SYLLABUS

UNIX OPERATING SYSTEM

Overview of UNIX Operating System, basic features of Unix operating System, File Structure, CPU Scheduling, Memory Management, File System Implementation of Operating System Functions in UNIX.

Starting Of Unix and Text Manipulation and user-to-user communication

User Names and Groups, Logging In, Format of Unix Commands, Changing your password, Unix Documentation, Files and Directories, File permission, Basic Operation on Files, Changing Permission Modes, Standard files, Processes

Inspecting Files, Operating On Files, Printing Files, Rearranging Files, Sorting Files, Splitting Files, Translating Characters, On line communication, Off line communication.

VI EDITORS

General characteristics, Adding text and Navigation, changing text, searching for text, copying and Moving text, Features of Ex, Line Editors Ex and Ed, Stream editor SED, changing several files in SED, AWK.

Shell Programming

Programming in the Bourne and C-Shell, Wild Cards, Simple Shell program, variables, Programming Construct, Interactive Shell scripts, Advanced Features, Unix Compiler, Maintaining program

System Administration

Define system Administration, Booting the system, Maintaining User Accounts, File System, and special files, Backup and Restoration

Text Book:

Unix Concept and application- Sumitabhadas

References:

Unix Shell Programming-Yashwant Kanetkar

Unix Programming Environment- RobPike

Unix in a Nutshell- Donill Gily

Contents

4 SHELL PROGRAMMING

5

System administration

1

UNIX OPERATING SYSTEM

1.1 Unix introduction

Unix is the multi-user multitasking Operating system. UNIX is an operating system which was first developed in the 1960s, and has been under constant development ever since. By operating system, we mean the suite of programs which make the computer work. It is a stable, multi-user, multi-tasking system for servers, desktops and laptops.

UNIX systems also have a graphical user interface (GUI) similar to Microsoft Windows, which provides an easy to use environment. However, knowledge of UNIX is required for operations, which aren't covered by a graphical program, or for when there are no windows interface available, for example, in a telnet session.

1.2 History of Unix

The Unix operating system found its beginnings in MULTICS, which stands for Multiplexed Operating and Computing System. The MULTICS project began in the mid 1960s as a joint effort by General Electric, Massachusetts Institute for Technology and Bell Laboratories. In 1969 Bell Laboratories pulled out of the project.

One of Bell Laboratories people involved in the project was Ken Thompson. He liked the potential MULTICS had, but felt it was too complex and that the same thing could be done in simpler way. In 1969 he wrote the first version of Unix, called UNICS. UNICS stood for Uniplexed Operating and Computing System. Although the operating system has changed, the name stuck and was eventually shortened to Unix.

Ken Thompson teamed up with Dennis Ritchie, who wrote the first C compiler. In 1973 they rewrote the Unix kernel in C. The following year a version of Unix known as the Fifth Edition was first licensed to universities. The Seventh Edition, released in 1978, served as a dividing point for two divergent lines of Unix development. These two branches are known as SVR4 (System V) and BSD.

Ken Thompson spent a year's sabbatical with the University of California at Berkeley. While there he and two graduate students, Bill Joy and Chuck Haley, wrote the first Berkely version of Unix, which was distributed to students. This resulted in the source code being worked on and developed by many different people. The Berkeley version of Unix is known as BSD, Berkeley Software Distribution. From BSD came the vi editor, C shell, virtual memory, Sendmail, and support for TCP/IP.

For several years SVR4 was the more conservative, commercial, and well supported. Today SVR4 and BSD look very much alike. Probably the biggest cosmetic difference between them is the way the *ps* command functions.

The Linux operating system was developed as a Unix look alike and has a user command interface that resembles SVR4.

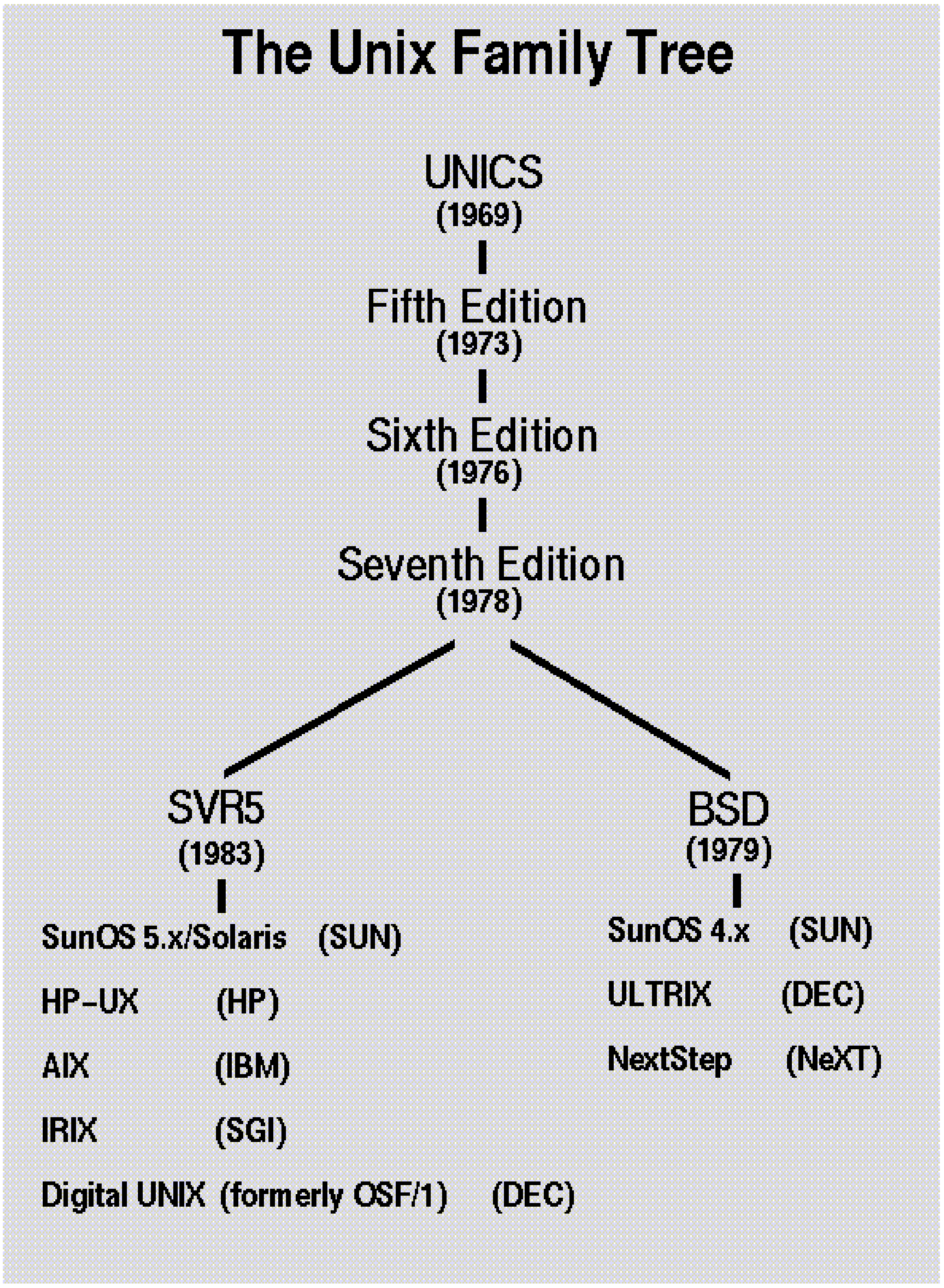
- 

Figure 1: The Unix Family Tree

1.2.1 The Connection Between Unix and C

At the time the first Unix was written, most operating systems developers believed that an operating system must be written in an assembly language so that it could function effectively and gain access to the hardware. Not only was Unix innovative as an operating system, it was ground-breaking in that it was written in a language (C) that was not an assembly language.

The C language itself operates at a level that is just high enough to be portable to variety of computer hardware. A great deal of publicly available Unix software is distributed as C programs that must be complied before use.

Many Unix programs follow C's syntax. Unix system calls are regarded as C functions.

What this means for Unix system administrators is that an understanding of C can make Unix easier to understand.

1.2.2 Why Use Unix?

One of the biggest reasons for using Unix is networking capability. With other operating systems, additional software must be purchased for networking. With Unix, networking capability is simply part of the operating system. Unix is ideal for such things as world wide e-mail and connecting to the Internet.

Unix was founded on what could be called a "small is good" philosophy. The idea is that each program is designed to do one job well. Because Unix was developed different people with different needs it has grown to an operating system that is both flexible and easy to adapt for specific needs.

Unix was written in a machine independent language. So Unix and unix-like operating systems can run on a variety of hardware. These systems are available from many different sources, some of them at no cost. Because of this diversity and the ability to utilize the same "user-interface" on many different systems, Unix is said to be an open system.

1.3 Components of UNIX operating system

The UNIX operating system is made up of three parts; the kernel, the shell and the programs.

The kernel

The kernel of UNIX is the hub of the operating system: it allocates time and memory to programs and handles the file store and communications in response to system calls.

As an illustration of the way that the shell and the kernel work together, suppose a user types rm myfile (which has the effect of removing the file myfile). The shell searches the file store for the file containing the program rm, and then requests the kernel, through system calls, to execute the program rm on myfile. When the process rm myfile has finished running, the shell then returns the UNIX prompt % to the user, indicating that it is waiting for further commands.

The shell

The shell acts as an interface between the user and the kernel. When a user logs in, the login program checks the username and password, and then starts another program called the shell. The shell is a command line interpreter (CLI). It interprets the commands the user types in and arranges for them to be carried out. The commands are themselves programs: when they terminate, the shell gives the user another prompt (% on our systems).

The adept user can customize his/her own shell, and users can use different shells on the same machine. Staff and students in the school have the tcsh shell by default.

The tcsh shell has certain features to help the user inputting commands.

Filename Completion - By typing part of the name of a command, filename or directory and pressing the [Tab] key, the tcsh shell will complete the rest of the name automatically. If the shell finds more than one name beginning with those letters you have typed, it will beep, prompting you to type a few more letters before pressing the tab key again.

History - The shell keeps a list of the commands you have typed in. If you need to repeat a command, use the cursor keys to scroll up and down the list or type history for a list of previous commands.

Files and processes

Everything in UNIX is either a file or a process.

A process is an executing program identified by a unique PID (process identifier).

A file is a collection of data. They are created by users using text editors, running compilers etc.

Examples of files:

- a document (report, essay etc.)
- the text of a program written in some high-level programming language
- instructions comprehensible directly to the machine and incomprehensible to a casual user, for example, a collection of binary digits (an executable or binary file);
- a directory, containing information about its contents, which may be a mixture of other directories (subdirectories) and ordinary files.

The Directory Structure

All the files are grouped together in the directory structure. The file-system is arranged in a hierarchical structure, like an inverted tree. The top of the hierarchy is traditionally called root (written as a slash /)

In the diagram above, we see that the home directory of the undergraduate student "ee51vn" contains two sub-directories (docs and pics) and a file called report.doc.

The full path to the file report.doc is "/home/its/ug1/ee51vn/report.doc"

1.4 Basic Characteristics of Unix

Unix is, at its base level, a multi-user, multitasking, virtual memory operating system that runs on a wide variety of hardware platforms. This means that Unix is able to do many things at the same time, for many different users, and using more memory than it really has physically installed. From a user's perspective this is very nice, and from an operating systems point of view, it is very interesting. But UNIX really is much more than just an operating system; it is a philosophy of using and programming a computer that gets its power from the relationships between programs rather than the programs themselves. This is evident from many of the design points of UNIX, which will be mentioned later.

Let's look at each of the three characteristics of Unix listed above.

a) Unix is a multi-user system. This means that inherent to Unix is the idea that there are different users of the system, and that different users may have different sorts of privileges and types of access to different parts of the system. It allows for the idea that some users may want to protect some of their data from being accessed by other users on the system. So, in being a multi-user system, the basic ideas of system security and data privacy come up. Likewise, there must be some way to identify one user from another in a multi-user system. Unix uses a system of login names to identify users and passwords to authenticate that a user is, in fact, who she claims to be.

b) Unix is a multitasking system. This means that Unix has the ability to handle more than one task at a time. These tasks might be several programs that any particular user wants to run, or they may be programs run by several users at once. Multitasking, combined with being a multi-user system, makes it possible for more than one person to be logged in and using a Unix system at once. This is true for any Unix system, not just large timesharing systems. Even small, desktop systems have the capability to support multiple concurrent users. Additionally, some of the tasks that the system could be doing, and probably is doing, can be system tasks related to keeping the system going and providing the services needed by the users.

c) Unix is a virtual memory operating system. The concept of virtual memory and memory management is a bit beyond the scope of this seminar, but a few key concepts are important to note. In a virtual memory system, the system behaves as if it has much more memory than is physically installed. Some portion of the disk is used to simulate extra memory. This idea makes it possible to run large programs on a smaller system. Some tradeoff exists between the amount of memory that a program uses and how fast it will run, since if it is very large, it must access the disk a lot, but it will eventually run. In a multitasking environment, tasks that are not currently running can be moved out of memory and onto disk to free up memory for tasks that more urgently need it. The overall result is better use of the system resources for all involved.

1.5 Unix File structures

Once we understand the Unix file system, its structure, and how to move around in it, you will be able to easily interact with Unix. It is important that you remember and understand that in Unix, everything is treated as a file. Unix files are for the most part simply collections of bytes. There is no concept of a record in Unix. Some programs choose to treat certain bytes (like newline characters) differently than others, but to Unix, they're just bytes. Directories are basically just files as well, although they *contain* other files. Special files are still just files to Unix and include things like external devices, memory, and terminals.

Since Unix is a multi-user system, each user is allocated his own file space. Each user has a directory on the system where his files will be kept. This file is known as that user's *home* directory. When you log into a Unix system, you are automatically placed in your home directory. You can create, delete, and modify files in this directory, but probably not in many other places on the system. Likewise, other users are not able to access files in your home directory unless you allow them to.

Let's assume that you have logged in and some files already exist in your account. It would be helpful to be able to find out what files are there. The ls command is used to list files in Unix. Using the ls command might give an output something like the following:

```
% ls

baby.1 ig.discography src

bin mbox unix.refer
```

From this, we can see that several files exist in our account already. We don't know anything about them, but that won't bother us for now. What we care about now is the fact that we can tell what files exist. ls has given us a list of the names of the files. Notice that the list of files is formatted nicely to fit within the width of the screen. ls works at making the output look relatively attractive and not jumbled. File names are laid out in columns, the number of columns depending on the size of the display and the size of the filenames. By default, the names are sorted alphabetically.

1.5.1 Unix Command Format

Since ls is the first Unix command that we have looked at, now is a good time to stand back and see how most Unix commands work. In almost all cases, the syntax of the Unix command is the same and follows the pattern:

command options arguments

where command is the command that you want to execute, options are different flags and options to the command, usually preceded with a minus sign, and argument specifies what to do the command to. It is not always necessary to give options and arguments to commands, since most commands have a default set of options and a default object of their action. Some commands have only a few options while other have so many options.

1.5.2 Listing Files with ls

The ls command, as it turns out, has many modifiers and options. These change things, like which files are listed, in what order the files are sorted, and what additional information to give about the files.

