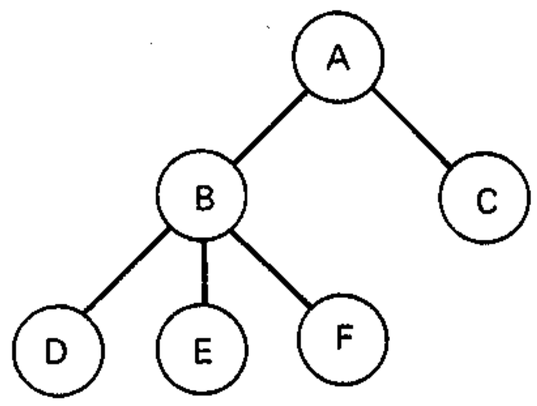

Fig. 6-1. A process tree. Process *A* created two child processes, *B* and *C*. Process *B* created three child processes, *D*, *E*, and *F*.

Other process system calls are available to request more memory (or release unused memory), wait for a child process to terminate, and overlay its program with a different one.

Occasionally, there is a need to convey information to a running process that is not sitting around waiting for it. For example, a process that is communicating with another process on a different computer does so by sending messages over a network. To guard against the possibility that a message or its reply is lost, the sender may request that its own operating system notify it after a specified number of seconds, so that it can retransmit the message if no acknowledgement has been received yet. After setting this timer, the program may continue doing other work.

When the specified number of seconds has elapsed, the operating system sends a **signal** to the process. The signal causes the process to temporarily suspend whatever it was doing, save its registers on the stack, and start running a special signal

handling procedure, for example, to retransmit a presumably lost message. When the signal handler is done, the running process is restarted in the state it was just before the signal. Signals are the software analog of hardware interrupts, and can be generated by a variety of causes in addition to timers expiring. Many traps detected by hardware, such as executing an illegal instruction or using an invalid address, are also converted into signals to the guilty process.

Each person authorized to use MINIX is assigned a **uid** (user identification) by the system administrator. Every process started in MINIX has the uid of the person who started it (except for so-called setuid programs). A child process has the same uid as its parent. One uid, called the **superuser**, has special power, and may violate many of the protection rules. In large installations, only the system administrator knows the password needed to become superuser, but many of the ordinary users (especially students) devote considerable effort to trying to find flaws in the system that allow them to become superuser without the password.

6.2.2. Files

A major function of the operating system is to hide the peculiarities of the disks and other I/O devices, and present the programmer with a nice, clean abstract model of device-independent files. System calls are obviously needed to create files, remove files, read files, and write files. Before a file can be read, it must be opened, and after it has been read it should be closed, so calls are provided to do these things.

In order to provide a place to keep files, MINIX has the concept of a **directory** as a way of grouping files together. A student, for example, might have one directory for each course he was taking (for the programs needed for that course), another directory for his electronic mail, and still another directory for his computer games. System calls are then needed to create and remove directories. Calls are also provided to put an existing file in a directory, and to remove a file from a directory. Directory entries may be either files or other directories. This model also gives rise to a hierarchy—the file system, as shown in Fig. 6-2.

The process and file hierarchies both are organized as trees, but the similarity stops there. Process hierarchies usually are not very deep (more than three levels is unusual), whereas file hierarchies are commonly four, five, or even more levels deep. Process hierarchies are typically short-lived, generally a few minutes at most, whereas the directory hierarchy may exist for years. Ownership and protection also differ for processes and files. Typically, only a parent process may control or even access a child process, but mechanisms exist to allow files and directories to be read by a wider group than just the owner.

Every file within the directory hierarchy can be specified by giving its **path name** from the top of the directory hierarchy, the **root directory**. Such absolute path names consist of the list of directories that must be traversed from the root directory to get to the file, with slashes separating the components. In Fig. 6-2, the

