A Basic UNIX Tutorial

This tutorial comprises fourteen sections, each of which addresses a fundamental aspect of UNIX computing. It concentrates on illustrating the central concepts by providing short explanations, along with examples, and exercises.

This tutorial covers the "Introduction to UNIX" and "Intermediate UNIX" workshops offered by the Idaho State University Computer Center. Check the ISU workshop schedule to see when the workshops are offered.

Table of Contents

Processes and Multitasking

Managing processes.

Interaction and Job Control

More on managing processes.

Text Editing with Emacs

Creating and editing text files with the emacs editor.

The Execution Environment

The environment under which shell commands and programs run.

Customizing the Shell

Personalizing your UNIX shell environment.

Interactive Use of the Shell

Tips and tricks to enhance your efficiency with the command line interface.

The UNIX Filesystem

A closer look at UNIX files and directories.

---

upper level ISU Home

IDAHO STATE UNIVERSITY
Contact: webmaster@isu.edu
Revised: February 5, 1997
URL: http://www.isu.edu/departments/comcom/workshops/unix/
Author: Jonathan Byrd

Section 1: What Is Unix?

Unix is an *operating system*. The job of an operating system is to orchestrate the various parts of the computer -- the processor, the on-board memory, the disk drives, keyboards, video monitors, etc. -- to perform useful tasks. The operating system is the master controller of the computer, the glue that holds together all the components of the system, including the administrators, programmers, and users. When you want the computer to do something for you, like start a program, copy a file, or display the contents of a directory, it is the operating system that must perform those tasks for you.

More than anything else, the operating system gives the computer its recognizable characteristics. It would be difficult to distinguish between two completely different computers, if they were running the same operating system. Conversely, two identical computers, running different operating systems, would appear completely different to the user.

Unix was created in the late 1960s, in an effort to provide a multiuser, multitasking system for use by programmers. The philosophy behind the design of Unix was to provide simple, yet powerful utilities that could be pieced together in a flexible manner to perform a wide variety of tasks.

The Unix operating system comprises three parts: The kernel, the standard utility programs, and the system configuration files.

The kernel

The kernel is the core of the Unix operating system. Basically, the kernel is a large program that is loaded into memory when the machine is turned on, and it controls the allocation of hardware resources from that point forward. The kernel knows what hardware resources are available (like the processor(s), the on-board memory, the disk drives, network interfaces, etc.), and it has the necessary programs to talk to all the devices connected to it.

The standard utility programs

These programs include simple utilities like cp, which copies files, and complex utilities, like the shell that allows you to issue commands to the operating system.

The system configuration files

The system configuration files are read by the kernel, and some of the standard utilities. The Unix kernel and the utilities are flexible programs, and certain aspects of their behavior can be controlled by changing the standard configuration files. One example of a system configuration file is the filesystem table "fstab" , which tells the kernel where to find all the files on the disk drives. Another example is the system log configuration file "syslog.conf", which tells the kernel how to record the various kinds of events and errors it may encounter.

Section 2: Accessing a Unix System

There are many ways that you can access a Unix system. If you want the fullest possible access to the computer's commands and utilities, you must initiate a login session. The main mode of initiating a login session to a Unix machine is through a *terminal*, which usually includes a keyboard, and a video monitor.

When a terminal establishes a connection to the Unix system, the Unix kernel runs a process called a *tty* to accept input from the terminal, and send output to the terminal. When the tty process is created, it must be told the capabilities of the terminal, so it can correctly read from, and write to, the terminal. If the tty process receives incorrect information about the terminal type, unexpected results can occur.

Console

Every Unix system has a main console that is connected directly to the machine. The console is a special type of terminal that is recognized when the system is started. Some Unix system operations must be performed at the console. Typically, the console is only accessible by the system operators, and administrators.

Dumb terminals

Some terminals are referred to as "dumb" terminals because they have only the minimum amount of power required to send characters as input to the Unix system, and receive characters as output from the Unix system.

Personal computers are often used to emulate dumb terminals, so that they can be connected to a Unix system.

Dumb terminals can be connected directly to a Unix machine, or may be connected remotely, through a modem, a terminal server, or other network connection.

Smart terminals

Smart terminals, like the X terminal, can interact with the Unix system at a higher level. Smart terminals have enough on-board memory and processing power to support graphical interfaces. The interaction between a smart terminal and a Unix system can go beyond simple characters to include icons, windows, menus, and mouse actions.

Network-based access modes

Unix computers were designed early in their history to be network-aware. The fact that Unix computers were prevalent in academic and research environments led to their broad use in the

implementation of the Department of Defense's Advanced Research Projects Administration (DARPA) computer network. The DARPA network laid the foundations for the Internet.

FTP

The FTP (File Transfer Protocol) provides a simple means of transferring files to and from a Unix computer. FTP access to a Unix machine may be authenticated by means of a username and password pair, or may be anonymous. An FTP session provides the user with a limited set of commands with which to manipulate and transfer files.

Telnet

Telnet is a means by which one can initiate a Unix shell login across the Internet. The normal login procedure takes place when the telnet session is initiated.

HTTP

The HTTP protocol has become important in recent years, because it is the primary way in which the documents that constitute the World Wide Web are served. HTTP servers are most often publicly accessible. In some cases, access to documents provided by HTTP servers will require some form of authentication.

HTTPS

A variation of HTTP that is likely to become increasingly important in the future. The "S" stands for "secure." When communications are initiated via the HTTPS protocol, the sender and recipient use an encryption scheme for the information to be exchanged. When the sending computer transmits the message, the information is encrypted so that outside parties cannot examine it. Once the message is received by the destination machine, decryption restores the original information.

---

prev page next page upper level

Section 3: Logging In and Logging Out

To ensure security and organization on a system with many users, Unix machines employ a system of user accounts. The user accounting features of Unix provide a basis for analysis and control of system resources, preventing any user from taking up more than his or her share, and preventing unauthorized people from accessing the system. Every user of a Unix system must get permission by some access control mechanism.

Logging in

Logging in to a Unix system requires two pieces of information: A username, and a password. When you sit down for a Unix session, you are given a login prompt that looks like this:

```
login:
```

Type your username at the login prompt, and press the return key. The system will then ask you for your password. When you type your password, the screen will not display what you type.

Your username

Your username is assigned by the person who creates your account. At ISU, the standard username is the first four letters of your last name concatenated with the first four letters of your first name.

Your username must be unique on the system where your account exists since it is the means by which you are identified on the system.

Your password

When your account is created, a password is assigned. The first thing you should do is change your password, using the passwd utility. To change your password, type the command

```
passwd
```

after you have logged in. The system will ask for your old password, to prevent someone else from sneaking up, and changing your password. Then it will ask for your new password. You will be asked to confirm your new password, to make sure that you didn't mistype. It is very important that you choose a good password, so that someone else cannot guess it. Here are some rules for selecting a good password:

- Do not use any part of your name, your spouse's name, your child's name, your pet's name, or anybody's name. Do not use any backward spellings of any name, either.
- Do not use an easily-guessable number, like your phone number, your social security number, your address, license plate number, etc.
- Do not use any word that can be found in an English or foreign-language dictionary.
- Do not use all the same letter, or a simple sequence of keys on the keyboard, like qwerty.

- **Do** use a mix of upper-case and lower-case letters, numbers, and control characters.
- **Do** use at least six characters.

If you have accounts on multiple machines, use a different password on each machine. Do not choose a password that is so difficult to remember that you must write it down.

Logging Out

When you're ready to quit, type the command

```
exit
```

Before you leave your terminal, make sure that you see the login prompt, indicating that you have successfully logged out. If you have left any unresolved processes, the Unix system will require you to resolve them before it will let you log out. Some shells will recognize other commands to log you out, like "logout" or even "bye".

It is always a good idea to clear the display before you log out, so that the next user doesn't get a screenful of information about you, your work, or your user account. You can type the command

```
clear
```

right before you log out, or you can press the return key until all the information is scrolled off the screen.

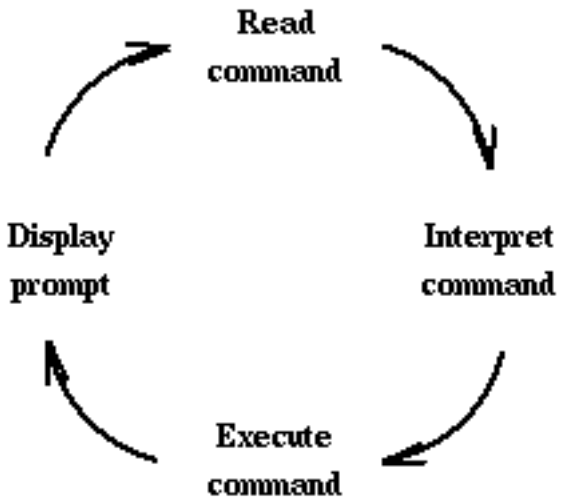
---

prev page | next page | upper level

Section 4: The Unix Shell

The shell is perhaps the most important program on the Unix system, from the end-user's standpoint. The shell is your interface with the Unix system, the middleman between you and the kernel.