For example, if you name a file so that the first character of its name is a dot (.), then ls does not, by default, show you this file. This is helpful so that you don't see the many system files that control your Unix session each time you list your files. However, the modifier a causes ls to display all of the files, even ones starting with dot. The special files . and .. will be discussed later, along with the purpose of many of the other dot files.

```
          % ls a
          . .emacs .mailrc bin unix.refer
          .. .exrc .plan ig.discography
          .aliases .login .rhosts mbox
.cshrc .logout baby.1 src
```

Files have many more attributes than just their names. Each file has a size, an owning user and owning group, a date of last modification, as well as other system related information. The l option to ls shows you more about the files.

```
% ls l

total 23

rwrr 1 dickson 2639 Jan 30 18:02 baby.1

drwxrxrx 2 dickson 512 Jan 30 18:02 bin

rwrr 1 dickson 9396 Jan 30 18:03 ig.discography

rw 1 dickson 5239 Jan 30 18:10 mbox

drwxrwxrx 3 dickson 512 Jan 30 18:06 src

rwrr 1 dickson 1311 Jan 30 18:03 unix.refer
```

This listing shows us several fields of information for each file. The first field specifies the file type and access permissions. More about this shortly. The second field tells the number of links to the file. For now, we can disregard this. Links are sort of like aliases for files and are generally important only to the system. The third column tells us what user owns the file. For all of these files, since they are our files, we own them. The next column shows us how many bytes are in the file. Next comes the time when the file was last modified. Finally comes the name of the file. There are many, many more options to ls, so refer to man ls for the details.

1.5.3 File Types, Permissions, and Modes

At this point, we need to talk a little bit about file access permissions and file types. The field in ls for file types and permissions is made up of ten characters, never more, never less. The first character tells us what kind of file this is. Most commonly, there will be a in this field specifying that this is a plain file. The second most common file type to be found here is d for a directory. Any other character means that it is a special file. These would include things like symbolic links, different kinds of devices, pipes, etc. It is not really important at this point to distinguish between types of special files, but it is important to be able to recognize them. In the example above, we can see that src and bin are both directories and that the rest of the files are just plain files.

The next nine characters specify who is able to read, write, and execute this file. Unix access permissions are broken into three classes: the owner of the file, other people in the same account group as the owner, and the rest of the world. Every user is part of at least one account group. This is just a collection of users assigned to a group due to a similar need for access to data, enrollment in a class, or any other reason that the system administrator might choose. Groups can only be created by the system administrator.

Just as there are three groups for access permissions, the permission field is broken into three sets of three characters. In each set of three characters, the first is for permission to read the file. Second is the permission to write or change the file, and last is the ability to execute the file. If a letter is found in the field corresponding to a permission, then that permission is granted. If a is found, then that permission is denied.

With respect to the three sets of letters, the first triplet corresponds to the permissions that the owner of the file has, the second set of three belongs to the group permission, and the last set of three corresponds to the rest of the users on the system. In the example above, we can see that for all of the files above, the owner of the file is able to read and write the file. None of the plain files are executable by anyone. This would indicate that they are not things that are meant to be run, but rather are probably just some sort of data. All of the files, except the one named mbox are readable by other people in the user's account group, as well as anyone else on the system. mbox, the mailbox for the user, is protected so that only he can read its contents. Notice that for the two directories, bin and src, they have execute permissions set.

Permissions are interpreted differently for directories than for plain files. Read permission on a directory grants the ability to list the contents of that directory (in other words, see the list of files in it). Write permission on a directory grants the ability to create or remove files in the directory. In order to delete a file, you must have write permission on the directory, not the file. Finally, execute permission on a directory grants the ability to change to (cd) that directory.

1.5.4 Changing permissions with chmod

What if we decided that we didn't want people outside of our group to be able to read what was in unix.refer? The chmod (CHange MODe) command alters the access permissions to files. If we wanted to remove the right of anyone else to read unix.refer, the command

```
chmod or unix.refer
```

would do the trick.

The modifiers for chmod are fairly complex. In general, they can be summarized by saying that you specify whose permissions you are modifying (u for the user, g for the group, and o for others), what you are doing to the permission (+ to add, to remove, or = to set), and what permissions you plan to modify (r for read, w for write, and x for execute). See man chmod for the details. If we then decided that we want to allow people within our group to modify this bibliography (refer is the standard Unix bibliography program), we would use the following command:

```
chmod g+w unix.refer
```

The following shows this transaction and its output.

```
% chmod or unix.refer

% chmod g+w unix.refer

% ls l unix.refer

rwrw 1 dickson 1311 Jan 30 18:03 unix.refer
```

We can now see that the changes we wanted have been made to the file permissions.

<u>1.5.5 Manipulating files with cp and mv</u>

1.5.5.1 cp command

The cp command is used to copy a file from one file to another or one directory to another. In general, cp makes an identical copy of a file, updating its last time of modification, but maintaining most of its permissions (subject to some conditions see man umask for more details). If you want to copy a file and preserve its modification time and its permissions, use the p option. If you want to copy a directory and all of its contents, use the r, for recursive, option.

The following is an example of just copying a file.

```
% cp unix.refer NewBibliography.refer

% ls l *.refer

rwr 1 dickson 1311 Jan 30 19:20 NewBibligraphy.refer

rwrw 1 dickson 1311 Jan 30 18:03 unix.refer
```

In the following example, we will copy several files into a directory.

```
% mkdir bibs

% cp *.refer bibs

% ls l bibs

total 4

rwr 1 dickson 1311 Jan 30 19:42 NewBibligraphy.refer

rwr 1 dickson 1311 Jan 30 19:42 unix.refer
```

1.5.5.2 mv command

If you want to change the name of a file, mv will rename the file. Actually, mv moves the file within the filesystem, but that is, for all intents and purposes, the same thing. You can move one file or directory to another, just renaming it. Or, you can move one or more files or directories into a target directory. If we had wanted to move our bibliographies from the above example into the bibs directory instead of copying them, the following sequence of commands would have done the trick.

```
% mkdir bibs

% mv *.refer bibs

% ls l bibs

total 4

rwr 1 dickson 1311 Jan 30 19:42 NewBibligraphy.refer

rwr 1 dickson 1311 Jan 30 19:42 unix.refer
```

1.5.5.3 Viewing Files

So far, we have seen how to list, copy, and rename files. Another necessary operation is to view the contents of a file. There are a large number of commands in Unix that will display the contents of a file in a variety of ways. The simplest of these is the cat command. cat, in this application, reads the named file and displays it on the screen. In actuality, cat is doing something a little simpler and more generic than that even. We'll talk more about that shortly, as well as looking at a number of other commands for viewing files. The following is a simple example of the cat command.

```
% cat ShortFile

This is a short file so that we can see what a file looks like.

There are only about four lines in this file.

And this is the fourth line.
```

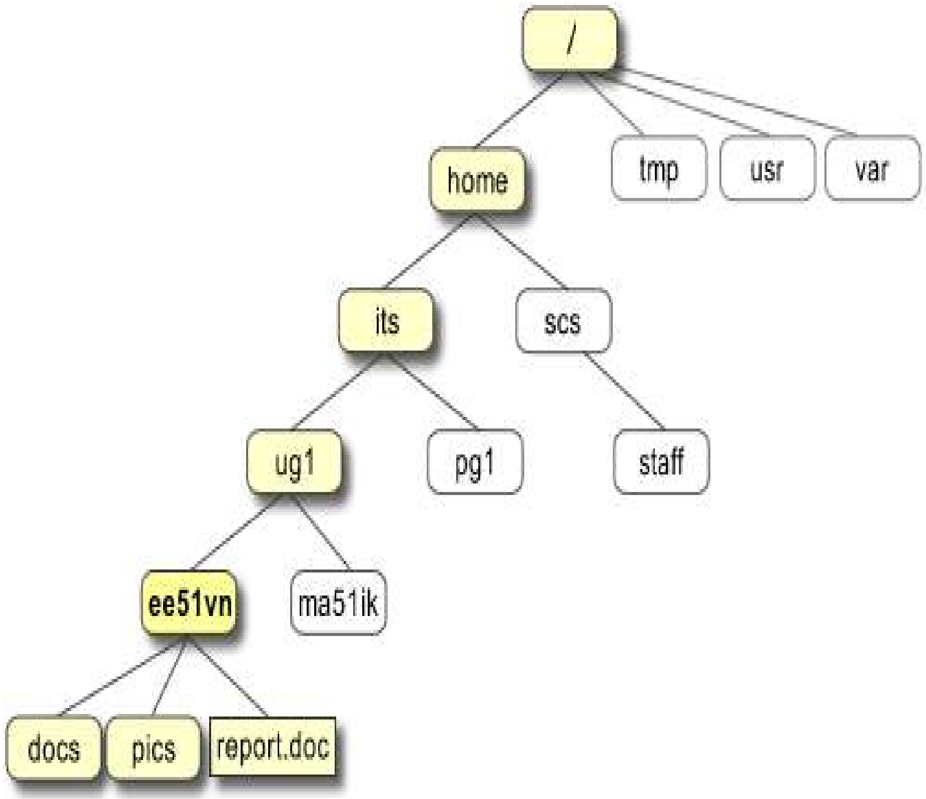
1.6 Directories and the Unix File System

1.6.1 What are Directories?

So far, we have mentioned directories several times and used them as objects and targets for copying or renaming files, but have not discussed just what the filesystem looks like and how directories are arranged.

Directories are basically just files that contain other files. If you are familiar with MSDOS, the idea is just the same. If you are a Macintosh user, directories are very similar to folders. A directory can contain any sort of file that you might want, including other directories. This leads to a tree structured file system, with the top of the file system being a directory called the root and labeled as / . The files in a directory are called the children its children. So, every file on the system is either a child of the root, or a descendent at some level of the root.

When you create a new directory, it is not completely empty at its creation time. Every directory has at least two children, . and .. . The directory labeled . refers to the directory itself, and .. refers to its immediate parent directory. These allow us to have a larger degree of flexibility in naming files.



1.6.2 Current and Home Directories

Now, it is important to understand the idea of your current directory. At any time, you will be positioned within some directory on the system. This is just like in MSDOS, and is similar to working with files in a particular folder on a Mac. The directory where you are sitting is known as your current directory. The command pwd will display to you your current directory.

```
% pwd
/home/iris2/class9
```

When you first log into a Unix machine, your current directory is set for you to your home directory. This is the directory on the system that will hold your personal files. In this directory, you can create, delete, and modify files, as well as controlling how others access your files.

1.6.3 Naming Unix Files and Directories

Within the Unix directory structure, there are two ways to name any file: relative naming, and absolute naming.

1.6.3.1 Absolute Naming

An absolute name, or absolute path as it is often called, specifies exactly where in the filesystem the particular file is. It tells the whole name of the file, starting at the root of the filesystem. An absolute name starts with /, the root, and names each directory along the path to the file, separating the directories with /. This is very similar to MSDOS, except for the direction of the slash and the fact that there is no disk drive designation. As an example, the absolute name for your mailbox might be /home/iris2/class9/mbox. The pwd command always reports an absolute pathname.

1.6.3.2 Relative Naming

The second way to name a file in Unix is with a relative name. Whereas an absolute name specifies exactly where in the filesystem a particular file exists, a relative name specifies how to get to it from your current directory. The look of a relative name may vary a lot, since depending on your starting directory, there are a number of paths to a particular file.

In the simplest case, just naming a file in your current directory is a relative name. You are specifying how to get to this file from your current directory, and the path is to just open the contained in the current directory.

When using relative paths, the special directories . and .. that are contained in every directory are used quite a bit. Recall that . specifies the directory itself, and .. specifies its parent. So, if the file mbox is contained in your current directory, naming the file with ./mbox and mbox are equivalent. The special directory .. is used to name a directory at the same level in the tree as your current directory, that is, a sibling of your current directory. The following example illustrates using .. to look at a sibling directory.

```
% pwd
    /home/iris2/class9
    % ls
    NewBibligraphy.refer bibs mbox
    ShortFile bin src
    baby.1 ig.discography unix.refer
    % cd bin
    % ls
    pwgen
    % ls ../src
    helloworld.c pwgen.c
```

1.6.3.3 Shortcuts for File Naming

Since certain file naming patterns are used over and over, Unix provides some shortcuts for file naming. Actually, it is the shell that provides these, but the distinction is not critical at this point. In particular, very many file accesses are either to your own home directory or to the home directory of another user. To make it easier to point to these places, the ~ character is used. Alone, ~ refers to your home directory. So the file ~/mbox refers to the file mbox in your home directory. Likewise ~username refers to the home directory of that particular user. So, ~dickson/mbox refers to the file mbox in the user dickson's home directory.

1.6.3.4 File Naming Limitations

Unix allows you great flexibility in naming your files. Older System V Release 3 systems limited the length of a filename to 14 characters. Berkeley Unix systems, as well as newer versions of System V have substantially relaxed this limitation. Many systems allow the name of individual files to be up to 256 characters, and the maximum absolute pathname to be about 1023 characters. This means that files can be given meaningful names quite easily. Also, since Unix is sensitive to the case of the filenames, you can use mixed case names to add clarity. While long, meaningful names can be quite nice, it is also possible to go overboard with file naming. So, try to give your files meaningful names, but try not to overburden yourself or others that have to access the files with overly long names.

1.6.3.5 File Name Extensions

While Unix does not actually have the same notion of a file extension as is found in some other systems, notably MSDOS, many user and applications programs behave as it did. Unix does not consider the character. Any differently than any other character in a file name. However, applications programs often use it to add an extension to a filename that specifies something about the contents of the file. These extensions sometimes tell what programs were used to create a file. If more than one was used, and is needed to decode the file, multiple extensions are used. Typically, the extension will tell what language the data in the file is in. A good example of this is the C compiler. It assumes that its input files will have an extension of .c. Its executable output generally has no extension, unlike MSDOS which uses a .EXE extension for executables. If we have a program called hello.c, its executable version is likely to be called just hello. Table 1 gives some of the more common file extensions and what they are used for.

Table 1: Standard Unix Filename Extensions

.c	C language source
.f	Fortran language source
.o	Object code from a compiler
.pl	Perl language source
.ps	PostScript language source
.tex	TeX document
.dvi	TeX device independent output
.gif	CompuServ GIF image
.jpg	JPEG image
.1.8	Unix Manual Page
.tar	tar format archive
.Z	Compressed file, made with compress
.tar.Z	Compressed tar archive
.a	ar library archive file
.gz	Compressed file, made with gzip
.tar.gz	Compressed (gzip) tar archive

1.6.4 More About Listing Directories

The ls command has some additional options to it that deserve mention in our discussion of directories. These are the F and R options. The F option to ls tells it to note the type of file for each file in the directory. This is done by adding a special character to symbolize the type of file to the names in the listing. It is important not to confuse the marker character with the name of the file. Plain files are given no marker, since these are far and away the most common sort of file on the system. Directories are given a trailing /, executable files show a trailing *, symbolic links (which we have not yet discussed) show a trailing @, and some special files use a few other special characters such as =.

The R option will give a recursive listing of the files in a directory. First all of the contents of the directory are listed, then the ls R command is issued for each child directory. Like all options to ls, these may be used separately or grouped together.

```
% ls F

NewBibligraphy.refer bibs/ mbox

ShortFile bin/ src/

baby.1 ig.discography unix.refer

% ls RF

NewBibligraphy.refer bibs/ mbox

ShortFile bin/ src/

baby.1 ig.discography unix.refer

bibs:

NewBibligraphy.refer unix.refer

bin:
```

```
pwgen*


src:

helloworld.c pwgen.c
```

1.6.5 Moving Around the File System

Now that we have an understanding of how our files are laid out, it would be helpful to know how to move around the file system. Similar to MSDOS, the cd command is used to change from one directory to another. Its use is simple; its one argument is the directory that you want to move to.

```
% pwd
/home/iris2/class9
% cd bin
% pwd
/home/iris2/class9/bin
```

1.6.6 Creating and Removing Directories

Also similar to MSDOS, the mkdir and rmdir commands are used to create and remove directories. The arguments to mkdir are a list of directories to be created. The arguments to rmdir are a list of directories to be removed. In order to remove a directory with rmdir, the directory must be empty. If you want to remove a directory that contains files, use the rm r command, which recursively removes all files and directories in the directory given as its argument.