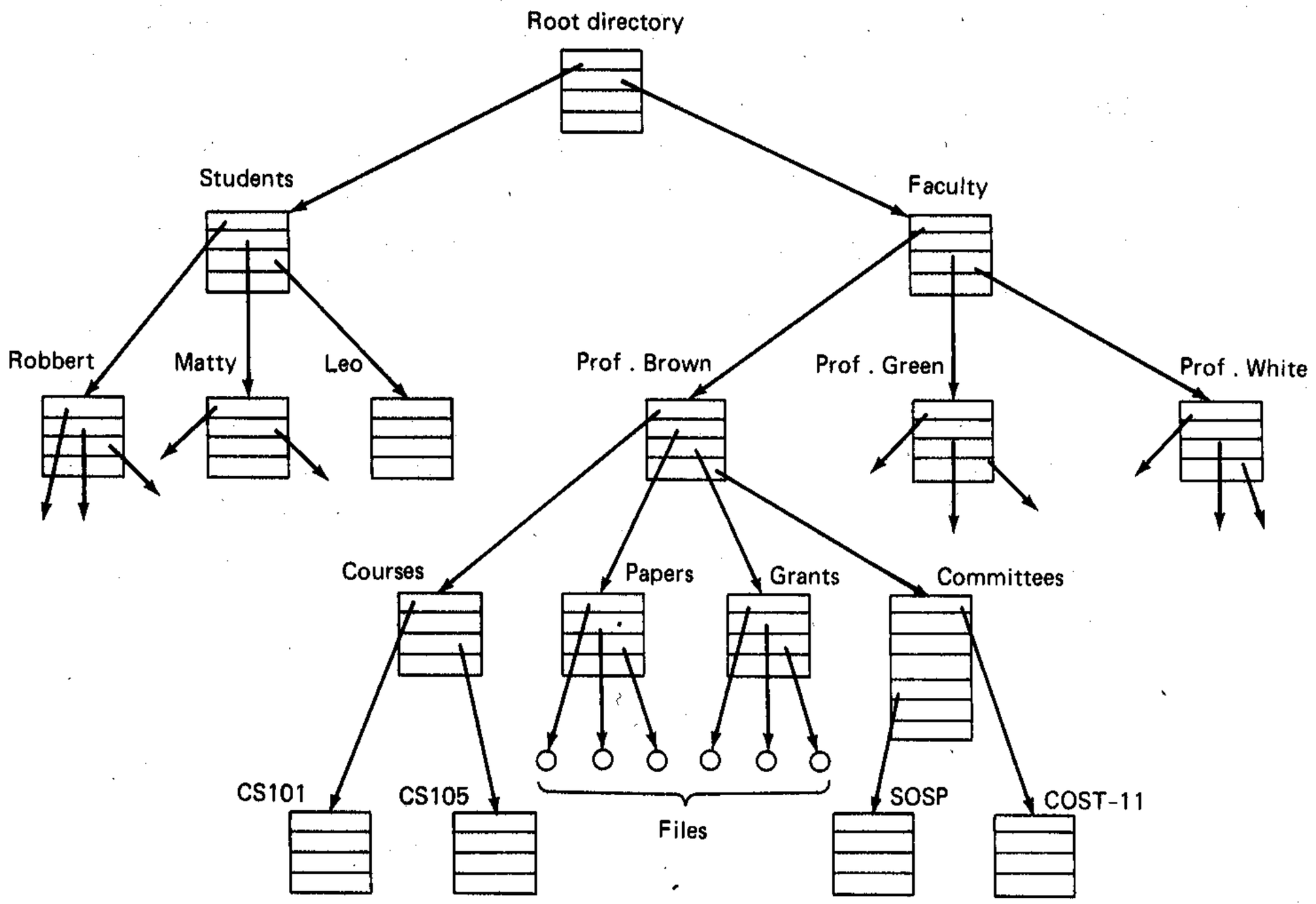

Fig. 6-2. A file system for a university department.

path for file *CS101* is */Faculty/Prof.Brown/Courses/CS101*. The leading slash indicates that the path is absolute, that is, starting at the root directory (as opposed to a relative path starting at the working directory).

At every instant, each process has a current **working directory**, in which path names not beginning with a slash are looked for. In Fig. 6-2, if */Faculty/Prof.Brown* were the working directory, then use of the path name *Courses/CS101* would yield the same file as the absolute path name given above. Processes can change their working directory by issuing a system call specifying the new working directory.

Files and directories in MINIX are protected by assigning each one a 9-bit binary protection code. The protection code consists of three 3-bit fields, one for the owner, one for other members of the owner's group (users are divided into groups by the system administrator), and one for everyone else. Each field has a bit for read access, a bit for write access, and a bit for execute access. These 3 bits are known as the **rwx bits**. For example, the protection code *rwxr-x--x* means that the owner can read, write, or execute the file, other group members can read or execute (but not write) the file, and everyone else can execute (but not read or write) the file. For a directory, *x* indicates search permission. A dash means that the corresponding permission is absent.

Before a file can be read or written, it must be opened, at which time the permissions are checked. If the access is permitted, the system returns a small integer

called a **file descriptor** to use in subsequent operations. If the access is prohibited, an error code is returned.

Another important concept in MINIX is the **mounted file system**. To provide a clean way to deal with removable media (e.g. diskettes), MINIX allows the file system on a diskette to be attached to the main tree. Consider the situation of Fig. 6-3(a). Before the MOUNT call, the RAM disk (simulated disk in main memory) contains the primary, or **root file system**, and drive 0 contains a diskette containing another file system.

However, the file system on drive 0 cannot be used, because there is no way to specify path names on it. MINIX does not allow path names to be prefixed by a drive letter or number; that would be precisely the kind of device dependence that operating systems ought to eliminate. Instead, the MOUNT system call allows the file system on drive 0 to be attached to the root file system wherever the program wants it to be. In Fig. 6-3(b) the file system on drive 0 has been mounted on directory *b*, thus allowing access to files */b/x* and */b/y*. If directory *b* had contained any files they would not be accessible while drive 0 was mounted, since */b* would refer to the root directory of drive 0. (Not being able to access these files is not as serious as it at first seems: file systems are nearly always mounted on empty directories.)