CONCEPT: The shell is a type of program called an *interpreter*. An interpreter operates in a simple loop: It accepts a command, interprets the command, executes the command, and then waits for another command. The shell displays a "prompt," to notify you that it is ready to accept your command.



Interpreter loop

The shell recognizes a limited set of commands, and you must give commands to the shell in a way that it understands: Each shell command consists of a command name, followed by command options (if any are desired) and command arguments (if any are desired). The command name, options, and arguments, are separated by blank space.

CONCEPT: The shell is a program that the Unix kernel runs for you. A program is referred to as a *process* while the kernel is running it. The kernel can run the same shell program (or any other program) simultaneously for many users on a Unix system, and each running copy of the program is a separate process.

Many basic shell commands are actually subroutines built in to the shell program. The commands that are not built in to the shell require the kernel to start another process to run them.

CONCEPT: When you execute a non built-in shell command, the shell asks the kernel to create a new subprocess (called a "child" process) to perform the command. The child process exists just long enough to execute the command. The shell waits until the child process finishes before it will

accept the next command.

EXERCISE: Explain why the exit (logout) procedure must be built in to the shell.

EXPLANATION: If the logout procedure were not built in to the shell, the kernel would start a new child process to run it. The new process would logout, and then return you to the original shell. You would thus find yourself back where you started, without having logged out.

Unlike DOS, the Unix shell is case-sensitive, meaning that an uppercase letter is not equivalent to the same lower case letter (i.e., "A" is not equal to "a"). Most all Unix commands are lower case.

Entering shell commands

The basic form of a Unix command is: **commandname [-options] [arguments]**

The command name is the name of the program you want the shell to execute. The command options, usually indicated by a dash, allow you to alter the behavior of the command. The arguments are the names of files, directories, or programs that the command needs to access.

The square brackets ([and]) signify optional parts of the command that may be omitted.

EXAMPLE: Type the command

```
ls -l /tmp
```

to get a long listing of the contents of the /tmp directory. In this example, "ls" is the command name, "-l" is an option that tells ls to create a long, detailed output, and "/tmp" is an argument naming the directory that ls is to list.

Aborting a shell command

Most Unix systems will allow you to abort the current command by typing Control-C. To issue a Control-C abort, hold the control key down, and press the "c" key.

Special characters in Unix

Unix recognizes certain special characters, called "meta characters," as command directives. The shell meta characters are recognized anywhere they appear in the command line, even if they are not surrounded by blank space. For that reason, it is safest to only use the characters A-Z, a-z, 0-9, and the period, dash, and underscore characters when naming files and directories on Unix. If your file or directory has a shell meta character in the name, you will find it difficult to use the name in a shell command.

The shell meta characters include:

```
\ / < > ! $ % ^ & * | { } [ ] " ' ` ~ ;
```

Different shells may differ in the meta characters recognized.

The meaning of some of the meta characters, and ways to use them, will be introduced as the tutorial progresses.

Getting help on Unix

To access the on-line manuals, use the **man** command, followed by the name of the command you need help with.

EXAMPLE: Type

```
man ls
```

to see the manual page for the "ls" command.

EXAMPLE: To get help on using the manual, type

```
man man
```

to the Unix shell.

---

prev page next page upper level

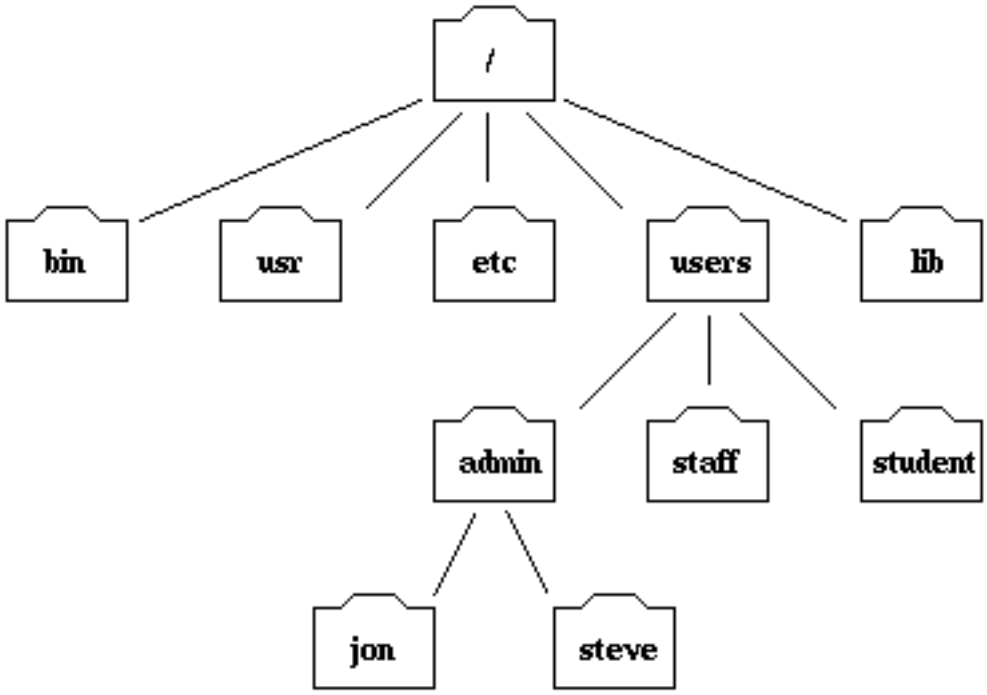
Section 5: Working with Files and Directories

Here is an index to the topics in this section:

The Unix filesystem structure

All the stored information on a Unix computer is kept in a *filesystem*. Any time you initiate a shell login session, the shell considers you to be located somewhere within a filesystem. Although it may seem strange to be "located" somewhere in a computer's filesystem, the concept is not so different from real life. After all, you can't just *be*, you have to be *somewhere*. The place in the filesystem tree where you are located is called the *current working directory*.

CONCEPT: The Unix filesystem is hierarchical (resembling a tree structure). The tree is anchored at a place called the root, designated by a slash "/". Every item in the Unix filesystem tree is either a file, or a directory. A directory is like a container. A directory can contain files, and other directories. A directory contained within another is called the *child* of the other. A directory in the filesystem tree may have many children, but it can only have one parent. A file can hold information, but cannot contain other files, or directories.



Part of the filesystem tree

CONCEPT: To describe a specific file or directory in the filesystem hierarchy, you must specify a "path." The path to a location can be defined as an *absolute path*, starting from the root anchor point, or as a *relative path*, starting from the current location. When specifying a path, you simply trace a route through the filesystem tree, listing the sequence of directories you pass through as you go from one point to another. Each directory listed in the sequence is separated by a slash.

It is initially confusing to some that Unix uses the slash character "/" to denote the filesystem root directory, and as a directory separator in paths. Just remember, when the slash is the first thing in the path to the file or directory, the path begins at the root directory. Otherwise, the slash is a separator.

Unix provides the shorthand notation of "." to refer to the current location, and ".." to refer to the parent directory. Because Unix filesystem trees have no loops, the ".." notation refers unambiguously to the directory's parent.

EXERCISE: Specify the absolute path to the directory named "jon" at the bottom of the tree diagram.

EXPLANATION: Since the absolute path must always begin at the root (/) directory, the path would be:

```
/users/admin/jon
```

EXERCISE: Specify the relative path from the directory named "student" to the directory named "jon" in the tree diagram.

EXPLANATION: Starting from the student directory, we would first have to move up the filesystem tree (using the ".." notation) to the directory called "users" before we could descend to the directory called "jon". The path would be:

```
../admin/jon
```

File and directory permissions

CONCEPT: Unix supports access control. Every file and directory has associated with it ownership, and access permissions. Furthermore, one is able to specify those to whom the permissions apply.

Permissions are defined as read, write, and execute. The read, write, and execute permissions are referred to as r, w, and x, respectively.

Those to whom the permissions apply are the user who owns the file, those who are in the same group as the owner, and all others. The user, group, and other permissions are referred to as u, g, and o, respectively.

A short note on groups: Unix allows users to be placed in groups, so that the control of access is made simpler for administrators.

The meaning of file and directory permissions

Read permission

: For a file, read permission allows you to view the contents of the file. For a directory, read permission allows you to list the directory's contents.

Write permission

: For a file, write permission allows you to modify the contents of the file. For a directory, write permission, along with execute permission, allows you to alter the contents of the directory, i.e., to add and delete files and subdirectories.

Execute permission

: For a file, execute permission allows you to run the file, if it is an executable program, or script. Note that file execute permission is irrelevant for non executable files. For a directory, execute permission allows you to refer to the contents of the directory. Without execute permission on a directory, read and write permissions within that directory are limited.

Viewing permissions

To see the permissions on a file, use the ls command, with the -l option.