1.7 Standard Filesystem Layout

Within the Unix filesystem, there are some standard places to put different types of files. This makes it easy to find things, regardless of which Unix system you are using.

Some standard conventions are used: bin is used for binaries or executable files, lib is used as the library, src is used for source code, etc is used for miscellaneous system files. Directories that are children of the directory /usr are used for things that users would encounter, as opposed to system files. The following table lists some commonly found directories and their usual contents.

/	Root of the filesystem
/bin	Basic Unix commands
/dev	Special device files
/etc	System configuration files
/home	Use home directories
/tmp	Temporary files
/usr	The bulk of the operating system files
/usr/local	Local additions and modifications to Unix
/var	Log files and frequently changed files

These are only a few of the standard directories that might be found on a Unix machine. Check your system's manual for specifics.

1.7.1 Using MetaCharacters for File Matching

Sometimes, it is easier to specify some pattern of characters that match the names of files rather than listing all of the files themselves. Metacharacters make this easier. These are characters that are used for pattern matching. Some of these will be familiar to you from other computer systems, such as MSDOS, however, Unix pattern matching is much stronger than most other systems. The following table shows some of the most commonly used metacharacters.

*	Matches 0 or more characters
?	Matches exactly 1 character
[]	Matches any single character listed inside the brackets
[]	Matches any single character listed in a range of characters
{}	Matches any word listed within the braces
\	Removes the special meaning of any of the above metacharacters

1.8 Data Redirection and Pipes

Early on, we said one of the real strengths of the Unix system was its ability to allow programs to work together well. One of the pillars of this ability is the concept of data redirection. Every command has three files associated with it: stdin, stdout, and stderr. stdin is the standard input, the default file from which data will be read. By default, this is usually the keyboard of the terminal you are logged onto. stdout is the standard place that output will go, also usually the screen of the terminal you are logged onto. stderr is the standard place for error or diagnostic output to go, again, usually the screen as well.

Every Unix program that runs implicitly uses these three files. If we could connect the standard input and output facilities to real files, or better yet, other programs, then we could really chain together some interesting combinations. Well, as it turns out, this is not only possible, but is one of the cornerstones of Unix. It is possible to redirect the input, output, and error streams to or from files. Likewise it is possible to chain programs together into pipes. Some examples will clarify what we mean.

The cat program discussed earlier reads a file, but if no file is specified, it will read from its standard input. We can make it take its standard input from a file by using the < symbol. Angle brackets are used for redirection. An easy way to remember which is which is to remember that the bracket points in the direction the data flows.

```
% cat ShortFile

This is a short file so that we can see what a file looks like.


There are only about four lines in this file.

And this is the fourth line.
```

We can send the output to a new file using the > symbol. Since the diff command, which lists the differences between files doesn't flag any differences; we can tell that ShortFile and NewFile are identical.

```
% cat < ShortFile > NewFile

% diff ShortFile NewFile

%
```

The >> symbol acts just like > except that the output is appended to the file, instead of creating a new file.

Several other data redirection symbols are available. You can read about these in man csh.

In addition to being able to redirect our input and output to files, we can direct it to other programs using pipes and the | symbol. Pipes cause the output from one program to become the input to the next program in the list. This can be simple, or can lead to powerful and complex pipelines. For example, we can pipe cat into pr to make the output prettier, and pipe that to more so that we can look at things one page at a time.

```
% cat newrut | pr | more
```

Using pipes, you can connect as many different utilities as you might want to complete a particular job. MSDOS also provides pipes, but while one program is running, the others are waiting for it. In Unix, all of the different parts of the pipeline are running concurrently, waiting only when they must for input from a previous step. Pipes, then, with data redirection, make it possible to construct quite powerful tools.

1.9 CPU Scheduling

- What is CPU scheduling? Determining which processes run when there are multiple runnable processes. Why is it important? Because it can have a big effect on resource utilization and the overall performance of the system.
- By the way, the world went through a long period (late 80's, early 90's) in which the most popular operating systems (DOS, Mac) had NO sophisticated CPU scheduling algorithms. They were single threaded and ran one process at a time until the user directs them to run another process. Why was this true? More recent systems (Windows NT) are back to having sophisticated CPU scheduling algorithms. What drove the change, and what will happen in the future?
- Basic assumptions behind most scheduling algorithms:
    - There is a pool of runnable processes contending for the CPU.
    - The processes are independent and compete for resources.
    - The job of the scheduler is to distribute the scarce resource of the CPU to the different processes ``fairly'' (according to some definition of fairness) and in a way that optimizes some performance criteria.

  In general, these assumptions are starting to break down. First of all, CPUs are not really that scarce - almost everybody has several, and pretty soon people will be able to afford lots. Second, many applications are starting to be structured as multiple cooperating processes. So, a view of the scheduler as mediating between competing entities may be partially obsolete.

- How do processes behave? First, CPU/IO burst cycle. A process will run for a while (the CPU burst), perform some IO (the IO burst), then run for a while more (the next CPU burst). How long between IO operations? Depends on the process.
    - IO Bound processes: processes that perform lots of IO operations. Each IO operation is followed by a short CPU burst to process the IO, then more IO happens.
    - CPU bound processes: processes that perform lots of computation and do little IO. Tend to have a few long CPU bursts.

  One of the things a scheduler will typically do is switch the CPU to another process when one process does IO. Why? The IO will take a long time, and don't want to leave the CPU idle while wait for the IO to finish.

- When look at CPU burst times across the whole system, have the exponential or hyperexponential distribution in Fig. 5.2.
- What are possible process states?
    - Running - process is running on CPU.
    - Ready - ready to run, but not actually running on the CPU.
    - Waiting - waiting for some event like IO to happen.
- When do scheduling decisions take place? When does CPU choose which process to run? Are a variety of possibilities:
    - When process switches from running to waiting. Could be because of IO request, because wait for child to terminate, or wait for synchronization operation (like lock acquisition) to complete.
    - When process switches from running to ready - on completion of interrupt handler, for example. Common example of interrupt handler - timer

interrupt in interactive systems. If scheduler switches processes in this case, it has preempted the running process. Another common case interrupt handler is the IO completion handler.

  - When process switches from waiting to ready state (on completion of IO or acquisition of a lock, for example).
  - When a process terminates.

- How to evaluate scheduling algorithm? There are many possible criteria:
  - **CPU Utilization:** Keep CPU utilization as high as possible. (What is utilization, by the way?).
  - **Throughput**: number of processes completed per unit time.
  - **Turnaround Time:** mean time from submission to completion of process.
  - **Waiting Time:** Amount of time spent ready to run but not running.
  - **Response Time:** Time between submission of requests and first response to the request.
  - **Scheduler Efficiency:** The scheduler doesn't perform any useful work, so any time it takes is pure overhead. So, need to make the scheduler very efficient.
- Big difference: Batch and Interactive systems. In batch systems, typically want good throughput or turnaround time. In interactive systems, both of these are still usually important (after all, want some computation to happen), but response time is usually a primary consideration. And, for some systems, throughput or turnaround time is not really relevant - some processes conceptually run forever.
- **Difference between long and short term scheduling.** Long term scheduler is given a set of processes and decides which ones should start to run. Once they start running, they may suspend because of IO or because of preemption. Short term scheduler decides which of the available jobs that long term scheduler has decided are runnable to actually run.
- Let's start looking at several vanilla scheduling algorithms.

- First-Come, First-Served. One ready queue, OS runs the process at head of queue, new processes come in at the end of the queue. A process does not give up CPU until it either terminates or performs IO.
- Consider performance of FCFS algorithm for three compute-bound processes. What if have 4 processes P1 (takes 24 seconds), P2 (takes 3 seconds) and P3 (takes 3 seconds). If arrive in order P1, P2, P3, what is
    - Waiting Time? (24 + 27) / 3 = 17
    - Turnaround Time? (24 + 27 + 30) = 27.
    - Throughput? 30 / 3 = 10.

  What about if processes come in order P2, P3, P1? What is

    - Waiting Time? (3 + 3) / 2 = 6
    - Turnaround Time? (3 + 6 + 30) = 13.
    - Throughput? 30 / 3 = 10.
- **Shortest-Job-First (SJF)** can eliminate some of the variance in Waiting and Turnaround time. In fact, it is optimal with respect to average waiting time. Big problem: how does scheduler figure out how long will it take the process to run?
- **For long term scheduler** running on a batch system, user will give an estimate. Usually pretty good - if it is too short, system will cancel job before it finishes. If too long, system will hold off on running the process. So, users give pretty good estimates of overall running time.
- **For short-term scheduler,** must use the past to predict the future. Standard way: use a time-decayed exponentially weighted average of previous CPU bursts for each process. Let T_n be the measured burst time of the nth burst, s_n be the predicted size of next CPU burst. Then, choose a weighting factor w, where $0 <= w <= 1$ and compute $s_{n+1} = w\ T_n + (1 - w)s_n$. s_0 is defined as some default constant or system average.

- *We* tell how to weight the past relative to future. If choose $w = .5$, last observation has as much weight as entire rest of the history. If choose $w = 1$, only last observation has any weight. Do a quick example.
- **Preemptive vs. Non-preemptive SJF scheduler**. Preemptive scheduler reruns scheduling decision when process becomes ready. If the new process has priority over running process, the CPU preempts the running process and executes the new process. Non-preemptive scheduler only does scheduling decision when running process voluntarily gives up CPU. In effect, it allows every running process to finish its CPU burst.
- Consider 4 processes P1 (burst time 8), P2 (burst time 4), P3 (burst time 9) P4 (burst time 5) that arrive one time unit apart in order P1, P2, P3, P4. Assume that after burst happens, process is not reenabled for a long time (at least 100, for example). What does a preemptive SJF scheduler do? What about a non-preemptive scheduler?
- **Priority Scheduling.** Each process is given a priority, then CPU executes process with highest priority. If multiple processes with same priority are runnable, use some other criteria - typically FCFS. SJF is an example of a priority-based scheduling algorithm. With the exponential decay algorithm above, the priorities of a given process change over time.
- Assume we have 5 processes P1 (burst time 10, priority 3), P2 (burst time 1, priority 1), P3 (burst time 2, priority 3), P4 (burst time 1, priority 4), P5 (burst time 5, priority 2). Lower numbers represent higher priorities. What would a standard priority scheduler do?
- Big problem with priority scheduling algorithms: starvation or blocking of low-priority processes. Can use aging to prevent this - make the priority of a process go up the longer it stays runnable but isn't run.
- **What about interactive systems?** Cannot just let any process run on the CPU until it gives it up - must give response to users in a reasonable time. So, use an algorithm called round-robin scheduling. Similar to FCFS but with preemption.

Have a time quantum or time slice. Let the first process in the queue run until it expires its quantum (i.e. runs for as long as the time quantum), then run the next process in the queue.

- Implementing round-robin requires timer interrupts. When schedule a process, set the timer to go off after the time quantum amount of time expires. If process does IO before timer goes off, no problem - just run next process. But if process expires its quantum, do a context switch. Save the state of the running process and run the next process.
- How well does RR work? Well, it gives good response time, but can give bad waiting time. Consider the waiting times under round robin for 3 processes P1 (burst time 24), P2 (burst time 3), and P3 (burst time 4) with time quantum 4. What happens, and what is average waiting time? What gives best waiting time?
- What happens with really a really small quantum? It looks like you've got a CPU that is 1/n as powerful as the real CPU, where n is the number of processes. Problem with a small quantum - context switch overhead.
- What about having a really small quantum supported in hardware? Then, you have something called multithreading. Give the CPU a bunch of registers and heavily pipeline the execution. Feed the processes into the pipe one by one. Treat memory access like IO - suspend the thread until the data comes back from the memory. In the meantime, execute other threads. Use computation to hide the latency of accessing memory.
- What about a really big quantum? It turns into FCFS. Rule of thumb - want 80 percent of CPU bursts to be shorter than time quantum.
- Multilevel Queue Scheduling - like RR, except have multiple queues. Typically, classify processes into separate categories and give a queue to each category. So, might have system, interactive and batch processes, with the priorities in that order. Could also allocate a percentage of the CPU to each queue.
- Multilevel Feedback Queue Scheduling - Like multilevel scheduling, except processes can move between queues as their priority changes. Can be used to give

IO bound and interactive processes CPU priority over CPU bound processes. Can also prevent starvation by increasing the priority of processes that have been idle for a long time.

- A simple example of a multilevel feedback queue scheduling algorithm. Have 3 queues, numbered 0, 1, 2 with corresponding priority. So, for example, execute a task in queue 2 only when queues 0 and 1 are empty.
- A process goes into queue 0 when it becomes ready. When run a process from queue 0, give it a quantum of 8 ms. If it expires its quantum, move to queue 1. When execute a process from queue 1, give it a quantum of 16. If it expires its quantum, move to queue 2. In queue 2, run a RR scheduler with a large quantum if in an interactive system or an FCFS scheduler if in a batch system. Of course, preempt queue 2 processes when a new process becomes ready.
- Another example of a multilevel feedback queue scheduling algorithm: the Unix scheduler. We will go over a simplified version that does not include kernel priorities. The point of the algorithm is to fairly allocate the CPU between processes, with processes that have not recently used a lot of CPU resources given priority over processes that have.
- Processes are given a base priority of 60, with lower numbers representing higher priorities. The system clock generates an interrupt between 50 and 100 times a second, so we will assume a value of 60 clock interrupts per second. The clock interrupt handler increments a CPU usage field in the PCB of the interrupted process every time it runs.
- The system always runs the highest priority process. If there is a tie, it runs the process that has been ready longest. Every second, it recalculates the priority and CPU usage field for every process according to the following formulas.
    - CPU usage field = CPU usage field / 2
    - Priority = CPU usage field / 2 + base priority
- So, when a process does not use much CPU recently, its priority rises. The priorities of IO bound processes and interactive processes therefore tend to be

high and the priorities of CPU bound processes tend to be low (which is what you want).

- Unix also allows users to provide a ``nice" value for each process. Nice values modify the priority calculation as follows:
    - Priority = CPU usage field / 2 + base priority + nice value

  So, you can reduce the priority of your process to be ``nice" to other processes (which may include your own).

- In general, multilevel feedback queue schedulers are complex pieces of software that must be tuned to meet requirements.
- Anomalies and system effects associated with schedulers.
- Priority interacts with synchronization to create a really nasty effect called priority inversion. A priority inversion happens when a low-priority thread acquires a lock, then a high-priority thread tries to acquire the lock and blocks. Any middle-priority threads will prevent the low-priority thread from running and unlocking the lock. In effect, the middle-priority threads block the high-priority thread.
- How to prevent priority inversions? Use priority inheritance. Any time a thread holds a lock that other threads are waiting on, give the thread the priority of the highest-priority thread waiting to get the lock. Problem is that priority inheritance makes the scheduling algorithm less efficient and increases the overhead.
- Preemption can interact with synchronization in a multiprocessor context to create another nasty effect - the convoy effect. One thread acquires the lock, then suspends. Other threads come along, and need to acquire the lock to perform their operations. Everybody suspends until the lock that has the thread wakes up. At this point the threads are synchronized, and will convoy their way through the lock, serializing the computation. So, drives down the processor utilization.
- If have non-blocking synchronization via operations like LL/SC, don't get convoy effects caused by suspending a thread competing for access to a resource. Why

not? Because threads don't hold resources and prevent other threads from accessing them.