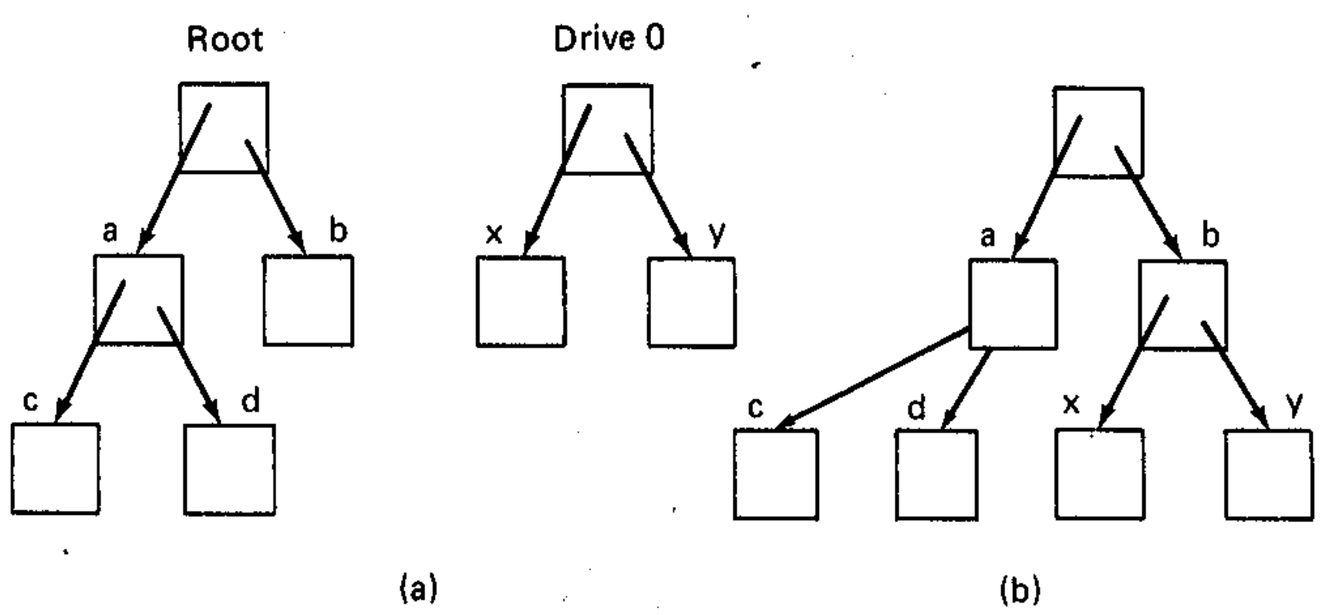

Fig. 6-3. (a) Before mounting, the files on drive 0 are not accessible. (b) After mounting, they are part of the file hierarchy.

Another important concept in MINIX is the **special file**. Special files are provided in order to make I/O devices look like files. That way, they can be read and written using the same system calls as are used for reading and writing files. Two kinds of special files exist: **block special files** and **character special files**. Block special files are used to model devices that consist of a collection of randomly addressable blocks, such as disks. By opening a block special file and reading, say, block 4, a program can directly access the fourth block on the device, without regard to the structure of the file system contained on it. Programs that do system maintenance often need this facility. Access to special files is controlled by the same *rwx* bits used to protect all files, so the power to directly access I/O devices can be restricted to the system administrator, for example.

Character special files are used to model devices that consist of character streams, rather than fixed-size randomly addressable blocks. Terminals, line

printers, and network interfaces are typical examples of character special devices. The normal way for a program to read and write on the user's terminal is to read and write the corresponding character special file. When a process is started up, file descriptor 0, called **standard input**, is normally arranged to refer to the terminal for the purpose of reading. File descriptor 1, called **standard output**, refers to the terminal for writing. File descriptor 2, called **standard error**, also refers to the terminal for output, but normally is used only for writing error messages.

All special files have a **major device number** and a **minor device number**. The major device number specifies the device class, such as diskette, hard disk, or terminal. The minor device number specifies which of the devices in the class is being addressed, for example, which diskette drive. All devices with the same major device number share the same device driver code within the operating system. The minor device number is passed as a parameter to the device driver to tell it which device to read or write. The device numbers can be seen by listing */dev* with the *ls –l* command.

The last feature we will discuss in this overview is one that relates to both processes and files: pipes. A **pipe** is a sort of pseudo-file that can be used to connect two processes together, as shown in Fig. 6-4. When process *A* wants to send data to process *B*, it writes on the pipe as though it were an output file. Process *B* can read the data by reading from the pipe as though it were an input file. Thus, communication between processes in MINIX looks very much like ordinary file reads and writes. Stronger yet, the only way a process can discover that the output file it is writing on is not really a file, but a pipe, is by making a special system call.

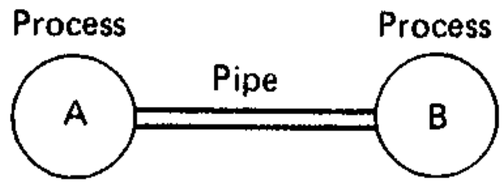

Fig. 6-4. Two processes connected by a pipe.