EXAMPLE: Execute the command

```
ls -l /etc/passwd
```

to view the information on the system password database. The output should look similar to this:

```
-rw-r--r-- 1 root sys 41002 Apr 17 12:05 /etc/passwd
```

The first 10 characters describe the access permissions. The first dash indicates the type of file (d for directory, s for special file, - for a regular file). The next three characters ("rw-") describe the permissions of the owner of the file: read and write, but no execute. The next three characters ("r--") describe the permissions for those in the same group as the owner: read, no write, no execute. The next three characters describe the permissions for all others: read, no write, no execute.

Setting permissions

Unix allows you to set the permissions on files that you own. The command to change the file permission mode is chmod. Chmod requires you to specify the new permissions you want, and specify the file or directory you want the changes applied to.

To set file permissions, you may use to the "rwx" notation to specify the type of permissions, and the "ugo" notation to specify those the permissions apply to.

To define the kind of change you want to make to the permissions, use the plus sign (+) to add a permission, the minus sign (-) to remove a permission, and the equal sign (=) to set a permission directly.

EXAMPLE: Type the command

```
chmod g=rw- ~/.shrc
```

to change the file permissions on the file .shrc, in your home directory. Specifically, you are specifying group read access and write access, with no execute access.

EXERCISE: Change the permissions on the .shrc file in your home directory so that group and others have read permission only.

EXPLANATION: Typing the command

```
chmod go=r-- ~/.shrc
```

would accomplish the task.

You can also combine multiple operation codes in a single statement. Separate each operation code with a comma, and do not have any embedded space anywhere in the operation code sequence. For example:

```
chmod u=rwx,go=r-x foo
```

Would set owner permissions to rwx, with group and other permissions set to r-x.

Changing Directories

In Unix, your location in the filesystem hierarchy is known as your "current working directory." When you log in, you are automatically placed in your "home directory." To see where you are, type the command

```
pwd
```

which stands for "print working directory."

To change your location in the filesystem hierarchy, use the cd (change directory) command, followed by an argument defining where you want to go. The argument can be either an absolute path to the destination, or a relative path.

EXAMPLE: Type the command

```
cd /tmp
```

to go to the /tmp directory. You can type

```
pwd
```

to confirm that you're actually there.

If you type the cd command without an argument, the shell will place you in your home directory.

EXERCISE: Type the command

```
pwd
```

and note the result. Then type

```
cd ..
```

to the shell. Type

```
pwd
```

again to see where you ended up.

EXPLANATION: The "cd .." command should have moved you up one level in the directory tree, because ".." is Unix shorthand for the parent directory. The result of the second "pwd" command should be the same as the first, with the last directory in the path omitted.

Listing the contents of a directory

The ls command allows you to see the contents of a directory, and to view basic information (like size, ownership, and access permissions) about files and directories. The ls command has numerous options, so see the manual page on ls (type `man ls`) for a complete listing. The ls command also accepts one or more arguments. The arguments can be directories, or files.

EXAMPLE: Type the command

```
ls -lR /lib/l*
```

to the Unix shell.

In the example, the "l" and "R" options of the ls command are invoked together. Some commands permit you to group options in that way, and some commands require the options to be named separately, e.g., `ls -l -R`. The l option calls for a long output, and the R option causes ls to operate recursively, moving down directory trees.

The last part of the example, "/lib/l*", directs the ls command to list files and directories in the /lib directory, that begin with the letter l. The wild card character, "*", matches any character(s).

EXERCISE: Type the command

```
ls -m /etc/i*g
```

to the shell. How did the shell respond, and why?

EXPLANATION: The shell responded by printing all the entries in the /etc directory that start with the letter i, and end with the letter g. The -m option causes the output to be streamed into a single line. See the manual page for ls to get a complete description of the ls command's options.

EXERCISE: Find the permissions on your home directory.

EXPLANATION: There are many ways to accomplish this. You could type

```
cd
```

to get to your home directory, and then type

```
ls -la
```

The -a option instructs the ls command to list all files, including those that start with the period character (ls normally ignores files that begin with a period). The directory permissions are listed next to the "." symbol. Remember that "." is Unix shorthand for the current working directory.

Viewing the contents of a file

CONCEPT: Text files are intended for direct viewing, and other files are intended for computer interpretation.

The Unix file command allows you to determine whether an unknown file is in text format, suitable for direct viewing.

EXERCISE: Type the command

```
file /bin/sh
```

to see what kind of file the shell is.

EXPLANATION: The shell is a shared executable, indicating that the file contains binary instructions to be executed by the computer.

The cat command
The cat command concatenates files and sends them to the screen. You can specify one or more files as arguments. Cat makes no attempt to format the text in any way, and long output may scroll off the screen before you can read it.

EXAMPLE: Send the contents of your .profile file to the screen by typing

```
cat ~/.profile
```

to the shell. The tilde character (~) is Unix shorthand for your home directory.

The more command
The more command displays a text file, one screenful at a time. You can scroll forward a line at a time by pressing the return key, or a screenful at a time by pressing the spacebar. You can quit at any time by pressing the q key.

EXAMPLE: Type

```
more /etc/rc
```

to the shell. Scroll down by pressing return, and by pressing the spacebar. Stop the more command from displaying the rest of the file by typing q.

The head and tail commands

The head command allows you to see the top part of a file. You may specify the number of lines you want, or default to ten lines.

EXAMPLE: Type

```
head -15 /etc/rc
```

to see the first fifteen lines of the /etc/rc file.

The tail command works like head, except that it shows the last lines of of file.

EXAMPLE: Type

```
tail /etc/rc
```

to see the last ten lines of the file /etc/rc. Because we did not specify the number of lines as an option, the tail command defaulted to ten lines.

Copying files and directories

The Unix command to copy a file or directory is cp. The basic cp command syntax is **cp source destination**.

EXAMPLE: The command

```
cp ~/.profile ~/pcopy
```

makes a copy of your .profile file, and stores it in a file called "pcopy" in your home directory.

Before creating the new copy, the cp command is clever enough to check to see if a file or directory with the destination name already exists. If a file with that name already exists, the shell will overwrite the old file. If a directory with that name already exists, the shell will place the copy in that directory, and make the name of the new copy the same as the original.

EXERCISE: Describe the permissions necessary to successfully execute the command in the previous example.

EXPLANATION: To copy the .profile file, one must have read permission on the file, and execute permission in the directory where the file resides. To create the new file called pcopy, one must have write and execute permission in the directory where the file will be created.

Moving and renaming files

The Unix mv command moves files and directories. You can move a file to a different location in the filesystem, or change the name by moving the file within the current location.

EXAMPLE: The command

```
mv ~/pcopy ~/qcopy
```

takes the pcopy file you created in the cp exercise, and renames it "qcopy".

Removing files

The rm command is used for removing files and directories. The syntax of the rm command is **rm filename**. You may include many filenames on the command line.

EXAMPLE: Remove the the shrccopy file that you placed in your home directory in the section on moving files by typing

```
rm ~/.shrccopy
```

Creating a directory

The Unix mkdir command is used to make directories. The basic syntax is **mkdir directory-name**. If you do not specify the place where you want the directory created (by giving a path as part of the directory name), the shell assumes that you want the new directory placed within the current working directory.

EXAMPLE: Create a directory called foo within your home directory by typing

```
mkdir ~/foo
```

EXERCISE: Create a directory called bar, within the directory called foo, within your home directory.

EXPLANATION: Once the foo directory is created, you could just type

```
mkdir ~/foo/bar
```

Alternately, you could type

```
cd ~/foo; mkdir bar
```

In the second solution, two Unix commands are given, separated by a semicolon. The first part of the command makes foo the current working directory. The second part of the command creates the bar directory in the current working directory.

Removing a directory

The Unix rmdir command removes a directory from the filesystem tree. The rmdir command does not work unless the directory to be removed is completely empty.

The rm command, used with the -r option can also be used to remove directories. The rm -r command will first remove the contents of the directory, and then remove the directory itself.

EXERCISE: Describe how to remove the "foo" directory you created, using both rmdir, and rm with the -r option.

EXPLANATION: You could enter the commands

```
rmdir ~/foo/bar; rmdir ~/foo
```

to accomplish the task with the rmdir command. Note that you have to rmdir the bar subdirectory before you can rmdir the foo directory. Alternately, you could remove the foo directory with the command

```
rm -r ~/foo
```

Section 6: Redirecting Input and Output

CONCEPT: Every program you run from the shell opens three files: Standard input, standard output, and standard error. The files provide the primary means of communications between the programs, and exist for as long as the process runs.

The standard input file provides a way to send data to a process. As a default, the standard input is read from the terminal keyboard.

The standard output provides a means for the program to output data. As a default, the standard output goes to the terminal display screen.

The standard error is where the program reports any errors encountered during execution. By default, the standard error goes to the terminal display.

CONCEPT: A program can be told where to look for input and where to send output, using *input/output redirection*. Unix uses the "less than" and "greater than" special characters (< and >) to signify input and output redirection, respectively.