- Similar effect when scheduling CPU and IO bound processes. Consider a FCFS algorithm with several IO bound and one CPU bound process. All of the IO bound processes execute their bursts quickly and queue up for access to the IO device. The CPU bound process then executes for a long time. During this time all of the IO bound processes have their IO requests satisfied and move back into the run queue. But they don't run - the CPU bound process is running instead - so the IO device idles. Finally, the CPU bound process gets off the CPU, and all of the IO bound processes run for a short time then queue up again for the IO devices. Result is poor utilization of IO device - it is busy for a time while it processes the IO requests, then idle while the IO bound processes wait in the run queues for their short CPU bursts. In this case an easy solution is to give IO bound processes priority over CPU bound processes.
- In general, a convoy effect happens when a set of processes need to use a resource for a short time, and one process holds the resource for a long time, blocking all of the other processes. Causes poor utilization of the other resources in the system.

1.10 Memory Management

Unlike traditional PC operating systems, Unix related systems use very sophisticated memory management algorithms to make efficient use of memory resources. This makes the questions "*How much memory do I have?*" and "*How much memory is being used?*" rather complicated to answer. First you must realize that there are three different kinds of memory, three different ways they can be used by the operating system, and three different ways they can be used by processes.

Kinds of Memory:

- **Main -** The physical Random Access Memory located on the CPU motherboard that most people think of when they talk about RAM. Also called Real Memory. This does not include processor caches, video memory, or other peripheral memory.
- **File System** - Disk memory accessible via pathnames. This does not include raw devices, tape drives, swap space, or other storage not addressable via normal pathnames. It does include all network file systems.
- **Swap Space** - Disk memory used to hold data that is not in Real or File System memory. Swap space is most efficient when it is on a separate disk or partition, but sometimes it is just a large file in the File System.

1.10.1 OS Memory Uses:

- Kernel - The Operating System's own (semi-)private memory space. This is always in Main memory.
- Cache - Main memory that is used to hold elements of the File System and other I/O operations. Not to be confused with the CPU cache or or disk drive cache, which are not part of main memory.
- Virtual - The total addressable memory space of all processes running on the given machine. The physical location of such data may be spread among any of the three kinds of memory.

1.10.2 Process Memory Uses:

- Data - Memory allocated and used by the program (usually via *malloc*, *new*, or similar runtime calls).
- Stack - The program's execution stack (managed by the OS).
- Mapped - File contents addressable within the process memory space.

The amount of memory available for processes is at least the size of Swap, minus Kernel. On more modern systems (since around 1994) it is at least Main plus Swap minus Kernel and may also include any files via mapping.

1.10.3 Swapping

Virtual memory is divided up into pages, chunks that are usually either 4096 or 8192 bytes in size. The memory manager considers pages to be the atomic (indivisible) unit of memory. For the best performance, we want each page to be accessible in Main memory as it is needed by the CPU. When a page is not needed, it does not matter where it is located.

The collection of pages which a process is expected to use in the very near future is called its resident set. (Some OSs consider all the pages currently located in main memory to be the resident set, even if they aren't being used.) The process of moving some pages out of main memory and moving others in, is called swapping.

A page fault occurs when the CPU tries to access a page that is not in main memory, thus forcing the CPU to wait for the page to be swapped in. Since moving data to and from disks takes a significant amount of time, the goal of the memory manager is to minimize the number of page faults.

Where a page will go when it is "swapped-out" depends on how it is being used. In general, pages are swapped out as follows:

Kernel

Never swapped out.

Cache

Page is discarded.

Data

Moved to swap space.

Stack

Moved to swap space.

Mapped

Moved to originating file if changed and shared.

Moved to swap space if changed and private.

It is important to note that swapping itself does not necessarily slow down the computer. Performance is only impeded when a page fault occurs. At that time, if memory is scarce, a page of main memory must be freed for every page that is needed. If a page that is being swapped out has changed since it was last written to disk, it can't be freed from main memory until the changes have been recorded (either in swap space or a mapped file).

Writing a page to disk need not wait until a page fault occurs. Most modern UNIX systems implement preemptive swapping, in which the contents of changed pages are copied to disk during times when the disk is otherwise idle. The page is also kept in main memory so that it can be accessed if necessary. But, if a page fault occurs, the system can instantly reclaim the preemptively swapped pages in only the time needed to read in the new page. This saves a tremendous amount of time since writing to disk usually takes two to four times longer than reading. Thus preemptive swapping may occur even when main memory is plentiful, as a hedge against future shortages.

Since it is extremely rare for all (or even most) of the processes on a UNIX system to be in use at once, most of virtual memory may be swapped out at any given time without significantly impeding performance. If the activation of one process occurs at a time when another is idle, they simply trade places with minimum impact. Performance is only significantly affect when more memory is needed at once than is available. This is discussed more below.

1.10.4 Mapped Files

The subject of file mapping deserves special attention simply because most people, even experienced programmers, never have direct experience with it and yet it is integral to the function of modern operating systems. When a process maps a file, a segment of its virtual memory is designated as corresponding to the contents of the given file. Retrieving data from those memory addresses actually retrieves the data from the file. Because the retrieval is handled transparently by the OS, it is typically much faster and more efficient than the standard file access methods. (See the manual page *mmap* and its associated system calls.)

In general, if multiple processes map and access the same file, the same real memory and swap pages will be shared among all the processes. This allows multiple programs to share data without having to maintain multiple copies in memory.

The primary use for file mapping is for the loading of executable code. When a program is executed, one of the first actions is to map the program executable and all of its shared libraries into the newly created virtual memory space. (Some systems let you see this effect by using *trace*, *ktrace*, *truss*, or *strace*, depending on which UNIX you have. Try this on a simple command like *ls* and notice the multiple calls to mmap.)

As the program begins execution, it page faults, forcing the machine instructions to be loaded into memory as they are needed. Multiple invocations of the same executable, or programs which use the same code libraries, will share the same pages of real memory.

What happens when a process attempts to change a mapped page depends upon the particular OS and the parameters of the mapping. Executable pages are usually mapped "read-only" so that a segmentation fault occurs if they are written to. Pages mapped as "shared" will have changes marked in the shared real memory pages and eventually will be written back to the file. Those marked as "private" will have private copies created in swap space as they are changed and will not be written back to the originating file.

Some older operating systems (such as SunOS 4) always copy the contents of a mapped file into the swap space. This limits the quantity of mapped files to the size of swap space, but it means that if a mapped file is deleted or changed (by means other than a mapping), the mapping processes will still have a clean copy. Other operating systems (such as SunOS 5) only copy privately changed pages into swap space. This allows an arbitrary quantity of mapped files, but means that deleting or changing such a file may cause a bus in error in the processes using it.

1.10.5 How much memory is there?

The total real memory is calculated by subtracting the kernel memory from the amount of RAM. (Utilities like *top* or *yamm* can show you the total real memory available.) Some of the rest may be used for caching, but process needs usually take priority over cached data.

The total virtual memory depends on the degree to which processes use mapped files. For data and stack space, the limitation is the amount of real and swap memory. On some systems it is simply the amount of swap space, on others it is the sum of the two. If mapped files are automatically copied into swap space, then they must also fit into swap memory making that amount the limiting factor. But if mapped files act as their own swap area, or if swap space is just a growable file in the file system, then the limit to the amount of virtual memory that could be mapped onto them is the amount of hard drive space available.

In practice, it is easy and cheap to add arbitrary amounts of swap space and thus virtual memory. The real limiting factor on performance will be the amount real memory.

1.10.6 How much memory is being used?

If no programs were sharing memory or mapping files, you could just add up their resident sets to get the amount of real memory in use and their virtual memories to get the amount of swap space in use. But shared memory means that the resident sets of multiple processes may be counting the same real memory pages more than once. Likewise, mapped files (on OSs that use them for swapping) will count toward a process' virtual memory use but won't consume swap space. The issue is further confused by the fact that the system will use any available RAM for disk caching. Since cache memory is low priority and can be freed at any time, it doesn't impede process performance, but is usually counted as "used".

Thus on most OSs there is no easy way to calculate how much memory is being used for what. Some utility programs, like *top* or *yamm*, may be able to give you an idea, but their numbers can be misleading since its not possible distinguish different types of use. Just because *top* says you have very little memory free doesn't necessarily mean you need to buy more. A better indication is to look for swap activity numbers. These may be reported as "swap-outs", or "pageouts", and can usually be found using utilities like "vm_stat". If this number is frequently increasing, then you may need more RAM.

One of the best and simplest indications of memory problems is to simply listen to the hard drive. If there is less memory available than the total resident sets of all running processes (after accounting for sharing), then the computer will need to be continuously swapping. This non-stop disk activity is called thrashing and is an indication that there are too many active processes or that more memory is needed. If there is just barely enough memory for the resident sets but not enough for all virtual memory, then thrashing will occur only when new programs are run or patterns of user interaction change. If there is more real memory than virtual memory, then there will be plenty of extra for disk caching and repeated launching of applications or access to files will produce little or no disk sounds.

2 STARTING OF UNIX

Starting of Unix and Text manipulation

2.1 Usernames

Every person who uses a UNIX computer should have an *account*. An account is identified by a *username.* Traditionally, each account also has a secret *password* associated with it to prevent unauthorized use. Usernames are sometimes called *account names*. You need to know both your username and your password to log into a UNIX system. For example, Rama has an account on his college computer system. His username is rama. His password is "ayodhya" When he wants to log into the college computer system, he types:

```
login: rama
password: ayodhya
```

The username is an *identifier:* it tells the computer who you are. The password is an *authenticator*: you use it to prove to the operating system that you are who you claim to be.

Standard UNIX usernames may be between one and eight characters long. Within a single UNIX computer, usernames must be unique: no two users can have the same one. A single person can have more than one UNIX account on the same computer. In this case, each account would have its own username. A username can be any sequence of characters you want (with some exceptions), and does not necessarily correspond to a real person's name.

Your username identifies you to UNIX the same way your first name identifies you to your friends. When you log into the UNIX system, you tell it your username the same way you might say, "Hello, this is Sabrina," when you pick up the telephone.

<u>**2.2 What is a 'password'?**</u>

Passwords are one form of user authentication. This means the system relies on something you know to validate who is logging onto the server. This works based on the idea of each user having a unique login, and a secret password that only they know. Under this model, the system verifies your password and knows it is truly you logging in. The problem with this, is that the unix system assumes only you have your password. It does not make provisions or understand that you may not be the only one with your password. Some examples of why you may not be the only one include:

* Writing it down and losing the paper
* Someone watching your keystrokes as you log in
* A network intruder snooping your password via technical means
* Someone guessing your password

With that in mind, it is apparent that you need to have a secret password, that only you know, that cannot be guessed. Your administrator is responsible for the security of your system and helping prevent network intruders from gaining your password. However, it is EVERYONE'S responsibility on the system.

2.3 Entering Your Password

Telling the computer your password is the way that you prove to the computer that you are you. In classical security parlance, your password is what the computer uses to *authenticate* your *identity* (two words that have a lot of significance to security gurus, but generally mean the same thing that they do to ordinary people).

When you log in, you tell the computer who you are by typing your username at the login prompt. You then type your password (in response to the password prompt) to prove that you are who you claim to be. For example:

```
login: sarah
password: tuna4fis
```

As we mentioned above, UNIX does not display your password when you type it.

If the password that you supply with your username corresponds to the one on file, UNIX logs you in and gives you full access to all of your files, commands, and devices. If either the password or the username does not match, UNIX does not log you in.

On some versions of UNIX, if somebody tries to log into your account and supplies an invalid password several times in succession, your account will be locked. Only the system administrator can unlock a locked account. Locking has two functions:

1. It protects the system from someone who persists in trying to guess a password; before they can guess the correct password, the account is shut down.
2. It notifies you that someone has been trying to break into your account.

If you find yourself locked out of your account, you should contact your system administrator and get your password changed to something new. Don't change your password back to what it was before you were locked out.

Changing Your Password

You can change your password with the UNIX `passwd` command. `passwd` first asks you to type your old password, then asks for a new one. By asking you to type your old password first, `passwd` prevents somebody from walking up to a terminal that you left yourself logged into and then changing your password without your knowledge.

UNIX makes you type the password twice when you change it:

```
% passwd
Changing password for sarah.
Old password:tuna4fis
New password: nosmis32
Retype new password: nosmis32
%
```

If the two passwords you type don't match, your password remains unchanged. This is a safety precaution: if you made a mistake typing the new password and UNIX only asked you once, then your password could be changed to some new value and you would have no way of knowing that value.

2.4 Verifying Your New Password

After you have changed your password, try logging into your account with the new password to make sure that you've entered the new password properly. Ideally, you should do this without logging out, so you will have some recourse if you did not change your password properly. This is especially crucial if you are logged in as *root* and you have just changed the *root* password.

2.5 Unix File

2.5.1 Comparing files

You can display the line-by-line difference between two files with the **diff** command.

diff ***file1 file2***

The information given by the command tells you what changes need to be made for *file1* and *file2* to match. If there is no difference between the files you are returned to the shell prompt.

diff indicates which lines need be added (`a`), deleted (`d`) or changed (`c`).

Lines in *file1* are identified with a (<) symbol: lines in *file2* with a (>) symbol.

Examples of using the diff command

To compare the contents of two files:

```
diff email addresses
2a3,4
> Jean JRS@pollux.ucs.co
> Jim  jim@frolix8
```

This displays a line by line difference between the file `email` and `addresses`.

To make these files match you need to add (`a`) lines 3 and 4 (`3,4`) of the file `addresses` (>) after line 2 in the file `email`.

Here are the contents of files `email` and `addresses` used in this example. Line numbers are shown at the beginning of each line to help you follow this example.

email	addresses
1 John erpl08@ed	1 John erpl08@ed
2 Joe CZT@cern.ch	2 Joe CZT@cern.ch
3 Kim ks@x.co	3 Jean JRS@pollux.ucs.co
4 Keith keith@festival	4 Jim jim@frolix8
	5 Kim ks@x.co
	6 Keith keith@festival

2.5.2 Controlling access to your files and directories

Every file and directory in your account can be protected from or made accessible to other users by changing its access permissions.

You can only change the permissions for files and directories that you own.

2.5.3 Displaying access permissions

To display the access permissions of a file or directory use the command:

ls -l *filename* (*directory*)

This displays a one line summary for each file or directory. For example:

```
-rwxr-xr-x  1 erpl08   staff      3649 Feb 22 15:51 my.html
```

This first item `-rwxr-xr-x` represents the access permissions on this file. The following items represent the number of links to it; the username of the person owning it; the name of the group which owns it; its size; the time and date it was last changed, and finally, its name.

2.5.4 Understanding access permissions

There are three types of permissions:

r read the file or directory
w write to the file or directory
x execute the file or search the directory

Each of these permissions can be set for any one of three types of user:

u the user who owns the file (usually you)

g members of the group to which the owner belongs

o all other users

The access permissions for all three types of user can be given as a string of nine characters:

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

These permissions have different meanings for files and directories.

Examples of access permissions

ls -l file1

```
-rw-------  2 unixjohn     3287 Apr  8 12:10 file1
```

The owner of the file has read and write permission.

---

ls -l file2

```
-rw-r--r--  2 unixjohn     3287 Apr  8 12:11 file2
```

The owner has read and write permission. Everyone else - the group and all other users - can read the file.