6.3. A TOUR THROUGH THE MINIX FILE SYSTEM

The MINIX file tree is organized the same way as the standard UNIX file tree. The standard MINIX file system contains the following directories:

Name	**- Description**
/bin	- Most common system binaries can be copied here from */usr/bin*
/dev	- Special files for I/O devices
/etc	- Miscellaneous system administration
/doc	- Place to put (user-supplied) online documentation
/lib	- Most common libraries can be copied here from */usr/lib*
/tmp	- Some utilities generate their temporary files here

/user	- Empty; can be used for mounting file systems
/usr	- Root of the user file system (usually mounted file system)
/usr/adm	- The */usr/adm/wtmp* file records logins
/usr/ast	- Home directory for user *ast*
/usr/bin	- System binaries are kept here
/usr/etc	- Main system administration directory
/usr/include	- System header files
/usr/include/minix	- MINIX-specific header files
/usr/include/sys	- More header files
/usr/lib	- Libraries, compiler passes, miscellanea
/usr/lib/tmac	- Holds macro packages for *nroff*
/usr/man	- Place to put user-written manual pages for *man* (if any)
/usr/spool	- Holds specialized spooling directories
/usr/spool/at	- Spooling directory for the *at* program
/usr/spool/lpd	- Spooling directory for line printer daemons (future)
/usr/spool/mail	- Spooling directory for local mail
/usr/spool/uucp	- Spooling directory for *kermit* and *uucp* (future)
/usr/src	- Start of the source tree
/usr/src/commands	- Sources for the utility programs (has many subdirectories)
/usr/src/fs	- Sources for MINIX file system
/usr/src/lib	- Holds library directories
/usr/src/lib/amiga	- Sources for Amiga-specific procedures
/usr/src/lib/ansi	- Sources for ANSI C procedures
/usr/src/lib/atari	- Sources for Atari-specific procedures
/usr/src/lib/ibm	- Sources for IBM PC-specific procedures
/usr/src/lib/mac	- Sources for Macintosh-specific procedures
/usr/src/lib/other	- Sources for other library procedures
/usr/src/lib/posix	- Sources for procedures required by POSIX
/usr/src/lib/string	- Sources for IBM assembly code string procedures
/usr/src/kernel	- Sources for MINIX kernel
/usr/src/mm	- Sources for MINIX memory manager
/usr/src/test	- Sources and binaries for testing MINIX
/usr/src/tools	- Utilities for building MINIX boot diskettes
/usr/tmp	- Alternative directory for temporary files

Let us briefly examine some of these directories. In */bin* we find the most heavily used programs such as *cat*, *cp*, and *ls* as well as some programs such as *login* and *sh* needed to bring the system up. If */bin* is being kept on RAM disk, it will normally contain a subset of */usr/bin*. The idea of putting it on the RAM disk is to speed up access, of course. If a RAM disk is not being used, it is not necessary to put any files in *bin* other than the ones it comes with.

The directory */dev* contains the special files for the I/O devices, including most of the following, although not every one is present in each version. Ethernet is not

supported on the 68000, for example. Also, */dev/hd5-9* are for an (optional) second hard disk.

Name	#	Description
/dev/ram	1, 0	- RAM disk
/dev/mem	1, 1	- Absolute memory
/dev/kmem	1, 2	- Kernel memory
/dev/null	1, 3	- Data written here vanishes; reads yield end of file
/dev/port	1, 4	- Access to I/O ports
/dev/fd0	2, 0	- Diskette drive 0
/dev/fd1	2, 1	- Diskette drive 1
/dev/hd0	3, 0	- IBM: Entire hard disk 0; Atari: boot block
/dev/hd1	3, 1	- Hard disk 0, partition 1
/dev/hd2	3, 2	- Hard disk 0, partition 2
/dev/hd3	3, 3	- Hard disk 0, partition 3
/dev/hd4	3, 4	- Hard disk 0, partition 4
/dev/hd5	3, 5	- IBM: Entire hard disk 1; Atari entire hard disk 0
/dev/hd6	3, 6	- Hard disk 1, partition 1
/dev/hd7	3, 7	- Hard disk 1, partition 2
/dev/hd8	3, 8	- Hard disk 1, partition 3
/dev/hd9	3, 9	- Hard disk 1, partition 4
/dev/console	4, 0	- Terminal 0 (main keyboard and screen)
/dev/tty0	4, 0	- Same as */dev/console*
/dev/tty1	4, 1	- RS232-C port 1
/dev/tty2	4, 2	- RS232-C port 2
/dev/tty	5, 0	- Current terminal
/dev/lp	6, 0	- Line printer (Centronics port)
/dev/net0	7, 0	- Ethernet

(The IBM diskette combinations are given in Chap. 2.) When */dev/ram* is opened and read, for example, by the command

```
od –x /dev/ram
```

the contents of the RAM disk are read out, byte by byte, starting at byte 0. Similarly, reading */dev/mem* reads out absolute memory, starting at address 0 (the interrupt vectors). The file */dev/kmem* is similar to */dev/mem*, except that it starts at the address in memory where the kernel is located. The next file, */dev/null*, is the null device. It is used as a place for redirecting program output that is not needed. Data copied to */dev/null* are lost forever. The final file in this group, */dev/port*, is used to access I/O ports in protected mode on the 80286 and 80386 CPUs.