Redirecting input

Using the "less-than" sign with a file name like this:

```
< file1
```

in a shell command instructs the shell to read input from a file called "file1" instead of from the keyboard.

EXAMPLE: Use standard input redirection to send the contents of the file /etc/passwd to the more command:

```
more < /etc/passwd
```

Many Unix commands that will accept a file name as a command line argument, will also accept input from standard input if no file is given on the command line.

EXAMPLE: To see the first ten lines of the /etc/passwd file, the command:

```
head /etc/passwd
```

will work just the same as the command:

```
head < /etc/passwd
```

Redirecting output

Using the "greater-than" sign with a file name like this:

```
> file2
```

causes the shell to place the output from the command in a file called "file2" instead of on the screen.

If the file "file2" already exists, the old version will be overwritten.

EXAMPLE: Type the command

```
ls /tmp > ~/ls.out
```

to redirect the output of the ls command into a file called "ls.out" in your home directory. Remember that the tilde (~) is Unix shorthand for your home directory. In this command, the ls command will list the contents of the /tmp directory.

Use two "greater-than" signs to append to an existing file. For example:

```
>> file2
```

causes the shell to append the output from a command to the end of a file called "file2". If the file "file2" does not already exist, it will be created.

EXAMPLE: In this example, I list the contents of the /tmp directory, and put it in a file called myls. Then, I list the contents of the /etc directory, and append it to the file myls:

```
ls /tmp > myls
ls /etc >> myls
```

Redirecting error

Redirecting standard error is a bit trickier, depending on the kind of shell you're using (there's more than one flavor of shell program!). In the POSIX shell and ksh, redirect the standard error with the symbol "2>".

EXAMPLE: Sort the /etc/passwd file, place the results in a file called foo, and trap any errors in a file called err with the command:

```
sort < /etc/passwd > foo 2> err
```

Section 7: Pipelines and Filters

CONCEPT: Unix allows you to connect processes, by letting the standard output of one process feed into the standard input of another process. That mechanism is called a *pipe*.

Connecting simple processes in a pipeline allows you to perform complex tasks without writing complex programs.

EXAMPLE: Using the more command, and a pipe, you can manage the screen presentation of command output. Examine the contents of the /etc directory by typing

```
ls -l /etc | more
```

to the shell.

If you type "q" to exit the more command, where does the remaining output of the ls command go? The answer lies in the way pipelined processes communicate. When the kernel creates a process, each of the process's three file descriptors (standard input, standard output, standard error) is assigned an area, occupying a small block of the computer's memory. The pipeline is established when the contents of the ls command's output buffer is copied into the input buffer of the more command. When the more command is terminated, the Unix operating system will terminate the ls command, as it has nowhere to copy it's output.

EXERCISE: How could you use head and tail in a pipeline to display lines 25 through 75 of a file?

ANSWER: The command

```
cat file | head -75 | tail -50
```

would work. The cat command feeds the file into the pipeline. The head command gets the first 75 lines of the file, and passes them down the pipeline to tail. The tail command then filters out all but the last 50 lines of the input it received from head. It is important to note that in the above example, tail never receives the original file, but only sees the 75 lines that were passed to it by the head command.

It is easy for beginners to confuse the usage of the input/output redirection symbols < and >, with the usage of the pipe. Remember that input/output redirection connects processes with *files*, while the pipe connects processes with *other processes.*

Grep

The grep utility is one of the most useful filters in Unix. Grep searches line-by-line for a specified pattern, and outputs any line that matches the pattern. The basic syntax for the grep command is **grep [-options] pattern [file]**. If the file argument is omitted, grep will read from standard input. It is always best to enclose the pattern within single quotes, to prevent the shell from misinterpreting the command.

The grep utility recognizes a variety of patterns, and the pattern specification syntax was taken from the vi editor. Here are some of the characters you can use to build grep expressions:

- The caret (^) matches the beginning of a line.
- The dollar sign ($) matches the end of a line.
- The period (.) matches any single character.
- The asterisk (*) matches zero or more occurrences of the previous character.
- The expression [*a*-*b*] matches any characters that are lexically between *a* and *b*.

Note that some of the pattern matching characters are also shell meta characters. If you use one of those characters in a grep command, make sure to enclose the pattern in single quote marks, to prevent the shell from trying to interpret them.

EXAMPLE: Type the command

```
grep 'jon' /etc/passwd
```

to search the /etc/passwd file for any lines containing the string "jon".

EXAMPLE: Type the command

```
grep '^jon' /etc/passwd
```

to see the lines in /etc/passwd that begin with the character string "jon".

EXERCISE: List all the files in the /tmp directory owned by the user root.

EXPLANATION: The command

```
ls -l /tmp | grep 'root'
```

would show a long listing of all files in the /tmp directory that contain the word "root". Note that files not owned by the root user may contain the string "root" somewhere in the name, and would appear in the output, but the grep filter can cut the down the number of lines of output you will have to look at.

---

prev page | next page | upper level

Section 8: Process Control and Multitasking

CONCEPT: The Unix kernel can keep track of many processes at once, dividing its time between the jobs submitted to it. Each process submitted to the kernel is given a unique *process ID*.

Single-tasking operating systems, like DOS, or the Macintosh System, can only perform one job at a time. A user of a single-tasking system can switch to different windows, running different applications, but only the application that is currently being used is active. Any other task that has been started is *suspended* until the user switches back to it. A suspended job receives no operating system resources, and stays just as it was when it was suspended. When a suspended job is reactivated, it begins right where it left off, as if nothing had happened.

The Unix operating system will simultaneously perform multiple tasks for a single user. Activating an application does not have to cause other applications to be suspended.

Actually, it only *appears* that Unix is performing the jobs simultaneously. In reality, it is running only one job at a time, but quickly switching between all of its ongoing tasks. The Unix kernel will execute some instructions from job A, and then set job A aside, and execute instructions from job B. The concept of switching between queued jobs is called process scheduling.

Viewing processes

Unix provides a utility called **ps** (process status) for viewing the status of all the unfinished jobs that have been submitted to the kernel. The ps command has a number of options to control which processes are displayed, and how the output is formatted.

EXAMPLE: Type the command

```
ps
```

to see the status of the "interesting" jobs that belong to you. The output of the ps command, without any options specified, will include the process ID, the terminal from which the process was started, the amount of time the process has been running, and the name of the command that started the process.

EXAMPLE: Type the command

```
ps -ef
```

to see a complete listing of all the processes currently scheduled. The -e option causes ps to include all processes (including ones that do not belong to you), and the -f option causes ps to give a long listing. The long listing includes the process owner, the process ID, the ID of the parent process, processor utilization, the time of submission, the process's terminal, the total time for the process, and the command that started the process.

EXERCISE: Use the ps command, and the grep command, in a pipeline to find all the processes owned by you.

EXPLANATION: The command

```
ps -ef | grep yourusername
```

where "yourusername" is replaced by your user name, will cause the output of the ps -ef command to be filtered for those entries that contain your username.

Killing processes

Occasionally, you will find a need to terminate a process. The Unix shell provides a utility called **kill** to terminate processes. You may only terminate processes that you own (i.e., processes that you started). The syntax for the kill command is **kill [-options] process-ID**.

To kill a process, you must first find its process ID number using the ps command. Some processes refuse to die easily, and you can use the "-9" option to force termination of the job.

EXAMPLE: To force termination of a job whose process ID is 111, enter the command

```
kill -9 111
```

---

prev page | next page | upper level

Section 9: Interaction and Job Control

When you log in to a Unix system, the kernel starts a shell for you, and connects the shell to your terminal. When you execute a command from the shell, the shell creates a child process to execute the command, and connects the child process to your terminal. By connecting the child process to your terminal, the shell allows you to send input to the child process, and receive output from it. When the child process finishes, the shell regains access to the terminal, redisplays the shell prompt, and waits for your next command.

Any task that requires you to actively participate (like word processing) must be in the foreground to run. Such jobs, termed "interactive," must periodically update the display, and accept input from you, and so require access to the terminal interface.

Other jobs do not require you to participate once they are started. For example, a job that sorts the contents of one file and places the results in another file, would not have to accept user commands, or write information to the screen while it runs. Some Unix shells allow such non interactive jobs to be disconnected from the terminal, freeing the terminal for interactive use.

Note that it is even possible to log out, and leave a background process running. Unfortunately, there is no way to reconnect a background job to the terminal after you've logged out.

Jobs that are disconnected from the terminal for input and output are called "background" jobs. You can have a large number of background jobs running at the same time, but you can only have one foreground job. That is because you only have one screen, and one keyboard at your terminal.

Background and foreground jobs

The process that is connected to the terminal is called the *foreground* job. A job is said to be in the foreground because it can communicate with the user via the screen, and the keyboard.

A Unix process can be disconnected from the terminal, and allowed to run in the *background*. Because background jobs are not connected to a terminal, they cannot communicate with the user. If the background job requires interaction with the user, it will stop, and wait to be reconnected to the terminal.