---

ls -l myprog

```
-rwx--x--x  2 unixjohn     3287 Apr  8 12:10 myprog
```

The user has read, write and execute permission. Everyone else -the group and all others- can execute the file.

---

ls -l

...

```
  drwxr-x---  2 erpl08       1024 Jun 17 10:00 SCCS
```

This is a directory. The user has read, write and execute permission. The group has read and execute permission on the directory. Nobody else can get access to it.

2.5.5 Default access permissions

When you create a file or directory its access permissions are set to a default value. These are usually:

`rw-------`
: gives you read and write permission for your files; no access permissions for the group or others.

`rwx------`
: gives you read write and execute permission for your directories; no access permissions for the group or others.

Access permissions for your home directory are usually set to `rwx--x--x` or `rwxr-xr-x`.

2.5.6 Changing default access permissions

You will need to understand how to set file access permissions numerically before reading this section.

Every time you create a file or a directory, its access permissions are set to a predetermined value. The file-creation mode mask defines this value.

To display the value of this mask enter the command:

umask

This will produce something like:

```
077
```

Sometimes the 0 (zero) is omitted. The values might be different on your system.

Without the mask the system would set permissions of 666 for files and 777 for directories when first created. The values in the mask are subtracted from these values to give a default value for access permissions for the files and directories created by you.

For example:

```
  777    (system value for directories)
 -077    (value of the umask)
  ---
  700    (default access permission of
         rwx------)
```

To change your default access permissions you use the command:

umask *nnn*

Each "n" is a number from 0 to 7.

Examples of using the umask command

To give yourself full permissions for both files and directories and prevent the group and other users from having access:

umask 077

This subtracts 077 from the system defaults for files and directories 666 and 777. Giving a default access permissions for your files of 600 `(rw-------)` and for directories of 700 `(rwx------)`.

To give all access permissions to the group and allow other users read and execute permission:

umask 002

This subtracts 002 from the sytem defaults to give default access permission for your files of 664 `(rw-rw-r--)` and for your directories of 775 `(rwxrwxr-x).`

To give the group and other users all access except write access:

umask 022

This subtracts 022 from the system defaults to give a default access permission for your files of 644 `(rw-r--r--)` and for your directories of 755 `(rwxr-xr-x).`

2.5.7 Changing group ownership of files and directories

Every user is a member of one or more groups. To find out which groups you belong to use the command:

groups

To find out which groups another user belongs to use the command:

groups *username*

Your files and directories are owned by the group (or one of the groups) that you belong to. This is known as their "group ownership".

To list the group ownership of your files:

ls -gl

You can change the group ownership of a file or directory with the command:

chgrp *group_name file/directory_name*

You must be a member of the group to which you are changing ownership.

2.5.8 Changing access permissions

To change the access permissions for a file or directory use the command

chmod ***mode filename***
chmod ***mode directory_name***

The "mode" consists of three parts: who the permissions apply to, how the permissions are set and which permissions to set.

Changing access permissions using the chmod command

To give yourself permission to execute a file that you own:

```
chmod u+x file1
```

This gives you execute permission for the file "file1".

To give members of your group permission to read a file:

```
chmod g+r file2
```

This gives the group permission to read the file "file2".

To give read permission to everyone for a particular type of file:

```
chmod a+r *.pub
```

This gives everyone permission to read all files with the extension `.pub`.

To give the group write and execute permission:

```
chmod g+wx $HOME/SCCS
```

This gives all members of the group permission to place files in the directory SCCS in your home directory. They can also list (`ls`) the contents of this directory.

2.5.9 Copying files

2.5.9.1 Copying files in the same directory

To create an exact copy of a file use the cp (copy) command.

cp [-option] ***source destination***

The source is the name of the file to be copied; the destination is the name of the file in which the copy is to be placed.

Examples of using the cp command

To copy a single file in the current directory:

```
cp notes sect3.txt
```

This makes a copy of the file `notes` in the file `sect3.txt`.

To copy a file using the -i option:

```
cp -i notes part2.txt
```

This makes a copy of the file `notes` in the file `part2.txt`. If this file exists, the message

```
part2.txt: File exists
```

is displayed. The file is not copied.

2.5.9.2 Copying more than one file

You can use special "wildcard" characters whenever you want to copy several files that have similar filenames.

Instead of entering the **cp** command followed by several filenames you can use a single filename that contains one or more wildcards.

Examples of using cp with special characters

To copy files that match on several characters:

```
cp *.txt chapt1
```

This copies all the files in the current working directory with the extension ".txt" to the sub-directory `chapt1`.

To copy files that match on a single character:

```
cp sect?b partA
```

This copies any file with the name `sect[1-9]b` to the sub-directory `partA`.

2.5.9.3 Copying files to another directory

To copy a file to another directory from your current directory give name of the source file followed by the pathname to the destination file.

cp ***source path_to_destination***

For the destination file to have the same name as the source file use:

cp ***source path_to_destination_directory***

2.5.9.4 Copying files from another directory

To copy a file from another directory to your current directory give the pathname to the source file followed by the name of the destination file.

cp ***path_to_source_file destination***

For the destination file to have the same name as the source file use:

cp ***path_to_source_file* .**

The . (dot) is shorthand for the current working directory.

2.5.10 Creating files

2.5.10.1 Create a file with the cat command

Type the command

cat > ***name_of_file***

Now type in your text. Press the `<Return>` key to start a new line.

When you have finished typing in your text, enter **Ctrl-d** (Press and hold down the Ctrl key and type a "d").

This stops the **cat** command and returns you to the system prompt.

Examples of creating files

To create a file with the **cat** command:

```
cat > memo
remember to make appointment at opticians
get cost of scheduled flight to Athens Ctrl-d
```

This creates a file called `memo` in the current directory containing two lines of text.

2.9.10.2 Create a file with the echo command

To create a file, type the command **echo** and then continue to type the text that you want to appear in the file.

When you have finished typing in your text, type

```
> name_of_file
```

and press the <Return> key.

This method is useful for creating short files containing only one or two lines of text.

Example

To create a file with the **echo** command:

```
echo use an editor for large files > tips
```

This creates a file called `tips` in the current directory containing a single line of text.

2.9.10.3 Removing files and directories

To remove a file use the command:

rm [option] ***filename***

To remove a directory use the command:

rmdir ***directory_name***

or

rm -r ***directory_name***

You cannot retrieve a file or directory that has been removed; it has been removed permanently.

Removing files

To remove a file use the command:

rm ***filename(s)***

You cannot remove a file in another user's account unless they have set access permissions for the file which allow you to.

Use the `-i` (interactive) option which makes the command prompt you for confirmation that you want to remove each file.

Examples of removing files with the rm command

To remove a single file:

rm help.txt

This removes the file `help.txt` from the current directory.

To remove several files:

```
rm artwork/figure[2-7]
```

This removes files `figure2` through to `figure7` from the subdirectory `artwork`.

2.5.10.4 Removing directories

To remove a directory use the command:

rmdir *directory_name*

The directory *must* be empty before you can delete it. You will need to remove any files and subdirectories that it contains.

To remove a directory that contains files use the command:

rm -r *directory_name*

This deletes all the contents of the directory including any subdirectories.

Examples of removing directories

To remove an empty directory:

```
rmdir docs
```

This removes the empty directory `docs` from the current directory.

To remove a directory containing files and subdirectories:

```
rm -r projectX
```

This deletes the directory `projectX` and any files or subdirectories that it holds.

To remove a directory interactively:

```
rm -ir plan9
```

This will prompt you for confirmation before removing each subdirectory, any files and the directory `plan9` itself.

2.9.10.5 Determining file type

The **file** command examines the content of a file and reports what type of file it is.

To use the command enter:

file ***filename***

Use this command to check the identity of a file, or to find out if executable files contain shell scripts, or are binaries. Shell scripts are text files and can be displayed and edited.

Examples of using the file command

To check the identity of a single file:

```
file ReportA
ReportA:        Frame Maker document
```

This checks the identity of the file `ReportA` and reports that this file has been created using the Frame Maker application.

To check the identity of several files:

```
file *
external.c:    c program text
external.h:     c program text
mc:             sparc demand paged dynamically linked executable
mcXThelp:       executable shell script
mcdemo:         executable shell script
mchelp:         executable shell script
mchome:         executable shell script
mctools:        executable shell script
```

This checks the identity of every file in the current directory. Notice how it reports whether a file is an executable binary or a shell script.

2.5.11 **Displaying files**

2.5.11.1 Viewing a file

You can look at a file without the risk of altering its contents by using **vi** in "read only" mode. To do this, use the command:

view ***filename***

The message `[Read only]` is displayed at the bottom of the screen.

You then use the same commands as you would in **vi** to move around, search for text and quit view.

2.5.11.2 Displaying files with the cat command

The **cat** command is useful for displaying short files of a few lines. To display longer files use an editor or pager.

To display the contents of a file use the command:

cat ***filename***

Example

To display a file:

```
cat README
```

This displays the contents of the file `README`. If the file contains more lines than can be displayed on the screen at once it will scroll by; use a pager instead.

2.5.11.3 Finding a file

To locate a file in the file system , use the **find** command.

find ***pathname*** **-name** ***filename*** **-print**

The *pathname* defines the directory to start from. Each subdirectory of this directory will be searched.

The `-print` option *must* be used to display results.

You can define the filename using wildcards. If these are used, the filename must be placed in 'quotes'.

2.5.11.5 Printing files

To print a file use the command:

lpr ***filename***

The job is placed in the print queue and you can carry on working.

Printing multiple copies

To print multiple copies of a file use the # (hash) option. For example:

```
lpr -#3 memo.txt
```

This prints three copies of the file memo.txt on your default printer.

You can also print multiple copies of several files. For example:

```
lpr -Pps1 -#3 survey.doc letter.txt
```

This prints three copies of the files `survey.doc` and `letter.txt` on the printer `ps1`.

2.5.11.6 Controlling access to your files and directories

Every file and directory in your account can be protected from or made accessible to other users by changing its access permissions.

You can only change the permissions for files and directories that you own. We are already discussed about file permissions.

2.5.11.7 Searching the contents of a file

To search a text file for a string of characters or a regular expression use the command:

grep ***pattern filename(s)***

Using this command you can check to see if a text file holds specific information. **grep** is often used to search the output from a command.

Any regular expression containing one or more special characters must be quoted to remove their meaning.

Examples of using the grep command

To search a file for a simple text string:

```
grep copying help
```

This searches the file `help` for the string `copying` and displays each line on your terminal.

To search a file using regular expression:

```
grep -n '[dD]on\'t' tasks
```

This uses a regular expression to find and display each line in the file `tasks` that contains the pattern `don't` or `Don't`. The line number for each line is also displayed.

The expression is quoted to prevent the shell expanding the metacharacters `[`, `]` and `'`. Double quotes are used to quote the single quote in dDon't.

To use the output of another command as input to the grep command:

```
ls -l | grep ' d........x'
```

This lists all the directories in the current directory for which other users have executed permission.

The expression is quoted to prevent the shell interpreting the metacharacter.

To redirect the results of a search to a file:

```
grep Smith /etc/passwd > smurffs
```

This searches the `passwd` file for each occurrence of the name `Smith` and places the results of this search in the file `smurffs`. There being a lot of Smiths everywhere this is quite a large file.

3

VI EDITOR

The vi editor (pronounced "vee eye") is available on all UNIX systems: other editors are not. Being able to use vi ensures that you will always have an editor available to you.

3.1. Operating Modes

There are two modes in which you use vi.

Command mode

This is the mode you are in whenever you begin to use vi. In this mode commands are used to move around and edit text objects such as words, sentences and paragraphs.

Pressing the ESC key returns you to command mode.

Insert mode

This is the mode you use to type (insert) text into a buffer. There are several commands that you can use to enter this mode.

3.2 About vi commands

While not entering text you use vi in command mode and enter editing commands consisting of one or more characters.

The syntax of an editing command is:

operator object

The operator is a character - such as d for delete - that describes the action to be carried out on the object; a word, sentence, paragraph or section.

To return to command mode from insert mode press the ESC key. If you are already in command mode the terminal will beep at you (or make some similar sound).

3.3 UNIX shell commands

You can run UNIX commands and see their output without leaving vi. You can also insert the output of a UNIX command into the file you that are editing.

To run a single UNIX command use the command:

:!***UNIX_command***

You can start a shell from within vi and use it as you would your usual UNIX environment, then exit the shell and return to vi.

To start up a shell enter the command:

:sh

The type of shell that is started is determined by the $SHELL variable. You can specify that some other shell is to be started by setting the vi shell option.

Return to using vi by entering the command exit or Ctrl-D.

To insert the output from a UNIX command into a file, immediately after the cursor:

:r!***command***

For example, this facility would be very useful if you were using vi to document a UNIX command and you wanted to include examples of the output from this command.

3.4 Starting vi and opening files

To start using vi enter the command:

vi ***filename***

The filename can be the name of an existing file or the name of the file that you want to create.

Examples

To open an existing file for editing:

```
vi intro
```

This opens the file intro for editing. This file already exists in the current directory.

To create a file to edit:

```
vi part2
```

This command starts vi with a new file named part2. This file does not exist in the current directory until it is written to using the write command.

3.5 Saving files and exiting vi

There are several ways in which you can end your editing session, leave vi and return to the shell.

- Saving changes

To quit vi and save the contents of the buffer to the file that vi is being used to edit:

```
ZZ
```

or

```
:wq
```

- Without saving changes

To quit vi without saving the contents of the buffer to the file:

```
:q!
```

- Saving to another file

To quit vi and save the contents of the buffer to a new file:

```
:w filename
:q
```

- Quitting vi

To quit the vi editor:

```
:q
```

If you have edited (changed) the file you will be prompted with the message:

```
No write since last change
```

You can either save the changes or quit without saving any of the changes you have made.

3.6 Viewing a file

You can look at a file without the risk of altering its contents by using vi in "read only" mode. To do this, use the command:

```
view filename
```

The message [Read only] is displayed at the bottom of the screen.

You then use the same commands as you would in vi to move around, search for text and quit view.

3.7 Moving around the text

There are a range of easy to use commands that provide a great degree of flexibility in how you move around the contents of the file you are editing.

Many of these commands can be combined with a numerical value to extend their functionality.

- Move the cursor along a line

Press the ESC key to enter command mode before using these commands.

To move to	*Do this ...*

next character	l
previous character	h
next word	w
next *n* words	w*n*
previous word	b
previous *n* words	b*n*
end of current word	e
start of current line	0 (zero)
end of current line	$

- Move the cursor between lines

Press the ESC key to enter command mode before using these commands.

To move to	*Do this ...*
next line down (same column)	j

start of next line down	+
previous line (same column)	k
start of previous line	-

- Move between text blocks

Press the ESC key to enter command mode before using these commands.

To move to	*Do this ...*
beginning of next sentence	)
beginning of previous sentence	(
beginning of next paragraph	}
beginning of previous paragraph	{

A full stop followed by two or more spaces is treated as the end of a sentence.

All text to the next empty line is treated as belonging to the current paragraph.

- Move over the file

Press the ESC key to enter command mode before using these commands.

To move to	*Do this ...*
top of the file	1G
bottom of the file	G

3.8 Entering text

To type text into the buffer you must enter insert mode. Choose the command that is best suited to the present position of the cursor and the editing task you want to do.

As the text you are typing in begins to approach the right-hand side of the screen press <RETURN> to start a new line.

- Tip ...

As you are entering text, make sure that each sentence ends with a fullstop followed by two or more spaces. You can then be sure of using vi's editing commands to manipulate the sentences in your text.

If necessary use vi's global replace command to change sentence endings that do not meet this convention.