The next group of files are for the diskette drives, with different names provided for different sizes (see Chap. 2).

Next come the special files for the hard disks. The first one refers to the entire device, with regard for the partition structure on it. It is occasionally used for

reading the boot block, or for copying one raw hard disk to another. The other entries refer to specific partitions. They are used in commands such as *df* to examine the amount of available space on a partition.

Groups 4 and 5 are for the terminals. The */dev/ttyX* entries are used to access a specific device, such as a modem or serial printer. In contrast, */dev/tty* refers to the current terminal, whatever its number may be.

The character special file */dev/lp* is for the line printer. It is write only. Bytes written to this file are sent to the line printer without modification (to make it possible to send escape sequences to graphics printers). Users normally print files by using the *lpr* program, rather than copying files directly to */dev/lp*. The latter method takes care of converting line feed to carriage return plus line feed, expanding tabs to spaces, etc., whereas the former method does not. Finally, */dev/net* is for networking.

To prevent problems, it is recommended that you remove entries in */dev* that correspond to nonexistent devices. For example, if you have only 1 diskette drive, you should remove */dev/fd1*, etc to eliminate the possibility that you inadvertently use one of them and thus hang the system (which will patiently wait until you insert a diskette in drive 1). If you have only 360K drives, you can remove */dev/atX*, but if you have 1.2M drives, you should *not* remove */dev/fdX* since they are needed when using 360K diskettes in your 1.2M drive.

Another important directory is */etc*. This directory contains files and programs used for mounting and unmounting file systems, the system profiles, the termcap data base, and general system files. For users who have */etc* on the RAM disk, */usr/etc* can be used to maintain a copy on the */usr* partition.

The directory */lib* holds libraries, such as *libc.a*, passes of the C compiler that are not normally directly called by users, and certain miscellaneous files related to compiling. As with *bin*, the full set of programs is kept in */usr/lib*, and the most important ones copied into */lib*. Please note that the *cc* program, which calls the compiler passes, has built-in path names using */usr/lib*. If you want to install parts of the compiler in */lib*, you will have to edit *cc* and recompile it.

The */tmp* and */usr/tmp* directories are used by many programs for temporary files. By putting */tmp* on the RAM disk, these programs are speeded up.

The directories */user* and */usr* are empty. They should be used for mounting file systems. Frequently, */usr* will be partition 1 or 2 of the hard disk, and will contain all the directories listed above, including all the sources.

6.3.1. Mounted File Systems

When MINIX is first started up, the only device present is the root device (default: RAM disk). After the files and directories that belong on the root device are copied there from the root file system diskette, MINIX prints a message asking the user to remove the diskette. It then executes the shell script */etc/rc* as the final step in bringing up the system.

The file */etc/rc* first prints a message asking the user to put the */usr* diskette in drive 0. Then it pauses to allow the diskette to be inserted and the date entered. The shell script now executes the command:

```
/etc/mount /dev/fd0 /usr
```

to mount the system disk on */usr*. From this point on, all the files in */usr*, including the binary programs in */usr/bin*, are available.

On a PC with two diskette drives and no hard disk, you should insert a file system diskette in drive 1 and type:

```
/etc/mount /dev/fd1 /user
```

If you want to mount the same diskette in drive 1 whenever the system is brought up can modify */etc/rc* to perform the mount on drive 1 analogously to the mount on drive 0. Alternatively, a hard disk partition can be mounted. Note, however, that changes made to */etc/rc* on the RAM disk will be lost when the system is next booted unless they are also made to the root file system diskette, which can be mounted and modified, just like any other diskette.

If it is desired to remove the diskette in drive 1 during operation, first type the command:

```
/etc/umount /dev/fd1
```

and wait for the prompt. (Note that the program is called *umount*, just as it is in UNIX, not *unmount*.) There is no *n* in *umount*. You cannot unmount a device holding the working or root directory of any process, or which is otherwise in use.

If you remove a diskette while it is still mounted, the system may hang, but it can be brought back to life by simply re-inserting the same diskette. If you remove a diskette while it is still mounted and insert another in its place, the contents of both file systems will be seriously damaged and information may be irretrievably lost (see below about repairing damaged file systems). Experienced MS-DOS users who are used to constantly switching diskettes without telling the operating system should post discrete KEEP OFF signs on their drives as a reminder.

Although it is permitted to *insert* a non-MINIX diskette in a drive (e.g., to read an MS-DOS diskette), only MINIX file system diskettes can be *mounted*. Attempts to mount a diskette not containing a MINIX file system will be detected and rejected.

6.4. HELPFUL HINTS

In this section we will point out several aspects of MINIX that will frequently be useful. Most of these relate to areas in which MINIX is different from UNIX, so even experienced UNIX users should read it carefully.