Jobs that do not require interaction from the user as they run(like sorting a large file) can be placed in the background, allowing the user to access the terminal, and continue to work, instead of waiting for a long job to finish.

Starting a job in the background

To place a job in the background, simply add the ampersand character (&) to the end of your shell command.

EXAMPLE: To sort a file called "foo", and place the results in a file called "bar", you could issue

the command

```
sort < foo > bar &
```

Examining your jobs

The Unix command **jobs** allows you to see a list of all the jobs you have invoked from the current shell. The shell will list the job ID of each job, along with the status (running, stopped, or otherwise), and the command that started the job. The shell considers the most recently-created (or most recently-manipulated) job to be the *current job*, marked with a plus sign. Other jobs are referred to as *previous jobs*, and are marked with a minus sign. The commands related to job control will apply to the current job, unless directed to do otherwise. You may refer to jobs by job ID by using the percent sign. Thus, job 1 is referred to as %1, job 2 is %2, and so forth.

Suspending the foreground job

You can (usually) tell Unix to suspend the job that is currently connected to your terminal by typing Control-Z (hold the control key down, and type the letter z). The shell will inform you that the process has been suspended, and it will assign the suspended job a job ID.

There is a big difference between a suspended job, and a job running in the background. A suspended job is stopped, and no processing will take place for that job until you run it, either in the foreground, or in the background.

Placing a foreground job into the background

If you started a job in the foreground, and would like to place it in the background, the first thing you must do is suspend the job with a Control-Z, freeing your terminal. Then, you can issue the Unix command

```
bg
```

to place the suspended job in the background. The bg command can accept a job ID as an argument. If no job ID is given, bg assumes you are referring to the current (suspended) job.

Bringing a background job to the foreground

You can reconnect a background job to your terminal with the Unix command

```
fg
```

The fg command will accept a job ID as an argument. Make sure to include the percent sign:

```
fg %2
```

will bring job 2 into the foreground. If no job ID is given, fg will assume you are referring to the current (suspended) job.

Starting a suspended job

If you have a suspended job that you'd like to resume running, first you must decide whether you want it running in the foreground, or the background. Find the job ID of the suspended job with the **jobs** command, and then use **bg** (to run the job in the background), or **fg** (to run the job in the foreground).

---

prev page next page upper level

Section 10: Editing Text with EMACS

One of the most basic operations you will need to perform on a Unix system is text editing. Whether you are preparing a document, writing a program, or sending email to a colleague, you will need a utility to allow you to enter and edit text.

There are many editors available for Unix systems, but this discussion will focus on the **emacs** text editing program because of its power, flexibility, extensibility, customizability, and prevalence. No matter where you get a Unix account, you are likely to have emacs at your disposal. Here is an index to the topics in this section:

General features of the emacs editor

Emacs is a visual editor. That means that you have a representation of your entire document on your screen, and you can move around freely, editing any part of the document you wish. Older editors, referred to as *line editors*, required all changes to the file to be made on a line-by-line basis. Each command to a line editor specified a line number, and the changes to be applied to that line. Line editors are truly horrible things, and you should feel lucky if you have never seen one.

Emacs uses control and escape characters to distinguish editor commands from text to be inserted in the buffer. In this document, the notation "Control-X" means to hold down the control key, and type the letter x. You don't need to capitalize the x, or any other control character, by holding down the shift key. "ESCAPE-X" means to press the escape key down, release it, and then type x.

Working with buffers

When you edit a file in emacs, you're not really editing the file itself, as it sits out on a disk somewhere. Instead, emacs makes a copy of the file, and stores the copy in a part of RAM memory called a *buffer*. All the changes you make to the file are applied to the buffer. When you save the file, emacs writes the contents of the buffer to the disk.

Because the buffer exists in RAM memory, it disappears if the power is turned off, or if the system crashes. Thus, you should use the save command often, flushing your current buffer to disk. Once the file is on disk, a power outage or system crash shouldn't harm it.

Basic operations in emacs

Here are some of the fundamental things you'll need to do when you edit a document in emacs.

Starting emacs

To start emacs, just type the command

```
emacs
```

to the Unix shell. If you want emacs to start with a file already loaded into a buffer, type

```
emacs filename
```

where "filename" is the name of the file you want to edit.

Quitting emacs

To exit emacs and return to the Unix shell, type Control-X-Control-C. If you have made changes to the buffer since the last time you saved it to disk, emacs will ask you if you want to save. Type y for yes, or n for no.

Getting help

Emacs has an on-line help system that can be invoked by typing Control-H. If you type the question mark (?), emacs will present a list of help topics you can choose.

The emacs display

The display in emacs is divided into three basic areas. The top area is called the *text window*. The text window takes up most of the screen, and is where the document being edited appears. At the bottom of the text window, there is a single *mode line*. The mode line gives information about the document, and about the emacs session. The bottom line of the emacs display is called the *mini buffer*. The mini buffer holds space for commands that you give to emacs, and displays status information.

Aborting a command

You can abort an emacs control or escape sequence by typing the command Control-G.

Working with files

To read a disk file into an emacs buffer, type the command Control-X-Control-F. Emacs will ask you for the name of the file. As you type the name of the file, it will be displayed in the mini buffer. When you have entered the file name, press the return key, and emacs will load the file into a buffer, and display it in the text window.

The command to save the contents of the buffer to a disk file is Control-X-Control-S. The save command overwrites the old version of the file. You may also write the contents of the buffer to a different file with the command Control-X-Control-W. Emacs will prompt you for the name of the file you want to create.

To create a new file, use Control-X-Control-F, just as if the file already existed. When emacs asks you for the file name, type in the name you want your new file to have, and emacs will create the file, and display an empty buffer for you to type in.

Emacs will perform file name completion for you. Type part of the name of the file you want, and press the spacebar or tab key to get emacs to complete a file name. If the partial name you've given matches more than one file, emacs will display a list of all potential matches. You can continue typing in more of the file's name, and pressing either file completion key, until you zero in on the file you want.

Cursor motion

On well-configured systems, you will find that the keyboard arrow keys will function correctly in emacs, moving you forward or backward one character at a time, and up or down one line at a time. If the arrow keys do not work, here's how to accomplish the same functions:

- Control-F moves the cursor forward to the next character.
- Control-B moves the cursor back to the previous character.
- Control-N moves the cursor to the next line.
- Control-P moves the cursor to the previous line.

In addition to basic cursor motion, emacs provides some other handy cursor motion functions:

- Control-A moves the cursor to the start of the current line.
- Control-E moves the cursor to the end of the current line.
- ESCAPE-F moves the cursor forward to the next word.
- ESCAPE-B moves the cursor back to the previous word.
- ESCAPE-< moves the cursor to the start of the buffer.
- ESCAPE-> moves the cursor to the end of the buffer.

Inserting and deleting text

To insert text into a buffer, place the cursor where you want to start inserting text, and start typing away.

If you want to insert the contents of another file into the current buffer, place the cursor at the desired insertion point, and type Control-X-I. Emacs will ask you for the name of the file you wish to insert.

You may also insert text by cutting it from one place, and pasting it at the insertion point. See the next section for information on cutting and pasting.

Deleting text is easy. As you'd expect, the delete key deletes backward one character. Here are some other ways to delete text:

- Control-D deletes forward one letter.
- Control-K deletes from the point to the end of the line.
- ESCAPE-D deletes forward one word.
- ESCAPE-delete deletes backward one word.

Cutting and pasting text regions

Emacs allows you to select a region of text, and perform cut and paste operations on the region. It uses a temporary storage area called the "kill buffer" to allow you to store and retrieve blocks of text. There is only one kill buffer in emacs, which means that you can cut text from one document, and paste it into another.

To define a region of text, place the cursor at one end of the region and press Control-spacebar. That sets the mark. Then, move the cursor to the other end of the region. The text between the mark and the cursor defines the region.

To cut a region of text, and place it in the kill buffer, use the command Control-W (think of **W**ipe).

The paste command is Control-Y. It **Y**anks the block of text from the kill buffer, and places it where the cursor rests. The Control-Y command only retrieves the most recently-cut block of text.

You can paste in earlier cuts by pressing ESCAPE-Y. The ESCAPE-Y command, used repeatedly, will take you back through several previous text blocks that were cut. The ESCAPE-Y command does not work unless you type Control-Y first.

You may copy a region of text into the kill buffer without cutting it. Define the text block by setting the mark at one end, and moving the cursor to the other end. Then type ESCAPE-W.

Undoing changes

It is possible to undo the changes you have made to a file by entering the command Control-_. (That's Control-underscore. On some keyboards, you'll have to hold down both the control and shift keys to enter the underscore character.)

Many word processing programs can only undo the most recent command, but emacs remembers a long history of commands, allowing you to undo many changes by repeatedly entering the Control-_ code.

Customizing Emacs

The emacs editor is customizable in several ways. You can set up your own key bindings, create your own macros, and even create your own custom functions. Also, some aspects of the behavior of emacs is controlled by variables that you can set.