- List of commands

List of insert commands

To do this ...	*Command*
Insert text after the cursor	a
Insert text before the cursor	i
Append text at the end of the current line	A
Insert text at the start of the current line	I
Open a new line below the current line	o
Open a new line above the current line	O

3.9 Changing text around

There are several ways in which you can change the structure and the content of the file you are editing.

- Repeat parts of the text

You can re-use a piece of text in another part of the file by yanking it into a temporary buffer and then putting the contents of this buffer back into the file.

The command y yanks the text into the temporary buffer.

Combined with other commands that are used to move around text objects, the yank command can be used to pull any part of a file into the buffer.

The command p pastes the contents of the temporary buffer back into the file immediately after the cursor.

- Insert the contents of another file

To insert the contents of another file use the command:

```
:r filename
```

This inserts the file immediately after the current line.

- Cut and paste text

Moving pieces of text around within a file is referred to as cut and paste. Doing this in vi is a three stage process:

1. Cut the text you want to move with one of the commands used to delete text.

2. Move the cursor to where you want to paste the text into the file using one of the commands for moving around the text.

3. Paste the text into the file with the command:

```
p
```

3.10 Moving around the text

There are a range of easy to use commands that provide a great degree of flexibility in how you move around the contents of the file you are editing.

Many of these commands can be combined with a numerical value to extend their functionality.

- Move the cursor along a line

Press the ESC key to enter command mode before using these commands.

To move to	*Do this ...*
next character	l
previous character	h
next word	w
next *n* words	w*n*
previous word	b
previous *n* words	b*n*
end of current word	e
start of current line	0 (zero)
end of current line	$

- Move the cursor between lines

Press the ESC key to enter command mode before using these commands.

To move to	*Do this ...*
next line down (same column)	j
start of next line down	+
previous line (same column)	k
start of previous line	-

- Move between text blocks

Press the ESC key to enter command mode before using these commands.

To move to	*Do this ...*
beginning of next sentence	)
beginning of previous sentence	(
beginning of next paragraph	}

beginning of previous paragraph	{

A full stop followed by two or more spaces is treated as the end of a sentence.

All text to the next empty line is treated as belonging to the current paragraph.

- Move over the file

Press the ESC key to enter command mode before using these commands.

To move to	*Do this ...*
top of the file	1G
bottom of the file	G

3.11 Deleting text

Text is deleted by combining the delete commands x and d with the type of text object to be deleted.

Numbers can also used with both the x and d commands to delete multiple instances of a text object. For example:

```
5dw
```

This deletes the current word and the following four words.

- Deleting characters

Press the ESC key to enter command mode before using these commands.

To delete	*Do this*
current character	x
previous character	dh

- Deleting words and lines

Press the ESC key to enter command mode before using these commands.

To delete	*Do this*
current word	dw
previous word	db
entire line	dd
to end of line	d$
to start of line	d0 (zero)

next *n* lines	*n*dd

- Deleting sentences and paragraphs

Press the ESC key to enter command mode before using these commands.

To delete	*Do this*
to end of sentence	d)
to beginning of sentence	d(
to end of paragraph	d}
to beginning of paragraph	d{

- Undeleting text

To get back a word or line that you have just deleted enter the command:

```
p
```

3.12 Searching for text

Press the ESC key to enter command mode before using these commands.

To search	*Do this ...*
forward for a pattern	/*pattern* <RETURN>
backward for a pattern	?*pattern* <RETURN>
repeat previous search	n
repeat previous search in reverse direction	N

3.13 Replacing text

Press the ESC key to enter command mode before using these commands.

To replace	Do this ...
pattern1 with *pattern2* on the same line	:s/*pattern1*/*pattern2*
every occurrence of *pattern1* with *pattern2*	:g/*pattern1*/s//*pattern2*/g

4

SHELL PROGRAMMING

4.1 What is a shell?

A shell is a program, which provides a user interface. With a shell, users can type in commands and run programs on a Unix system. Basically, the main function a shell performs is to read in from the terminal what one types, run the commands, and show the output of the commands. The main use of the shell scripts is as an interactive shell, but one can write programs using the C shell. These programs are called **shell scripts**. In other words

- A text file containing commands which could have been typed directly into the shell.
- The shell itself has limited capabilities -- the power comes from using it as a "glue" language to combine the standard Unix utilities, and custom software, to produce a tool more useful than the component parts alone.
- Any shell can be used for writing a shell script. To allow for this, the first line of every script is:

`#!/path/to/shell` (e.g. `#!/bin/ksh`).

- Any file can be used as input to a shell by using the syntax: `ksh myscript`
- If the file is made executable using `chmod`, it becomes a new command and available for use (subject to the usual $PATH search). `chmod +x myscript`

A shell script can be as simple as a sequence of commands that you type regularly. By putting them into a script, you reduce them to a single command.

Example:

```
#!/bin/sh
 date
 pwd
 du -k
```

4.2 Why use Shell Scripts

- Combine lengthy and repetitive sequences of commands into a single, simple command.
- Generalize a sequence of operations on one set of data, into a procedure that can be applied to any similar set of data. Create new commands using combinations of utilities in ways the original authors never thought of.
- Simple shell scripts might be written as shell aliases, but the script can be made available to all users and all processes. Shell aliases apply only to the current shell.
- Wrap programs over which you have no control inside an environment that you can control.
- Create customized datasets on the fly, and call applications (e.g. matlab, sas, idl, gnuplot) to work on them, or create customized application commands/procedures.
- Rapid prototyping (but avoid letting prototypes become production)

4.3 Typical uses of shell scripts

- System boot scripts (/etc/init.d)
- System administrators, for automating many aspects of computer maintenance, user account creation etc.
- Application package installation tools
- Application startup scripts, especially unattended applications (e.g. started from `cron` or `at`)

- Any user needing to automate the process of setting up and running commercial applications, or their own code.

4.4 History of Shells

`sh`

aka "Bourne" shell, written by Steve Bourne at AT&T Bell Labs for Unix V7 (1979). Small, simple, and (originally) *very few internal commands*, so it called external programs for even the simplest of tasks. It is always available on everything that looks vaguely like Unix.

`csh`

The "C" shell, Written by Bill Joy at Berkeley (who went on to found Sun Microsystems). Many things in common with the Bourne shell, but many enhancements to improve interactive use. The internal commands used only in scripts are very different from "sh", and similar (by design) to the "C" language syntax.

`tcsh`

The "TC" shell. Freely available and based on "csh". It has many additional features to make interactive use more convenient.

`ksh`

The "Korn" shell, written by David Korn of AT&T Bell Labs (now Lucent). Written as a major upgrade to "sh" and backwards compatible with it, but has many internal commands for the most frequently used functions. It also incorporates many of the features from tcsh, which enhance interactive use (command line history recall etc.).

4.5 POSIX 1003.2 Shell Standard.

Standards committees worked over the Bourne shell and added many features of the Korn shell (ksh88) and C shell to define a standard set of features that all compliant shells must have.

`bash`

The "Bourne again" shell. Written as part of the GNU/Linux Open Source effort, and the default shell for Linux and Mac OS-X. It is a functional clone of sh, with additional features to enhance interactive use, add POSIX compliance, and partial ksh compatability.

`zsh`

A freeware functional clone of sh, with parts of ksh, bash and full POSIX compliance, and many new interactive command-line editing features. It was installed as the default shell on early MacOSX systems.

4.6 Features of Shell programming

Some of the features of the shell programming or scripts are listed here:

1. Customizable environment.
2. Abbreviate commands. (Aliases.)
3. History. (Remembers commands typed before.)
4. Job control. (Run programs in the background or foreground.)
5. Shell scripting. (One can write programs using the shell.)
6. Keyboard shortcuts.

4.6.1 Features of the shell environment

The shell provides programming features listed below:

1. Control constructs. (For example, loop and conditional statements.)
2. File permissions/existence checking.
3. Variable assignment.
4. Built-in Variables.

4.6.2 Files for the Shell environment customization

The shell has three separate files that are used for customizing its environment. These three files are. cshrc**,. login, and .*logout*.** Because these files begin with a period (.) they do not usually appear when one types the **ls** command. In order to see all files beginning with periods, the **-a** option is used with the **ls** command.

The. cshrc file contains commands, variable definitions and aliases used any time the shell is run. When one logs in, the shell starts by reading the *.cshrc* file, and sets up any variables and aliases.

The shell reads the ***.login*** file after it has read the *.cshrc* file. This file is read once only for login shells. This file should be used to set up terminal settings, for example, backspace, suspend, and interrupt characters.

The ***.logout*** file contains commands that are run when the user logs out of the system.

Sample ***.cshrc*** file

```
#!/bin/csh
# Sample .cshrc file
setenv EXINIT 'set smd sw=4 wm=2'
set history=50
set savehist=50
set ignoreeof noclobber
if ($?prompt) then
  set prompt='[\!]% '
  alias f finger -R
  alias lo logout
endif
```

Sample *.login* file

```
#!/bin/csh
# Sample .login file
stty erase ^H intr ^C susp ^Z
echo "Welcome to Wiliki\!"
frm -s n
```

Sample *.logout* file

```
#!/bin/csh
# Sample .logout file
echo -n "Logged out of Wiliki "
date
```

4.6.3 Principal Differences

Between sh (+derivitives), and csh (+derivitives).

- Syntax of all the flow control constructs and conditional tests.
- Syntax for string manipulation inside of scripts
- Syntax for arithmetic manipulation inside of scripts
- Syntax for setting local variables (used only in the script) and environment variables (which are passed to child processes). `setenv` vs `export`
- Syntax for redirecting I/O streams other than stdin/stdout
- Login startup files (`.cshrc` and `.login`, vs `.profile`) and default options
- Reading other shell scripts into the current shell (`source filename`, vs `. filename`)
- Handling of signals (interrupts)

4.6.4 Other Scripting Languages

There are many other programs that read a file of commands and carry out a sequence of actions. The `"#!/path/to/program"` convention allows any of them to be used as a scripting language to create new commands. Some are highly specialized, and some are much more efficient than the equivalent shell scripts at certain tasks. There is never only one way to perform a function, and often the choice comes down to factors like:

- what is installed already - many other scripting languages are not available by default
- what similar code already exists
- what you are most familiar with and can use most efficiently. Your time is usually more expensive than computer cycles.

Some major players (all of these are freely available) in the general purpose scripting languages are:

- `awk`
- `perl`
- `python`
- `tcl/tk`

ksh/bash vs sh

Ksh and bash are both supersets of sh. For maximum portability, even to very old computers, you should stick to the commands found in sh. Where possible, ksh or bash-specific features will be noted in the following pages. In general, the newer shells run a little faster and scripts are often more readable because logic can be expressed cleanlier user the newer syntax. Many commands and conditional tests are now internal.

If you follow textbooks on Bourne shell programming, all of the advice should apply no matter which of the Bourne-derived shells you use. Unfortunately, many vendors have added features over the years and achieving complete portability can be a challenge.

Explicitly writing for ksh (or bash) and insisting on that shell being installed can often is simpler.

The sh and ksh *man* pages use the term *special command* for the internal commands - handled by the shell itself.

4.7 Basic shell script syntax

The most basic shell script is a list of commands exactly as could be typed interactively, prefaced by the `#!` magic header. All the parsing rules, filename wildcards, $PATH searches etc., which were summarized above, apply. In addition: `#` as the first non-whitespace character on a line flags the line as a comment, and the rest of the line is completely ignored. Use comments liberally in your scripts, as in all other forms of programming. `\` as the last character on a line causes the following line to be logically joined before interpretation. This allows single very long commands to be entered in the script in a more readable fashion. You can continue the line as many times as needed.
`;` as a separator between words on a line is interpreted as a newline. It allows you to put multiple commands on a single line. There are few occasions when you must do this, but often it is used to improve the layout of compound commands.
Example:

```
#!/bin/ksh
# For the purposes of display, parts of the script have
# been rendered in glorious technicolor.
## Some comments are bold to flag special sections

# Line numbers on the left are not part of the script.
# They are just added to the HTML for reference.

# Built-in commands and keywords (e.g. print) are in blue
# Command substitutions are purple. Variables are black
print "Disk usage summary for $USER on `date`"

# Everything else is red - mostly that is external
# commands, and the arguments to all of the commands.
print These are my files       # end of line comment for print
# List the files in columns
ls -C
# Summarize the disk usage
print
print Disk space usage
du -k
exit 0
```

Exit status

Every command (program) has a *value* or *exit status*, which it returns to the calling program. This is separate from any output generated. The exit status of a shell script can be explicitly set using `exit N`, or it defaults to the value of the last command run.

4.7.1 Special characters in Shell

Some characters are special to the shell, and in order to enter them, one has to precede it with a backslash (\). Some are listed here with their meaning to the shell.

!

History substitution.

< >

Output redirection.

|

Pipes.

*

Matches any string of zero or more characters.

?

Matches any single character.

[ ]

Matches any set of characters contained in brackets.

{ }

Matches any comma-separated list of words.

;

Used to separate commands.

&

Also used to separate commands, but puts them in the background.

\

Quote the following character.

$

Obtains the value of the variable.

'

Take text enclosed within quotes literally.

`

Take text enclosed within quotes as a command, and replace with output.

"

Take text enclosed within quotes literally, after substituting any variables.

4.8 Variables

Variables in C shell are defined using the internal **set** command. C shell supports both regular and array variables. Some examples are given below:

set var1=a3 #sets var1's value to a3.

set var2=(a b c)

# sets the array variable var2 to a b, and c.

4.8.1 Using variables

Variables can be used in C shell by typing a dollar sign ($) before the variable name. If the variable is an array, the subscript can be specified using brackets, and the number of elements can be obtained using the form `$#var2`.

The existence of variables can be checked using the form `$?variable`. If the variable exists, the expression evaluates to a one (true), otherwise, it evaluates to a zero (false). Simple integer calculations can be performed by C shell, using C language-type operators. To assign a calculated value, the **@** command is used as follows:

@ var = $a + $x * $z

4.8.2 Built-in shell variables

Certain variables control the behavior of the C shell, and some of these don't require a value. (I.e., can be set simply by using **set** command by itself without any value.) The **unset** command can be used to unset any undesirable variables.

`argv`

Special variable used in shell scripts to hold the value of arguments.

`autologout`

Contains the number of minutes the shell can be idle before it automatically logs out.

`history`

Sets how many lines of history (previous commands) to remember.

`ignoreeof`

Prevents logging out with a *control-D*.

`noclobber`

Prevents overwriting of files when using redirection.

`path`

Contains a list of directories to be searched when running programs or shell scripts.

`prompt`

Sets the prompt string.

`term`

Contains the current terminal type.

History

If the `history` variable is set to a numerical value, that many commands typed previous would be remembered in a history list. Commands from the history are numbered from the first command being 1. To see the history, the **history** command is used.

Commands from the history can be recalled using the exclamation point. For example, `!!` repeats the previous command, `!25` re-types command number 25 from the history, and `!-2` re-types the second line previous to the current line.

Individual words from these command lines can also be retrieved using this history. For example, `!25:$` returns the last argument (word) from command 25, `!!:*` returns all the arguments (all words but the first one) from the last command, and `!-2:0` returns the command (the first word) of the second line previous.

Aliasing

A shorthand can be assigned to a command or sequence of commands which are frequently used. By assigning an alias with the **alias** command, one can essentially create their own commands, and even "overwrite" existing commands. For example:

alias cc cc -Aa -D_HPUX_SOURCE

This alias definition will substitute the **cc** with the ANSI compiler option on an HP System (such as *Wiliki*) whenever `cc` is typed. To undefine an alias, the **unalias** command is used.