6.4.1. Making Backups

As a starter, it is wise to back up your files periodically. To make a backup, first format a diskette as you usually do. If you want to back up a diskette and you have two diskette drives, unmount the file systems in drives 0 and 1. It is possible to back up a mounted file system, but only if no background processes are running. To be doubly safe, give a *sync* command. Insert the newly formatted diskette in drive 1, and then type:

```
cp /dev/fd0 /dev/fd1
```

to copy information from drive 0 to drive 1, assuming you want to copy 360K diskettes. For 1.2M diskettes on the IBM PC, use */dev/at0* and */dev/at1*. When the drive lights go out, the diskettes can be removed.

If you have one diskette drive and a hard disk, to back up a diskette, insert the diskette to be backed up, and copy it to the hard disk. Then insert the new diskette and copy the image back. The following three commands will do the job:

```
cp /dev/fd0 /usr/tmp/image
cp /usr/tmp/image /dev/fd0
rm /usr/tmp/image
```

This command sequence presumes that enough free space exists in */usr/tmp*.

To back up a hard disk, it is best to do it directory by directory. Format enough blank diskettes, and put empty file systems on them using *mkfs*. Mount one of these diskettes. Then use the *backup* program. For example, one might use the sequence:

```
mkfs /dev/fd0 360
/etc/mount /dev/fd0 /user
backup –jmvz /usr/ast /user/ast
/etc/umount /dev/fd0
```

The *backup* program has a variety of useful flags. The **–j** flag suppresses the copying of useless junk, like old core images. The **–m** flag is used to backup large directories over multiple diskettes. The **–v** flag enables verbose mode. In this mode the names of the files are printed as they are backed up. Finally, the **–z** flag arranges for *compress* to be called to compress the files as they are backed up. While compression slows up *backup* considerably, it also doubles the effective capacity of each diskette. Note that *backup* also backs up all the subdirectories in the directory it is working on (i.e., it is recursive).

Suppose a directory is backed up onto a diskette Monday evening. On Tuesday, a number of files are changed in that directory. If the backup diskette from Monday is mounted (instead of a blank diskette) and *backup* called, only those files that have changed since the previous backup will be copied. Be sure to use the same flags (i.e., do not mixed compressed and uncompressed).

6.4.2. Printing

Files can be printed using the *lpr* program. It can be given an explicit list of files, as in

```
lpr file1 file2 file3 &
```

If no arguments are supplied, *lpr* prints its standard input, for example:

```
pr file1 file2 file3 | lpr &
```

Note that *lpr* is not a spooling daemon. It sits in a loop copying files to */dev/lp*. For this reason, it should be started off in the background with the ampersand, so the user can continue working while printing is going on. Only one *lpr* at a time may be running.

6.4.3. Checking on Disk Space

Disk space always seems to be in short supply, no matter how big the disks are. To find out how much space and how many i-nodes are left on diskette 0, type:

```
df /dev/fd0
```

Similar commands can be used for other devices, including */dev/ram* and the hard disk partitions. When *df* is called with no arguments, it checks */etc/mtab* and prints the statistics for the root device and all mounted file systems.

6.4.4. Profiles

When you log in, the shell checks to see if there is a file *.profile* in your home directory. If it finds one, it executes the file as a shell script. This file is commonly used to set shell variables, *stty* parameters, and so on. See */usr/ast/.profile* as a simple example. The system profile, */etc/rc* is executed when MINIX is booted.

6.4.5. Stack Size

The IBM PC does not have any protection hardware. Neither do the Atari, Amiga, or Macintosh. As a result, if a program's stack overruns the area available for it, it will overwrite the data segment. This usually results in a system crash. When a program crashes unexpectedly or acts strange, it is probably worthwhile to find out how much memory is allocated for it (see the "memory" column in the output of *size*). In many cases, increasing the stack space with *chmem* will make it work again. On the IBM PC, the largest executable program has 64K instruction space and 64K data space; the 68000 versions have no limit. To get separate instruction and data spaces, the **-i** flag should be used when compiling programs. When working with unreliable programs, doing *sync*s frequently is advisable.

The problems with memory allocation are due to a large chunk of memory being taken up by the operating system, its buffers, and the RAM disk, plus the fact that multiple programs can be running at once. This, plus the lack of hardware protection, requires that a more economical approach be taken to memory use than the standard MS-DOS method of just giving each program the whole machine to itself. In practice, once the sizes have been set right for a given configuration, they need not be fiddled with any more.

It sometimes happens that a program (or a compiler pass) cannot be executed due to lack of memory for it. When this happens, the shell may a message of the form *program: cannot execute*. The solution is to run fewer programs at once, or reduce the program's size with *chmem*. The amount of stack space assigned to the shell, *make*, etc. in the standard distribution may not be optimal for all applications. Change it if problems arise. To see how much is currently assigned, type

```
size /usr/bin/* | mined
```

In general, if a program goes berserk or the compiler gives nonsensical error messages, the first thing to suspect is stack overrun, which can be tackled with *chmem*.