You can learn more about emacs functions by invoking the online help facility (by typing ESC-X help) and then typing the "f" key to list functions. Pressing the space bar for completion will cause emacs to list all the built-in functions. A list of variables can be similarly obtained by invoking the online help, then typing "v" then the spacebar.

If you place variable settings, key bindings, and function declarations, in a text file called ".emacs" in your home directory, The emacs editor will load those definitions at startup time. Here is an emacs configuration file with some basic variable definitions and key bindings for you to peruse.

---

prev page next page upper level

Section 11: The Execution Environment

CONCEPT: The exact behavior of commands issued in the shell depends upon the execution environment provided by the shell.

The Unix shell maintains a set of *environment variables* that are used to provide information, like the current working directory, and the type of terminal being used, to the programs you run. The environment variables are passed to all programs that are not built in to the shell, and may be consulted, or modified, by the program. By convention, environment variables are given in upper case letters.

To view all the environment variables, use the command

```
printenv
```

You can also view a particular environment variable using the echo command:

```
echo $TERM
```

The above command echos the value of the TERM environment variable to the standard output.

The creation of the execution environment

When you log in, a sequence of events establishes the execution environment. The exact sequence of events depends on the particular flavor of Unix, and also depends upon the default shell for your account. The following is a description of the login process for the HP-UX operating system. Other operating systems may differ.

The getty process

The getty process provides the `login:` prompt that you see on the terminal screen. The getty process reads your username, and invokes the login program.

The login program

The login program receives the username from getty, and prompts you for your password. Login then consults the system database /etc/passwd to verify your password. (Note that login will request your password even if there is no entry in /etc/passwd for the username you've given. That prevents someone from finding valid usernames by trial and error.) Login turns off terminal echo so that your password is not displayed on the screen.

Having verified your password, login then uses information in /etc/passwd to invoke your default shell. If no default shell is specified in the /etc/passwd entry, login starts the Bourne shell (/bin/sh).

Shell startup: System login scripts

When the shell program starts, it reads configuration files called *login scripts* to configure the execution environment. On HP-UX, the file /etc/profile provides initialization parameters for ksh and sh, while the file /etc/csh.login is used for csh. After the system login scripts are read, the shell looks for user-specified login scripts.

Shell startup: User login scripts

After the system login scripts are read, the shell reads user login scripts. User login scripts are kept in one's home directory, and are the means by which one can customize the shell environment. Sh and ksh look for a file called .profile. Ksh also reads a file defined in the environment variable ENV. Csh reads a file called .cshrc, and (if it is the login shell), the file .login.

Important environment variables

Here are descriptions of some of the most important environment variables, and examples of how some variables can affect the execution of commands.

TERM

The TERM environment variable defines the type of terminal that you are using. Most Unix systems have a database of terminal types, and the capabilities of each terminal type.

PATH

The PATH variable contains the names of directories to be searched for programs that correspond to command names. When you issue a command to the shell, the shell searches sequentially through each directory in the PATH list until it finds an executable program with the command name you typed.

USER

The USER variable contains your username. Any time you access a file or directory, the access permissions are checked against the value of USER.

HOME

The HOME variable contains the name of your home directory. When you issue the cd command with no directory argument, you will be placed in the directory defined in the HOME environment variable. The HOME variable is also where the shell will look for the user login scripts.

MAIL

The MAIL variable contains the name of the directory where your incoming mail is stored. When you start a mail program, the program will look in the directory stored in the MAIL environment variable for your incoming mail messages.

EDITOR

The EDITOR variable is used by programs that must invoke a text editor to provide the ability to edit or compose documents. One example is the elm program, which is used to read and send electronic mail. If you elect to compose a new mail message while in elm, the elm program will check the contents of the EDITOR variable to determine which editor to invoke.

HOST

The HOST environment variable contains the name of the host machine that is running your shell program. When you connect to a remote host through telnet or ftp, the name of your host is relayed to the remote machine, so the administrators of the remote machine can keep track of who is connecting, and from where.

Setting environment and shell variables

The exact mechanism for setting the environment and shell variables depends upon the type of shell you're using.

sh, or ksh
To set an environment variable in sh or ksh, use the syntax VAR=value;export VAR, where VAR is the name of the environment variable and value is the value you wish to assign. Do not put spaces on either side of the equals sign. The export command instructs the shell to propagate the value of the variable to all programs that are run by the shell. If an environment variable is reset, but not exported, the change will only apply to the shell itself. To set the EDITOR variable to the value emacs in ksh or sh, use the command:

```
EDITOR=emacs;export EDITOR
```

It is also possible to unset environment variables, with the unset command. Unsetting an environment variable removes the definition of the variable.

csh
To set an environment variable in csh, use the setenv command. The command has the syntax: setenv VARIABLE value. To set the EDITOR variable to the value emacs in csh, use the command:

```
setenv EDITOR emacs
```

For more information about the shell environment, consult the manual page for the shell you're using.

---

prev page next page upper level

Section 12: Customizing the Unix Shell

The Unix shell is a actually a user program that the kernel runs for you when you log in. There is usually more than one shell available on most Unix systems. The most common shells available on Unix systems are the Bourne Shell (sh), the C Shell (csh) and the Korn shell (ksh). Here is a summary of features available on these three shells, adapted from the Hewlett Packard "Beginner's Guide to HPUX."

Feature	Function	sh	csh	ksh
Job control	Allows processes to be run in the background	No	Yes	Yes
History substitution	Allows previous commands to be saved, edited, and reused	No	Yes	Yes
File name completion	Allows automatic completion of partially typed file name	No	Yes	Yes
Command line editing	Allows the use of an editor to modify the command line text	No	No	Yes
Command aliasing	Allows the user to rename commands	No	Yes	Yes

Choosing your shell

It is possible to invoke any available shell from within another shell. To start a new shell, you can simply type the name of the shell you want to run, ksh, csh, or sh.

It is also possible to set the default startup shell for all your future sessions. The default shell for your account is stored in the system database /etc/passwd, along with the other information about your account. To change your default shell, use the chsh command The chsh command requires one argument, the name of the shell you want as your default. To change tour default shell to the C shell, you could enter the command

```
chsh /bin/csh
```

On ISU's HP-UX system, the available shells are /bin/sh, /bin/posix/sh, and /bin/csh. The default shell for accounts is /bin/posix/sh, which is, for all practical purposes, the same as ksh.

Default file access permissions

Whenever you create a file or directory in a Unix filesystem, the newly created file or directory is stamped with a default set of permissions. That default set of permissions is stored in a variable called the *umask*. You can change the value of umask to suit your preferences. To see the current value of the umask variable, enter the shell command:

```
umask
```

The umask is stored as an octal (base 8) number, that defines which permissions to deny. As you recall, three kinds of file permissions (read, write, and execute) are given for each of three classes of users (owner, group, and others). Each of the nine permissions is specified as a zero (allow access), or a one (deny access).

	User	Group	Other	
	rwx	rwx	rwx	
BIT PATTERN:	000	010	010	Denies write access to group, others
OCTAL:	0	2	2	
BIT PATTERN:	000	111	111	Denies all access to group, others
OCTAL:	0	7	7	

To set your umask to deny write permission to group and others, use the command

```
umask 022
```

To deny all access to group and others, use the command

```
umask 077
```

Some versions of Unix provide a more user-friendly way of specifying your umask. In HP-UX sh-posix (or ksh), you are allowed to specify the access permissions in manner of the chmod command. The command

```
umask u=rwx,g=r,o=r
```

would set the umask to deny write and execute permissions to the group, and to others. That kind of command syntax will not work in HP-UX's C shell or Bourne shell. The HP-UX posix shell also allows the use of the command

```
umask -S
```

to print your umask setting in a more readable fashion.

Customizing with user login scripts

The remainder of this section describes some of the ways you can initialize ksh, by presenting the default user login scripts provided to users of ISU's CWIS machine. The scripts have been annotated with descriptions of the purpose of each set of commands.