If the filenames used behind an alias must come before text being substituted, history substitution can be used, as follows:

alias manl 'man \!* | less -p'

This form of the command places the arguments placed after the `manl` alias between the **man** command and the | (pipe).

4.8.3 Conditional modifiers

There are various ways to conditionally use a variable in a command.

`${datafile-default}`

> Substitute the value of `$datafile`, if it has been defined, otherwise use the string "default". This is an easy way to allow for optional variables, and have sensible defaults if they haven't been set. If `datafile` was undefined, it remains so.

`${datafile=default}`

> Similar to the above, except if `datafile` has not been defined, set it to the string "default".

`${datafile+default}`

> If variable `datafile` has been defined, use the string "default", otherwise use null. In this case the actual value `$datafile` is not used.

`${datafile?"error message"}`

Substitute the value of `$datafile`, if it has been defined, otherwise display `datafile: error message`. This is used for diagnostics when a variable should have been set and there is no sensible default value to use.

4.9 Input/Output Redirection

The input and output of commands can be sent to or gotten from files using redirection. Some examples are shown below:

date > datefile

The output of the **date** command is saved into the contents of the file, *datefile*.

a.out < inputfile

The program, **a.out** receives its input from the input file, *inputfile*.

sort gradefile >> datafile

The **sort** command returns its output and appends it to the file, *datafile*.

A special form of redirection is used in shell scripts.

```
calculate << END_OF_FILE
...
...
END_OF_FILE
```

In this form, the input is taken from the current file (usually the shell script file) until the string following the "<<" is found.

If the special variable, `noclobber` is set, if any redirection operation will overwrite an existing file, an error message is given and the redirection will fail. In order to force an overwrite of an existing file using redirection, append an exclamation point (!) after the redirection command. For example for the command:

date >! datefile

The file *datefile* will be overwritten regardless of its existence.

Adding an ampersand (&) to the end of an output redirection command will combine both the standard error and the standard output and place the output into the specified file.

4.9.1 I/O redirection and pipelines

Any simple command (or shell function, or compound command) may have its input and output redirected using the following operators.

4.9.2 Output redirection

`>` *`filename`*

Standard ouput (file descriptor 1) is redirected to the named file. The file is overwritten unless the `noclobber` option is set. The file is created if it does not exist.

`>>` *`filename`*

Standard ouput is appended to the named file. The file is created if it does not exist.

`>|` *`filename`*

Output redirect, and override the *noclobber* option, if set.

4.9.3 Input redirection

`<` *`filename`*

Standard input (file descriptor 0) is redirected to the named file. The file must already exist.

4.9.4 Command pipelines

`command | command [ | command ...]`

Pipe multiple commands together. The standard output of the first command becomes the standard input of the second command. All commands run simultaneously, and data transfer happens via memory buffers. This is one of the most powerful constructs in Unix. *Compound* commands may also be used with pipes.

4.9.5 Input and Output

Shell scripts can generate output directly or read input into variables using the following commands:

4.9.6 Script output

`echo`

Print arguments, separated by spaces, and terminated by a newline, to stdout. Use quotes to preserve spacing. Echo also understands C-like escape conventions.

-n

suppress newline

`print` (ksh internal)

Print arguments, separated by spaces, and terminated by a newline, to stdout. Print observes the same escape conventions as echo.

-n

suppress newline

-r

raw mode - ignore \-escape conventions

-R

raw mode - ignore \-escape conventions and -options except -n.

4.9.7 Script input

`read var1 var2 rest`

read a line from stdin, parsing by $IFS, and placing the words into the named variables. Any left over words all go into the last variable. A '\' as the last character on a line removes significance of the newline, and input continues with the following line.

-r

raw mode - ignore \-escape conventions

Example:

#!/bin/sh

```
echo "Testing interactive user input: enter some keystrokes and press return""
read x more
echo "First word was \"$x\""
echo "Rest of the line (if any) was \"$more\""
```

4.10 Conditional tests for [...] and [[...]] commands

Most of the useful flow-control operators involve making some conditional test and branching on the result (true/false). The test can be either the `test` command, or its alias, `[`, or the ksh/bash built-in `[[ ... ]]` command, which has slightly different options, or it can be *any command which returns a suitable exit status*. Zero is taken to be "True", while any non-zero value is "False". Note that this is backwards from the C language convention.

4.10.1 File tests

`-e` *`file`*

True if *file* exists (can be of any type).

`-f` *`file`*

True if *file* exists and is an ordinary file.

`-d` *`file`*

True if *file* exists and is a directory.

`-r` *`file`*

True if *file* exists and is readable

Similarly, `-w` = writable, `-x` = executable, `-L` = is a symlink.

`-s` *`file`*

True if *file* exists and has size greater than zero

`-t` *`filedescriptor`*

True if the open *filedescriptor* is associated with a terminal device. E.g. this is used to determine if standard output has been redirected to a file.

4.10.2 Character string tests

`-n "string"`

true if *string* has non-zero length

`-z "string"`

true if *string* has zero length

`$variable = `*`text`*

True if *$variable* matches *text*.

`$variable < `*`text`*

True if *$variable* comes before (lexically) *text*
Similarly, `>` = comes after

4.10.3 Arithmetic tests

`$variable -eq `*`number`*

True if *$variable*, interpreted as a number, is equal to *number*.

`$variable -ne `*`number`*

True if *$variable*, interpreted as a number, is not equal to *number*. Similarly, `-lt` = less than, `-le` = less than or equal, `-gt` = greater than, `-ge` = greater than or equal

4.10.4 Negating and Combining tests

Tests may be negated by pretending the `!` Operator, and combined with boolean AND and OR operators using the syntax:

`Conditional`` -a `*`conditional`*`, `*`conditional`*` -o `*`conditional`* AND and OR syntax for `test` and `[`

`conditional`` && `*`conditional`*`, `*`conditional`*` || `*`conditional`* AND and OR syntax for `[[ ... ]]`

Parentheses may be inserted to resolve ambiguities or override the default operator precedence rules.

Examples:

```
if [[  -x /usr/local/bin/lserve && \
       -w /var/logs/lserve.log ]]; then
   /usr/local/bin/lserve >> /var/logs/lserve.log &
fi
```

```
pwent=`grep '^richard:' /etc/passwd`
if [ -z "$pwent" ]; then
   echo richard not found
fi
```

4.10.5 Flow Control and Compound Commands

A *list* in these descriptions is a simple command, or a pipeline. The value of the *list* is the value of the last simple command run in it.

4.10.6 Conditional execution: if/else

`list && list`

Execute the first *list*. If true (success), execute the second one.

`list || list`

Execute the first *list*. If false (failure), execute the second one.

Example:

```
mkdir tempdir && cp workfile tempdir

sshd || echo "sshd failed to start"
```

`if list; then list ; elif list; else list; fi`

Execute the first *list*, and if true (success), execute the "then" list, otherwise execute the "else" list. The "elif" and "else" lists are optional.

Example:

```
if [ -r $myfile ]
then
  cat $myfile
else
  echo $myfile not readable
fi
```

4.10.7 Looping: 'while' and 'for' loops

`while` *`list`*`; do `*`list`*`; done`
`until` *`list`*`; do `*`list`*`; done`

Execute the first *list* and if true (success), execute the second *list*. Repeat as long as the first *list* is true. The `until` form just negates the test.

Example:

```
#!/bin/ksh
count=0
max=10
while [[ $count -lt $max ]]
do
  echo $count
  count=$((count + 1))
done
```

`for` *`identifier`*` [ in `*`words`*` ]; do; `*`list`*`; done`

Set *identifier* in turn to each word in *words* and execute the *list*. Omitting the "in *words*" clause implies using $@, i.e. the *identifier* is set in turn to each positional argument.

Example:

```
for file in *.dat
```

```
    do
       echo Processing $file
    done
```

As with most programming languages, there are often several ways to express the same action. Running a command and then explicitly examining `$?` can be used instead of some of the above.

Compound commands can be thought of as running in an implicit sub shell. They can have I/O redirection independent of the rest of the script. Setting of variables in a real sub shell does not leave them set in the parent script. Setting variables in implicit subshells varies in behavior among shells. Older `sh` could not set variables in an implicit subshell and then use them later, but current `ksh` can do this (mostly).

Example:

```
#!/bin/sh

# Demonstrate reading a file line-by-line, using I/O
# redirection in a compound command
# Also test variable setting inside an implicit subshell.
# Test this under sh and ksh and compare the output.

line="TEST"
save=

if [ -z "$1" ]; then
  echo "Usage: $0 filename"
else
  if [ -r $1 ]; then
    while read line; do
      echo "$line"
      save=$line
    done < $1
  fi
fi
echo "End value of \$line is $line"
echo "End value of \$save is $save"
```

4.10.8 Case statement: pattern matching

`case` *`word`* `in` *`pattern`*`)` *`list`*`;; esac`

Compare *word* with each *pattern*) in turn, and executes the first *list* for which the *word* matches. The *patterns* follow the same rules as for filename wildcards.

Example:

```
case $filename in
*.dat)
   echo Processing a .dat file
   ;;
*.sas)
   echo Processing a .sas file
   ;;
*)
   # catch anything else that doesn't match patterns
   echo "Don't know how to deal with $filename"
   ;;
esac
```

4.10.9 miscellaneous flow control and sub shells

`break [n]`

Break out of the current (or n'th) enclosing loop. Control jumps to the next statement after the loop

`continue [n];`

Resume iteration of the current (or n'th) enclosing loop. Control jumps to the top of the loop, which generally causes re-evaluation of a `while` or processing the next element of a `for`.

`. filename`

Read the contents of the named file into the current shell and execute as if in line. Uses $PATH to locate the file, and can be passed positional parameters. This is often used to read in shell functions that are common to multiple scripts. There are security implications if the pathname is not fully specified.

`( ... )` Command grouping

Commands grouped in "( )" are executed in a subshell, with a separate environment (can not affect the variables in the rest of the script).

4.10.10 Conditional Test Examples

As with most aspects of shell scripting, there are usually several possible ways to accomplish a task. Certain idioms show up commonly. These are five ways to examine and branch on the initial character of a string.

Use `case` with a pattern:

```
case $var in
/*) echo "starts with /" ;;
```

Works in all shells, and uses no extra processes

Use `` `cut` ``:

```
if [ "`echo $var | cut -c1`" = "/" ] ; then .
```

Works in all shells, but inefficiently uses a pipe and external process for a trivial task.

Use POSIX variable truncation:

```
if [ "${var%${var#?}}" = "/" ]; then
```

Works with ksh, bash and other POSIX-compliant shells. Not obvious if you have not seen this one before. Fails on old Bourne shells. Dave Taylor in "Wicked Cool Shell Scripts" likes this one.

Use POSIX pattern match inside of [[...]]:

```
if [[ $var = /* ]]; then
```

Works with ksh, bash and other POSIX-compliant shells. Note that you must use [[...]] and no quotes around the pattern.

Use ksh (93 and later) and bash variable substrings:

```
if [ "${var:0:1}" = "/" ]; then
```

ksh93 and later versions, and bash, have a syntax for directly extracting substrings by character position. `${varname:start:length}`

Example:

```
#!/bin/ksh
# Example 17

# Shell idioms: multiple ways to accomplish
# common tasks

# For example, test the first character in a string

var=$1
echo "var=\"$var\""

# Use 'cut' - works everywhere, but uses an an extra process
# for a trivial job.
echo
echo "Using 'cut'"
if [ "`echo $var | cut -c1`" = "/" ] ; then
  echo "starts with /"
else
  echo "doesn't start with /"
fi
```

```
# Use 'case' with a pattern.  Works everywhere.
echo
echo "Using 'case'"
case $var in
/*) echo "starts with /" ;;
*)  echo "doesn't start with /" ;;
esac

# Use POSIX variable truncation - works with ksh and bash and POSIX-compliant sh
# Dave Taylor in "Wicked Cool Shell Scripts" likes this one.
echo
echo "Using POSIX variable truncation"
if [ "${var%${var#?}}" = "/" ]; then
  echo "starts with /"
else
  echo "doesn't start with /"
fi

# Use ksh/bash pattern template match inside of [[ ]]
echo
echo "Using ksh/bash pattern match in [[ ]]"
if [[ $var = /* ]]; then
  echo "starts with /"
else
  echo "doesn't start with /"
fi

# Use ksh (93 and later) and bash variable substrings
echo
echo "Using ksh93/bash variable substrings"
if [ "${var:0:1}" = "/" ]; then
```

```
  echo "starts with /"
else
  echo "doesn't start with /"
fi
```

4.10.11 Shell Functions

All but the earliest versions of `sh` allow you define *shell functions*, which are visible only to the shell script and can be used like any other command. Shell functions take precedence over external commands if the same name is used. Functions execute in the same process as the caller, and must be defined before use (appear earlier in the file). They allow a script to be broken into maintainable chunks, and encourage code reuse between scripts.

Defining functions

`identifier() { list; }`

POSIX syntax for shell functions. Such functions do not restrict scope of variables or signal traps. The identifier follows the rules for variable names, but uses a separate namespace.

`function identifier { list; }`

Ksh and bash optional syntax for defining a function. These functions may define local variables and local signal traps and so can more easily avoid side effects and be reused by multiple scripts.

A function may read or modify any shell variable that exists in the calling script. Such variables are *global*.

(ksh and bash only) Functions may also declare *local* variables in the function using `typeset` or `declare`. Local variables are visible to the current function and any functions called by it.

`return [n], exit [n]`

Return from a function with the given value, or exit the whole script with the given value.

Without a `return`, the function returns when it reaches the end, and the value is the exit status of the last command it ran.

Example:

```
die()
{
  # Print an error message and exit with given status
  # call as: die status "message" ["message" ...]
  exitstat=$1; shift
  for i in "$@"; do
    print -R "$i"
  done
  exit $exitstat
}
```

Calling functions.

Functions are called like any other command. The output may be redirected independantly of the script, and arguments passed to the function. Shell option flags like -x are unset in a function - you must explicitly set them in each function to trace the execution. Shell functions may even be backgrounded and run asynchronously, or run as coprocesses (ksh).

Example:

```
[ -w $filename ] || \
  die 1 "$file not writeable" "check permissions"
```

Example: Backgrounded function call.

```
#!/bin/ksh
```

```
background()
{
  sleep 10
  echo "Background"
  sleep 10
}

echo "ps before background function"
ps
background &
echo "My PID=$$"
echo "Background function PID=$!"
echo "ps after background function"
ps
exit 0
```

Example:

```
vprint()
{
  # Print or not depending on global "$verbosity"
  # Change the verbosity with a single variable.
  # Arg. 1 is the level for this message.
  level=$1; shift
  if [[ $level -le $verbosity ]]; then
    print -R $*
  fi
}
```

verbosity=2

vprint 1 This message will appear

vprint 3 This only appears if verbosity is 3 or higher

Reusable functions

Using only command line arguments, not global variables, and taking care to minimise the side effects of functions can make them made reusable by multiple scripts. Typically they would be placed in a separate file and read with the "`.`" operator.

Functions may generate output to stdout, stderr, or any other file or filehandle. Messages to stdout may be captured by command substitution (`` `myfunction` ``, which provides another way for a function to return information to the calling script. Beware of side-effects (and reducing reusability) in functions which perform I/O.

4.11 Security issues in shell scripts

Shell scripts are often used by system administrators and are run as a privileged user.