6.4.6. Compilation Problems

Space is often tight, especially when the amount of program memory is only 512K. It can happen that the C compiler fails due to lack of space, in which case the **-F** flag should be used.

Although an individual compilation can get into space problems, far more likely is that *make* will be unable to run the compiler. The problem is that in addition to the login shell and make itself, several other programs may be running simultaneously, including other shells started by *make*. If problems arise, several approaches can be taken. One is to run:

```
make -n >s
sh s
```

to find out what *make* wants to do, put it on a shell script, and then execute it without *make*. Often this helps.

Another method is to fiddle with the stack sizes of *make*, *sh*, and the compiler passes, *cpp*, *cem*, *opt*, *cg*, and *asld* (some of which can be found in */usr/lib*). By reducing the stack allocated to some of these programs using *chmem* it is frequently possible to solve the problem. Of course if they are given too little stack, they may go berserk. Thus fine tuning the sizes requires some patience.

One last note in this regard, sometimes it is necessary to do something as *root*. There are two ways to become *root*: to log in as *root* and to use the *su* program. They are not quite identical. When using *su* an additional shell is created, taking up memory. If space problems occur after having become *root* using *su*, it is best to hit CTRL-D twice to log out, then log in as *root* directly.

6.4.7. Temporary Files

Several of the utility programs, including the C compiler, create their temporary files in */tmp*, on the RAM disk. If the RAM disk fills up, a message will be printed on the terminal. The first thing to do is check */tmp* to see if there is any debris left over from previous commands, and if so, remove it. If that does not solve the problem, temporarily removing some of the larger files from */bin* or */lib* will usually be enough. These files can be restored later by mounting the root file system on any drive and copying the needed files from it. In a pinch, you can mount a diskette on */tmp* to provide more space for a command that needs a lot of it. When *cc* fills up */tmp*, the **–T** flag can be used to put the temporary files in another directory.

6.4.8. Aborting Commands

MINIX, like UNIX, will not break off a system call part way through just because the DEL key has been struck. When the system call in question happens to be an EXEC, which is loading a long program from a slow diskette, it can take a few seconds before the shell prompt appears. Be patient. Hitting DEL again makes things worse, rather than better.

6.4.9. System Status Reporting

Although it is really intended as a debugging aid, rather than a permanent part of the system, on the IBM PC version the F1 and F2 function keys cause dumps of some of the internal tables to be printed on the screen. (For the 68000s, other keys are used, as described later.) F1 gives a dump like *ps*, but instantly. Frequently, the system appears to be stopped, but it is actually thinking its little head off and using the RAM disk, which, unlike the other disks, is not accompanied by whirring and clicking noises and flashing lights. The nervous user can press F1 to see the internal process table to verify that progress is still being made. The F1 and F2 keys are intercepted directly by the keyboard driver, so they always work, no matter what the computer is doing. The values in the columns *user* and *sys* are the number of clock ticks charged to each process. By hitting F1 twice, a few seconds apart, it is possible to see where the CPU time is going.

6.4.10. Escape Sequences

MINIX supports ANSI escape sequences as well as Berkeley termcap entries. The latter can be found in the file */etc/termcap*. The entries use the ANSI escape sequences. The *TERM* variable should be set to *minix* to use these entries. A library routine, *termcap.c* is provided to manipulate them.

6.4.11. Serial Lines

Communication with the outside world over a modem is possible. The number of RS232 ports supported in controlled by the constant *NR_RS_LINES* defined in *kernel/tty.h*. This constant should be set to the proper number of ports for your configuration, since each port requires about 1K of table space in the kernel. To log into other systems or transfer files, see the manual pages for *kermit*, *rz*, *sz*, and *term*. On the Atari, *stterm* is also available.

6.4.12. Transferring Files to and from Other Operating systems

It is possible to copy files from an MS-DOS disk to MINIX or vice versa. See the description of *dosread* and *doswrite* for details. Similarly, see *tosread* and *toswrite* for the Atari, *macread* and *macwrite* for the Macintosh, and *transfer* for the Amiga.

6.4.13. Keyboard Mapping

The ASCII codes produced by the IBM PC keyboard are determined by software, not hardware. A mapping has been chosen to try to produce a unique value for each key, so programs can see the difference between, for example, the + in the top row and the + in the numeric keypad. Since the keyboards of the various machines differ, the mappings are not identical. To see which code or codes a given key produces, use *od –b*, and then type the key or keys followed by a carriage return and a CTRL-D.

6.5. SYSTEM ADMINISTRATION

Since MINIX is in principle a multiuser timesharing system, not unlike what large computer centers run, you will have to learn how to administer your system. Fortunately, doing this is not difficult. System administration tasks have to be done by the superuser. Superusers have more power than ordinary users. They can violate nearly all of the system's protection rules. Although there is no Hippocratic Oath for superusers (yet), tradition requires them to exercise their great power with care and responsibility. Superusers get a special prompt (#), to remind them of their awesome power.