The .profile file

```
#!/bin/posix/sh

#*****************************************************************************
# Run the script .shrc in every subshell.
#*****************************************************************************
ENV="$HOME/.shrc";export ENV

#*****************************************************************************
# Leave a sign of last login in user's home directory
# Remove this line if you don't want such records left.
# All logins are logged elsewhere, anyway.
#*****************************************************************************
touch $HOME/.lastlogin
chmod 600 $HOME/.lastlogin

#*****************************************************************************
# THREE TYPES OF TERMINAL-CHOOSING MECHANISMS ARE PRESENTED BELOW. ONLY
# ONE SHOULD BE USED. AUTOMATIC DETECTION OF TERMINAL TYPE IS THE DEFAULT.
#*****************************************************************************

#*****************************************************************************
# Uncomment the following line (i.e. remove # at the beginning of line)
# to make it so you're prompted for terminal type at login (rather than
# having term type hard-coded or having the system automatically detect
# your terminal type). Replace 'vt220' with another term type if you want a
# different default terminal type presented with the user prompt.
#*****************************************************************************
# eval ` tset -s -Q -m ":?vt220" `

#*****************************************************************************
# Uncomment the following line (i.e. remove # at beginning of line)
# if you want to hard-code a terminal type.
# Replace 'vt220' with any desired terminal type. Use this option only if
# you are always using the same type of terminal, and if the system doesn't
# seem to properly detect your terminal type.
```

```
#*****************************************************************************
# TERM=vt220

#*****************************************************************************
# If you chose either of the above terminal-choosing options, comment out
# the following section from 'if' to 'fi' (insert a # at the beginning of
# each line). Otherwise, the default terminal handling for terminal access
# is to have the system attempt to detect your terminal type.
#*****************************************************************************
if [ "${TERM}X" = "X" ]
then
  TERM=`ttytype`
  case $TERM in
    *2382* | *2392* | *2393* | *2394* |  *2397* | *2621*  )  TERM="hp" ;;
    *2621* | *2622* | *2623* | *2624* |  *2625* | *2626*  )  TERM="hp" ;;
    *2627* | *2628* | *2640* | *2641* |  *2644* | *2645*  )  TERM="hp" ;;
    *2647* | *2648* | *2649* | *150*  | *70092* | *70094* )  TERM="hp" ;;
  esac
fi

#*****************************************************************************
# Leave this line alone: it applies for all terminal-setting options
#*****************************************************************************
export TERM

#*****************************************************************************
# Set up the terminal:
#*****************************************************************************
stty hupcl ixon ixoff ienqak -parity
stty erase "^H" kill "^U" intr "^C" eof "^D" susp "^Z"
tabs

#*****************************************************************************
# Display little menu of options
#*****************************************************************************
/usr/local/bin/menu
```

The .shrc file

```
#!/bin/posix/sh

#*****************************************************************************
# Set default permissions so that you can read and write all files, and that
# others can't. Changing this can potentially mess up the security of your
# account, so make sure you know what you're doing before changing this.
#*****************************************************************************
umask 077

#*****************************************************************************
# set paths: PATH tells the shell where to look for programs/commands when
# you type command or program names. MANPATH tells the shell where to look
# for Unix 'man' pages. NNTPSERVER tells news readers to get Usenet news from
# the cwis computer.
#*****************************************************************************
PATH=$HOME/bin:.:/bin/posix:/bin:/usr/bin:/usr/contrib/bin:/usr/local/bin:/usr/bin/X11:/usr/local/bin/X11:/usr/contrib/bin/X11:/share/gen/bin:/share/X11/bin

MANPATH=/usr/man:/usr/local/man:/usr/contrib/man:/share/gen/man:/share/X11/man

NNTPSERVER=localhost

#*****************************************************************************
# Set up the shell environment variables:
# HOST and GROUPNAME get set for old cwis menus.
# If you want an editor other than pico (like vi or emacs) to be your default
#     editor, replace pico with the name of your preferred editor.
# HISTSIZE determines how many of your previous commands are retained for
#     recalling.
# The line with PAGER determines the default pager to use for reading through
#     documents one page at a time.
# The line with LESS makes it so informative keystroke prompts are put at the
#     bottom of the screen when using the less pager.
# LPDEST determines which printer queue your print jobs submitted with the lp
#     command go to. This line is changed by the 'printers' program, so try
#     not to radically alter this line. You can manually change the queue name
#     here if you want to.
#*****************************************************************************
export HOST=`hostname`
export GROUPNAME=`groups -p $LOGNAME`
EDITOR=pico;export EDITOR
HISTSIZE=200;export HISTSIZE
PAGER="less";export PAGER
LESS='-c -P spacebar\:page ahead b\:page back /\:search ahead \?\:search back h\:help
q\:quit';export LESS
LPDEST=laser_q2;export LPDEST

#*****************************************************************************
# Treat unset parameters as an error when substituting.
# Don't mess with this unless you're a guru.
#*****************************************************************************
set +u

#*****************************************************************************
# Make it so your terminal is not open to talk requests and placement of
# comments on your screen by other users. Change this line to 'mesg y' if
# you want to be open to talk requests by default. Otherwise, you can type
# 'mesg y' within a session to temporarily open yourself for talk requests.
#*****************************************************************************
mesg n

#*****************************************************************************
# Set up shell for vi-style command line editing (i.e. recalling commands with
# ESC-k, and using vi editor keystrokes to edit command lines).) If you prefer,
# you can replace 'vi' with 'emacs' for emacs-style command line editing (i.e.
```

```
# recalling commands with control-P, and using emacs editor keystrokes to edit
# the command line.
#*****************************************************************************
set -o vi

#*****************************************************************************
# Define user prompt to show hostname and current directory.
# This can be changed to anything.
# Change it to
#
#    PS1 = 'GET TO WORK, BOZO!! '
#
# if you want the command prompt to repeatedly insult you.
#*****************************************************************************
PS1='cwis:$PWD
$ '

#*****************************************************************************
# Create custom commands. All can be removed/altered *except* 'printers.
# Feel free to create your own custom commands.
# The command 'printers' is necessary for the proper functioning of the cwis
#   printer-choosing utility. The 'printers' program changes the line that
#   sets default printer queue in this file, and this file gets "run" again
#   to load the new value into the user environment.
# The command 'more' is altered to call the more capable 'less' pager. Disable
#   this alias if you want to use the 'more' pager.
# The command 'ls' is set up to identify executable
#    files with an * and directories with a /. Remove this alias if you want
#    the ls command to behave normally, i.e. without the * and /.
# The command 'edit' is set up to call default editor (see EDITOR=... above).
# The command 'oldmenu' brings up the old cwis menus.
# The command 'logout' calls the Unix command 'exit' to log out.
# The command 'webperms' sets up file permissions to be world-readable, for
#     web publishing.
# The command 'regperms' returns file permissions to readable by user only.
#*****************************************************************************
alias printers="/share/gen/bin/printers;. $HOME/.shrc"
alias more="less"
alias ls="ls -F"
alias dir="ls -F"
alias edit="$EDITOR"
alias oldmenu=". /usr/local/bin/cwis2"
alias logout="exit"
alias quit="exit"
alias bye="exit"
alias log="exit"
alias webperms="umask 022;chmod -R a+r $HOME/public_html"
alias regperms="umask 077"
```

Section 13: Interactive Use of the Shell

This section discusses tips and tricks to make your use of the shell more efficient.

File name completion

Both ksh and csh will perform file name completion for you. You can type in a partial file name, and press the ESCAPE key (once for csh, twice for ksh). The shell will then complete the name of the file for you. If no file exists that begins with the characters you typed, the shell will beep at you, and will not complete the name. If more than one file begins with the characters you typed, the shell will complete the name up to the point where the names differ. Then you can type additional letters to specify the file name you want, reusing the ESCAPE key if desired.

Command name aliasing

Both csh and ksh provide command name aliasing, to allow you to rename commands. Aliasing can save a lot of keystrokes if you must frequently issue the same lengthy command. The alias command requires two pieces of information: The command you wish to alias, and the alias you wish to use to refer to it.

EXAMPLE: To alias the "history" command to "hi" you could use the following command in the Korn shell:

```
alias hi='history'
```

After entering that alias, you could type the command "hi" and the shell would substitute "hi" with the string "history" before executing it. The same command could be accomplished in the C shell with the syntax:

```
alias hi history
```

EXERCISE: Create an alias in the Korn shell called "clean" that would remove any files from your home directory that have the extension .gif or .jpg.

EXPLANATION: The command

```
alias clean='rm ~/*.gif; rm ~/*.jpg'
```

would work.

Command aliasing can be tricky. Surround the alias string with single quotes (') to prevent the shell from interpreting special characters. If you use command history substitution in an alias, use the backslash character (\) to escape characters that you don't want the shell to interpret.

EXAMPLE: This example, written for the C shell, creates an alias for the cd command, so that it stores the current location in a shell variable called old before it changes to the new location. It also creates a new command alias called back that allows us to go back to the previous location:

```
alias cd 'set old=$cwd; chdir \!*; pwd'
```

```
alias back 'set foo=$old; cd $foo; unset foo'
```

There are several things to note in the above example. The alias for cd has three parts: The first reads the current working directory from the shell variable cwd, and saves it in a shell variable called old. The second part uses history substitution and chdir to change the current location. The use of chdir prevents an "aliasing loop," where the cd command calls itself. The third part executes the pwd command to print the new location on the screen.

The alias for back also has three parts: The first part reads the previous location from the shell variable old, and stores it in a shell variable called foo. That is necessary because the new cd alias will change the value of old when we call it in the second part of the back alias. The third part cleans up our mess by unsetting the variable foo, removing it from the environment.

You can remove an alias using the unalias command. To remove the "clean" alias you created in a previous exercise, enter the command:

```
unalias clean
```

Command history substitution

The C shell and Korn shell will keep an ordered list of the commands you have issued, and allow you to retrieve commands from the list. That facility, called *command history substitution*, makes it possible to reuse all or part of your previously issued commands. Each command on the list is given a command number, according to the order it was issued. You can view the command history list by issuing the command:

```
history
```

The exact mechanism of retrieving commands from the command history list depends on the shell you're using, and how you have customized your shell.

ksh

When using the Korn shell, the number of commands remembered by the shell is controlled by the HISTSIZE environment variable. Use the command

```
HISTSIZE=50;export HISTSIZE
```

to set the length of the history list to fifty. By default, the history size is set to 128 lines.

The shell command "set -o" is used to specify the editing mode for the command line, either emacs or vi. Since an earlier section of this workshop dealt with emacs, we will confine our discussion to the emacs editing style. To use the emacs editing mode, enter the command

```
set -o emacs
```

In emacs editing mode, recall previous commands with the emacs command for "previous line," or Control-P. Repeated use of Control-P will recall earlier commands. You can also use the emacs command for "next line," or Control-N, to go forward through your command history, toward more recently-issued commands. You can only use Control-N after you have used Control-P at least once.

csh

The C shell allows you to recall previous commands in whole or in part. In the C shell, the history shell variable is used to specify the number of lines the shell will remember. The statement

```
set history=60
```

will cause the C shell to remember sixty commands.

To recall previous commands from the history list, the C shell uses the exclamation point (!) character, sometimes referred to in computer jargon as "bang." The bang character can be used in combination with history line numbers, and text patterns. Here are some examples of how to use history substitution in the C shell:

Recall the last command:

```
!!
```

Recall the third most recent command:

```
!-3
```

Recall command number ten from the history list:

```
!10
```

Recall the last command that began with the letters "ls":

```
!ls
```

You can also recall specific pieces of previous commands, and use them to create new commands. The colon character is used to select specific words from a command. Each word in the command is referred to by position. The command name itself is item number zero. Here are some examples:

Recall the third word from the last command:

```
!:2
```

Perform an "ls" on the second word from command number 8:

```
ls !8:1
```

Use more to view the last item from command number ten:

```
more !10:$
```

Editing the command line

The Korn shell (ksh) provides the ability to edit the command history list almost as if your were in an editor program like vi, or emacs. The current command is always the last line in the history list, and begins blank. You can type in a new command, or recall an earlier command from the list. You can also modify the text on the current line using basic text editor commands.

The mechanism for setting the editor style in ksh is the "set -o" command. To edit in emacs mode, issue the command:

```
set -o emacs
```

Manipulating command line text in emacs mode is done in much the same way as text editing with emacs. When you finish editing the command line, press the return key to issue the command to the shell. You might want to go back and refresh your memory on emacs by reviewing section ten of this tutorial, titled "Text Editing with Emacs."

Section 14: The Unix File System

Most Unix machines store their files on magnetic disk drives. A disk drive is a device that can store information by making electrical imprints on a magnetic surface. One or more heads skim close to the spinning magnetic plate, and can detect, or change, the magnetic state of a given spot on the disk. The drives use disk controllers to position the head at the correct place at the correct time to read from, or write to, the magnetic surface of the plate. It is often possible to partition a single disk drive into more than one logical storage area. This section describes how the Unix operating system deals with a raw storage device like a disk drive, and how it manages to make organized use of the space.

How the Unix file system works

Every item in a Unix file system can be defined as belonging to one of four possible types:

Ordinary files

Ordinary files can contain text, data, or program information. An ordinary file cannot contain another file, or directory. An ordinary file can be thought of as a one-dimensional array of bytes.

Directories

In a previous section, we described directories as containers that can hold files, and other directories. A directory is actually implemented as a file that has one line for each item contained within the directory. Each line in a directory file contains only the name of the item, and a numerical reference to the location of the item. The reference is called an *i-number*, and is an index to a table known as the *i-list*. The i-list is a complete list of all the storage space available to the file system.

Special files

Special files represent input/output (i/o) devices, like a tty (terminal), a disk drive, or a printer. Because Unix treats such devices as files, a degree of compatibility can be achieved between device i/o, and ordinary file i/o, allowing for the more efficient use of software. Special files can be either *character special files*, that deal with streams of characters, or *block special files*, that operate on larger blocks of data. Typical block sizes are 512 bytes, 1024 bytes, and 2048 bytes.

Links

A link is a pointer to another file. Remember that a directory is nothing more than a list of the names and i-numbers of files. A directory entry can be a *hard link*, in which the i-number points directly to another file. A hard link to a file is indistinguishable from the file itself. When a hard link is made, then the i-numbers of two different directory file entries point to the same inode. For that reason, hard links cannot span across file systems. A *soft link* (or *symbolic link*) provides an indirect pointer to a file. A soft link is implemented as a directory file entry containing a pathname. Soft links are distinguishable from files, and can span across file systems. Not all versions of Unix support soft links.

The I-List

When we speak of a Unix file system, we are actually referring to an area of physical memory represented by a single i-list. A Unix machine may be connected to several file systems, each with its own i-list. One of those i-lists points to a special storage area, known as the *root file system*. The root file system contains the files for the operating system itself, and must be available at all times. Other file systems are removable. Removable file systems can be attached, or *mounted*, to the root file system. Typically, an empty directory is created on the root file system as a mount point, and a removable file system is attached there. When you issue a cd command to access the files and directories of a mounted removable file system, your file operations will be controlled through the i-list of the removable file system.

The purpose of the i-list is to provide the operating system with a map into the memory of some physical storage device. The map is continually being revised, as the files are created and removed, and as they shrink and grow in size. Thus, the mechanism of mapping must be very flexible to accommodate drastic changes in the number and size of files. The i-list is stored in a known location, on the same memory storage device that it maps.

Each entry in an i-list is called an *i-node*. An i-node is a complex structure that provides the necessary flexibility to track the changing file system. The i-nodes contain the information necessary to get information from the storage device, which typically communicates in fixed-size *disk blocks*. An i-node contains 10 direct pointers, which point to disk blocks on the storage device. In addition, each i-node also contains one *indirect pointer*, one *double indirect pointer*, and one *triple indirect pointer*. The indirect pointer points to a block of direct pointers. The double indirect pointer points to a block of indirect pointers, and the triple indirect pointer points to a block of double indirect pointers. By structuring the pointers in a geometric fashion, a single i-node can represent a very large file.

It now makes a little more sense to view a Unix directory as a list of i-numbers, each i-number referencing a specific i-node on a specific i-list. The operating system traces its way through a file path by following the i-nodes until it reaches the direct pointers that contain the actual location of the file on the storage device.

The file system table

Each file system that is mounted on a Unix machine is accessed through its own block special file. The information on each of the block special files is kept in a system database called the file system table, and is usually located in /etc/fstab. It includes information about the name of the device, the directory name under which it will be mounted, and the read and write privileges for the device. It is possible to mount a file system as "read-only," to prevent users from changing anything.

File system quotas

Although not originally part of the Unix filesystem, quotas quickly became a widely-used tool. Quotas allow the system administrator to place limits on the amount of space the users can allocate. Quotas usually place restrictions on the amount of space, and the number of files, that a user can take. The limit can be a *soft limit*, where only a warning is generated, or a *hard limit*, where no

further operations that create files will be allowed.

The command

```
quota
```

will let you know if you're over your soft limit. Adding the -v option will provide statistics about your disk usage.

File system related commands

Here are some commands related to file system usage, and other topics discussed in this section:

bdf
: On HP-UX systems, reports file system usage statistics

df
: On HP-UX systems, reports on free disk blocks, and i-nodes

du
: Summarizes disk usage in a specified directory hierarchy

ln
: Creates a hard link (default), or a soft link (with -s option)

mount, umount
: Attaches, or detaches, a file system (super user only)

mkfs
: Constructs a new file system (super user only)

fsck
: Evaluates the integrity of a file system (super user only)

A brief tour of the Unix filesystem

The actual locations and names of certain system configuration files will differ under different implementations of Unix. Here are some examples of important files and directories under version 9 of the HP-UX operating system:

/hp-ux
: The kernel program

/dev/
: Where special files are kept

/bin/
: Executable system utilities, like sh, cp, rm

/etc/
: System configuration files and databases

/lib/
: Operating system and programming libraries

/tmp/
: System scratch files (all users can write here)

/lost+found/
: Where the file system checker puts detached files

/usr/bin/
: Additional user commands

/usr/include/
: Standard system header files

/usr/lib/
: More programming and system call libraries

/usr/local/
: Typically a place where local utilities go

/usr/man
: The manual pages are kept here

Other places to look for useful stuff

If you get an account on an unfamiliar Unix system, take a tour of the directories listed above, and familiarize yourself with their contents. Another way to find out what is available is to look at the contents of your PATH environment variable:

```
echo $PATH
```

You can use the ls command to list the contents of each directory in your path, and the man command to get help on unfamiliar utilities. A good systems administrator will ensure that manual pages are provided for the utilities installed on the system.

---

prev page upper level