To become superuser, login as *root* using the password *Geheim*. (Notice the capital *G*). Alternatively, use the *su* program with *Geheim* as password. Please take note that these two methods of becoming superuser are not quite the same. Using *su* causes an extra shell to be created. If you are short on memory, and intend to do something complicated as superuser (such as running a large *make* job), you may have to log out and log in again as *root*.

6.5.1. Making New File Systems

One of the things that superusers do is make new file systems. This is possible using the program *mkfs* (make file system). To make an empty 360 block file system on diskette 0, type:

```
mkfs /dev/fd0 360
```

When the program finishes, the file system will be ready to mount. On a system with only one diskette drive and no hard disk, *mkfs* will first have to be copied to */bin*, (on the RAM disk), the */dev/fd0* file system unmounted, a blank diskette inserted into drive 0 and then the file system made.

It is also possible to make a file system that is initialized with files and directories. A command for doing this is:

```
mkfs /dev/fd0 proto
```

where *proto* is a prototype file. The manual entry for *mkfs* (in Chap. 8) gives an example of a prototype file.

6.5.2. File System Checking

File systems can be damaged by system crashes, by accidently removing a mounted file system, by forgetting to run *sync* before shutting the system down and in other ways. Repairing a file system by hand is a tricky business, so a program, called *fsck*, has been provided to automate the job. It is best to first copy *fsck*, to the root file system and then unmount the file system to be repaired, unless it is the root file system. If the root file system is on a hard disk partition, it is best to reboot MINIX and run *fsck* from a diskette so that the root file system is unmounted while *fsck* is modifying it.

The simplest way to repair a file system is to run *fsck* in automatic mode. To repair */dev/hd1*, for example, just type:

```
cd /
cp /usr/bin/fsck /fsck
/etc/umount /dev/hd1
fsck –a /dev/hd1
/etc/mount /dev/hd1 /usr
```

Fsck will run, ask some questions, answer its own questions, and fix everything. When it is done, you can remount the repaired file system and continue. Other options are described in the manual page for *fsck*.

6.5.3. The /etc Directory

The */etc* directory contains several files that superusers should know about. They are:

Name	**- Description**
gettydefs	- Used for configuring dial in lines using modems
group	- Contains names of the user groups
message	- Message of the day
passwd	- Password file
rc	- Shell script executed after the system is booted
setup_move	- Additional hard disk setup (remove after installation)
setup_root	- Used to set up the hard disk RAM image (remove after installation)
setup_usr	- Used to set up the hard disk partition for */usr* (remove after installation)
termcap	- Berkeley termcap entries for MINIX
ttys	- Enables/disables RS-232 ports for use as terminals
ttytype	- Terminal configuration

Probably the most important of these is the password file, */etc/passwd*. You can enter new users by editing this file and adding a line for each new user. The entry for a user named *fozzie* might be:

```
fozzie::15:1:Fozzie the Bear:/usr/fozzie:/bin/sh
```

The entry contains seven fields, separated by colons. These fields contain the login name, password (initially null), uid, gid, name, home directory, and shell for the new user. When a new user is entered, the corresponding home directory must also be created, using *mkdir*, and its owner set correctly, using *chown* and *chgrp*. Each user must have a unique uid, but the numerical values are unimportant. It is probably adequate to put all ordinary users in group 3, unless there really are distinct groups of users. When the new user logs in for the first time, he should choose a password and enter it using *passwd*.

Another important file is */etc/rc*. Each time the system is booted, this file is run as a shell script just before the

```
login:
```

message is printed. It can be used to mount file systems, request the date, erase temporary files, and anything else that needs to be done before starting the system. It also forks off *update*, which runs in the background and issues a SYNC system call every 30 seconds to flush the buffer cache.

If you do not have a hard disk and want to use two diskette drives during normal operations, it may be convenient to modify */etc/rc* to mount */dev/fd1* on */user* during system boot. If you do this, you can also change */etc/passwd* to put your home directory on */user* instead of */usr*. Of course you can also change */etc/rc* to mount a hard disk partition.

The file */etc/ttys* contains one line for each terminal in the system. During startup, *init* reads this file and forks off a login process for each terminal. When the console is the only terminal, *ttys* contains only 1 line.

Also contained in */etc* are the programs *mount*, and *umount* for mounting and unmounting file systems, respectively.

When any of the files on the RAM disk, such as */etc/passwd*, are modified, the changes will be lost when the system is shut down unless the modified files are explicitly copied back to the root file system. This can be done by mounting the root file system diskette and then copying the files with *cp*.

6.5.4. Miscellaneous Notes

A few MINIX programs can only be executed by the superuser. Some of these, such as *df*, are owned by the root and have the SETUID bit on, so that when they are executed, the effective uid is that of the superuser, even though the real uid is not.

In general, if a program, *prog*, needs to run as the superuser but is to be made generally available to all users, it can be made into a SETUID program owned by the root by the command lines:

```
chown root prog
chmod 4755 prog
```

Needless to say, only the superuser can execute these commands. The shell script *fixbin* can be run by the superuser to set all the permissions (and sizes) as follows:

```
fixbin /bin /bin
```

If the two arguments are different, the executable files will be first copied from the first directory to the second one.