- Don't use set-UID scripts.
- Always explicitly set `$PATH` at the start of a script, so that you know exactly which external programs will be used.
- If possible, don't use temporary files. If they cannot be avoided, use `$TMPDIR`, and create files safely (e.g. `mktemp`).
- Check exit status of everything you do.
- Don't trust user input
    - contents of files
    - data piped from other programs

file *names*. Output of filename generation with wildcards, or directly from `ls` or `find`

4.12 When not to use shell scripts

- If an existing tool already does what you need - use it.
- If the script is more than a few hundred lines, you are probably using the wrong tool.
- If performance is horrible, and the script is used a lot, you might want to consider another language.

4.13 A Toolkit of commonly used external commands

The following commands are very frequently used in shell scripts. Many of them are used in the examples in these notes. This is just a brief recap -- see the man pages for details on usage. The most useful are flagged with *.

Most of these commands will operate on a one or more named files, or will operate on a stream of data from standard input if no files are named.

Listing, copying and moving files and directories

`ls` *

list contents of a directory, or list details of files and directories.

`mkdir`; `rmdir` *

Make and Remove directories.

`rm`; `cp`; `mv` *

Remove (delete), Copy and Move (rename) files.

`touch` *

Update the last modifed timestamp on a file, to make it appear to have just been written.

`tee`

Make a duplicate copy of a data stream - used in pipelines to send one copy to a log file and a second copy on to another program. (Think plumbing).

Displaying text, files or parts of files

`echo` *

Echo the arguments to standard output -- used for messages from scripts. Some versions of "sh", and all csh/ksh/bash shells internalized "echo".

`cat` *

Copy and concatenate files; display contents of a file

`head`, `tail` *

Display the beginning of a file, or the end of it.

`cut`

Extract selected fields from each line of a file. Often awk is easier to use, even though it is a more complex program.

Compression and archiving

`compress`; `gzip`, `zip`; `tar` *

Various utilities to compress/uncompress individual files, combine multiple files into a single archive, or do both.

Sorting and searching for patterns

`sort` *

Sort data alphabetically or numerically.

`grep` *

Search a file for lines containing character patterns. The patterns can be simple fixed text, or very complex regular expressions.

`uniq` *

Remove duplicate lines, and generate a count of repeated lines.

`wc` *

Count lines, words and characters in a file.

System information (users, processes, time)

`date` *

Display the current date and time (flexible format). Useful for conditional execution based on time, and for timestamping output.

`ps` *

List the to a running processes.

`kill` *

Send a signal (interrupt) to a running process.

`id`

Print the user name and UID and group of the current user (e.g. to distinguish priviledged users before attempting to run programs which may fail with permission errors)

`who`

Display who is logged on the system, and from where they logged in.

`uname` *

Display information about the system, OS version, hardware architecture etc.

`mail` *

Send mail, from a file or standard input, to named recipients. Since scripts are often used to automate long-running background jobs, sending notification of completion by mail is a common trick.

`logger`

Place a message in the central system logging facility. Scripts can submit messages with all the facilities available to compiled programs.

Conditional tests

`test`; `[` *

The conditional test, used extensively in scripts, is also an external program which evaluates the expression given as an argument and returns true (0) or false (1) exit status. The name "[" is a link to the "test" program, so a line like: `if [ -w logfile ]` actually runs a program "[", with arguments "-w logfile]", and returns a true/false value to the "if" command.

Stream Editing

`awk` *

A pattern matching and data manipulation utility, which has its own scripting language. It also duplicates much functionality from 'sed','grep','cut','wc', etc.

`sed` *

Stream Editor. A flexible editor which operates by applying editing rules to every line in a data stream in turn.

`tr`

Transliterate - perform very simple single-character edits on a file.

Finding and comparing files

`find` *

Search the filesystem and find files matching certain criteria (name pattern, age, owner, size, last modified etc.)

`xargs` *

Apply multiple filename arguments to a named command and run it.

`diff` *

Compare two files and list the differences between them.

`basename` *pathname*

Returns the base filename portion of the named *pathname*, stripping off all the directories

`dirname` *pathname*

Returns the directory portion of the named *pathname*, stripping off the filename

Arithmetic and String Manipulation

`expr` *

The "expr" command takes an numeric or text pattern expression as an argument, evaluates it, and returns a result to stdout. Bourne shell has no built-in arithmetic operators or string manipulation. e.g.

```
expr 2 + 1
expr 2 '*' '(' 21 + 3 ')'
```

Used with text strings, "expr" can match regular expressions and extract sub expressions. Similar functionality can be achived with `sed`. e.g.

```
expr SP99302L.Z00 : '[A-Z0-9]\{4\}\([0-9]\{3\}\)L\.*'
```

`dc`

Desk Calculator - an RPN calculator, using arbitrary precision arithmetic and user-specified bases. Useful for more complex arithmetic expressions than can be performed internally or using `expr`

`bc`

A preprocessor for `dc` which provides infix notation and a C-like syntax for expressions and functions.

Merging files

`paste`

Merge lines from multiple files into tab-delimited columns.

`join`

Perform a join (in the relational database sense) of lines in two sorted input files.

Command line shortcuts

Here are a few keys which may be pressed to perform certain functions.

<escape>

The escape key preceded by a partial command or filename will attempt to complete the filename. If there are more than one filename matching, the common letters are completed, and the C shell beeps.

Control-D

When typed after a partial filename, C shell gives a list of all matching filenames or commands.

Control-W

Erases over the previous word.

Shell scripting

Shell scripts are programs written in C shell. They are plain text files which can be edited and created by any text editor. There are a few guidelines to follow, however.

1. Create a file using any text editor. The first line *must* begin with the string *#!/bin/csh*.
2. Give yourself execute permission with the **chmod u+x** *filename* command.
3. You can run the shell script by simply typing `filename` as if it were a regular command.

The shell script file can contain any commands, which can be typed in, as well as the control structures described above.

Shell script arguments

When you write a shell script, a special array variable `argv` is created with the arguments entered. For example, if the shell script *tester* is created, and is run by typing it with the arguments as shown, `tester one two jump`, the array variable `argv` will contain "one", "two", and "jump" in its three elements.

- Go to the introduction page.
- Go to the UH CoE WWW Server home page.

5

System administration

5.1 System administration Definition

Activities performed by a system administrator (or "admin", "sysadmin", "site admin") such as monitoring security configuration, managing allocation of {user names} and {passwords}, monitoring disk space and other resource use, performing {backups}, and setting up new hardware and software. System administrators often also help users, though in a large organisation this may be a separate job. Compare {postmaster}, {sysop}, {system management}, {systems programmer}.

5.2 The Boot Procedure

Bootstrapping is the process of starting up a computer from a halted or powered-down condition. When the computer is switched on, it activates the memory-resident code which resides on the CPU board. The normal facilities of the operating system are not available at this stage and the computer must 'pull itself up by its own boot-straps' so to speak. This procedure therefore is often referred to as *bootstrapping*, also known as *cold boot*. Although the bootstrap procedure is very hardware dependent, it typically consists of the following steps:

- The memory-resident code
  - Runs self-test.
  - Probes bus for the boot device
  - Reads the boot program from the boot device.
- Boot program reads in the kernel and passes control to it.
- Kernel identifies and configures the devices.
- Initializes the system and starts the system processes.
- Brings up the system in single-user mode (if necessary).
- Runs the appropriate startup scripts.
- Brings up the system for multi-user operation.

Kernel

Most Unix systems implement a two-stage loading process. During the first stage, a small boot program is read into memory from a default or specified device. It is this program that reads in the kernel and relinquishes the control to it. The path to the kernel is vendor-dependent. For example, it is */vmunix* on SunOS 4.x, Digital Unix and Ultrix, */kernel/unix* on SunOS 5.x, */hp-ux* on HP-UX, and */unix* on IRIX and AIX systems.

One of the very first, probably the most difficult, tasks a system administrator must perform, is configuring the kernel. You'll read the details later in the sections dealing with the 'Operating System Installation'. Once the kernel is loaded, it remains in the memory during the running of the system and is usually run in a fixed amount of memory. When the kernel starts, it normally displays its size and the amount of physical memory remaining after the kernel is loaded. The kernel probes the bus to locate the devices specified during the configuration, and initializes the located devices (ignoring those that it can't contact). Any device not detected and initialized during the boot will not be accessible to system until it is properly connected and the system is rebooted.

System Processes

The kernel identifies the root, swap, and dump devices and then starts programs to schedule processes, manage physical memory and virtual memory, and the *init* process. BSD systems starts three initialization processes; *swapper*, *init* and *pagedaemon*. On the SVR4 systems the initialization processes include *sched*, *init*, and various memory handlers (except on Solaris).

sched

The real-time scheduler, *sched*, runs as process 0 on SVR5 systems. It can be used to set priority for real-time processes so that they can be given fast access to the kernel.

swapper

The *swapper daemon* runs as process 0 on BSD systems. It manages the physical memory by moving process from physical memory to swap space when more physical memory is needed.

Page Daemon

Various memory handlers run as process 2. When a page of virtual memory is accessed, the page table within the kernel is consulted and if necessary, the *pagedaemon* (SunOS

4.x) or *pageout* (SunOS 5.x) is used to move pages in and out of physical memory and to update page tables. Similar memory handlers exist on other SVR5 systems.

init

The last step in bootstrapping the kernel starts the */etc/init* process. The init process runs as process 1 and always remains in the background when the system is running. If the system is brought up in a single user mode, init merely creates a shell on the system console (*/dev/console*) and waits for it to terminate before running other startup scripts.

Single User Mode

Single user shell is always Bourne shell (sh) and it runs as *'root'*. It enables the system manager to perform various administrative functions, such as setting the date, checking the consistency of the file system, reconfiguring the list of on-line terminals, and so on. At this stage only the root partition is usually mounted. Other file systems will have to be mounted manually to use programs that do not reside on the root volume. The file system consistency check may be performed by the command fsck, usually found in the */etc* directory.

Startup Scripts

The startup scripts are merely shell scripts, so init spawns a copy of sh to interpret them. The startup scripts are defined and organized differently on different systems. On BSD systems the startup scripts may be found in the */etc* directory and their names begin with rc, e.g., */etc/rc.boot*, */etc/rc.single*, */etc/rc.local* and so on. SVR5 systems define various run levels in which a specific set of processes are allowed to run. This set of processes is defined in the */etc/inittab* file. Each line in the inittab file describes an action to take. The syntax of inittab entries is:

id:run-level:action:process

- id uniquely identifies the entry. It may be a one or characters string.
- run-level defines the run level in which the entry can be processed. If this field is empty, all run levels are assumed.

- action identifies what actions to take for this entry. These actions may include:
  - sysinit - perform system initialization,
  - wait - wait for the process to complete,
  - once - run the process only once,
  - respawn - restart the process whenever it dies.
- process specifies the shell command to be run, if the entry's run level matches the run level, and/or the action field indicates such action.

In general, the following tasks are performed in the startup scripts.

- Set the hostname.
- Set the time zone.
- Run the file system consistency check.
- Mount the system's disk partitions.
- Start the daemons and network services.
- Configure the network interface.
- Turn on the accounting and quotas.

5.3 Unix Shutdown and Reboot

It is critical for system administrators to have a firm understanding of how the machine is being used and actively involve the users in scheduling downtime. For example, on most systems (except Cray to my knowledge), a shutdown will cause all user processes to be killed. If users on a system are running jobs that take days or weeks to complete then shutting the system down and cause all processes to be killed could severly impact the productivity of users. Whenever possible, users should be given as much lead time as possible when scheduling a shutdown. Once brought up to multi-user mode it is not uncommon for the system to run for many days, possibly even months, without being shutdown or rebooted. There are valid reasons for shutting down the system, these include:

- Many systems now have a graphics dispaly and use an assortment of X11 based applications. Also, it is not uncommon for a server machine to support remote X11 applications. Under many vendors version of X11 there are known memory leaks. These memory leaks result in the X11 server or application allocating memory and never releasing it. Over time you may find that free memory becomes tight. Rebooting will elliminate that.
- Installation of system software or changes in hardware often require a system reboot to take affect.
- Devices can get in a state where they don't function properly. The only fix is to shutdown the system and power off the component. Likewise, system software may get in a confused state and require a reboot to be corrected.
- Often, system administrators bring the system down to single-user mode to perform full backups of file systems. Performing a full backup on a quiescent is one way of guaranteeing a complete backup.

5.3.1 Methods of shutting down and rebooting

There are three possible states you can end up in when you start to shutdown the system, these are:

- Single-user mode;
- The system is completely shutdown and ready to be powered off;
- The system is shutdown put then brought immediately back up without any intervention.

Single-user mode

Previously when we discussed single-user mode we went over some of the tasks you may want to accomplish here. To leave multi-user mode under a BSD system you can enter the command shutdown time [message] , where time can be in absolute or relative terms. For relative time, use a value such as +5 to refer to five minutes from now. Absolute time is referenced as HH:MM and uses 24 hour notation. Finally, the keyword now may be specified to start a shutdown immediately. The message parameter is optional, but highly recommended. This should be enclosed in quotes and give a brief explanation of the reason for the shutdown and when the machine may be back up again.

Under System V, shutdown is accomplished by issuing the command shutdown -y -i1 -g### . Where the -y option informs shutdown to auto-answer all questions with yes; -i1 instructs the system to go to init level 1 (single-user mode); -g### is the grace period to give users before shutting down. The ### symbols should be replace with the number of seconds to wait. Note that there is no message field to use under System V. It is strongly recommended that the system manager use a command such as wall to send a message to all users informing them of the reason for the shutdown and the time when the machine will be available again.

A complete shutdown

A complete shutdown is often done when hardware maintenance is planned for the machine or any other time the machine is to be powered off. On BSD based systems the shutdown command may be specified with the command option of -h to specify that the system should be completely shutdown and the processor halted. As mentioned above, the shutdown command accepts options for the grace time to give before shutdown and a message to send to users. In addition, most systems have a command name halt. In fact, the shutdown -h command usually just invokes the halt command. When you halt the system, all processes are killed, and the sync command is called to write the memory-resident disk buffers to disk. After which, the CPU is halted.

Under System V. based systems the same shutdown command is used as was described above except the init-level is set to zero, as in shutdown -y -i0 -g### . Again, as in BSD based systems, all processes are killed and the sync command is called to write the memory-resident disk buffers to disk before halting the CPU.

The system is being rebooted

Systems are rebooted when changes have been made to the operating system and the Unix kernel must be restarted to take advantage of those changes. This is the case when Unix kernel parameters are changed. Often, for many changes software application changes a reboot is not required but may simplify the process of installing a new or updated piece of software.

Under BSD based systems, the shutdown command is again used to accomplish a reboot. The -r option is specified to the shutdown command and causes the system to be shutdown and then automtically rebooted. Similar to the halt, command there is a seperate command named reboot which is what the shutdown -r command actually invokes.

Under System V. based systems the same shutdown command is used as was described above except the init-level is set to six, as in

```
shutdown -y -i6 -g### .
```

As was mentioned previously, it is good policy to issue a wall command before starting the shutdown so you can inform users of the upcoming reboot.

Other ways to bring your system down, NOT!

Powering the machine off will shutdown the system, however it is very likely that some amount of disk corruption will occur when this happens. This is because the disk buffers cannot be written out to disk via a sync command. While the system performs a sync command at periodic intervals, if the system has heavy disk IO taking place corruption is likely to occur. That said, there are times when the system is hung, usually waiting on some device, and no keyboard input will be accepted.

5.4 Operating System install overview

The operating system boot and operation system installation process for a Unix system are conceptually the same for each vendor.

1. Find the media to install from, usually a CD-ROM.
2. Investigate the Unix system hardware using the boot monitor, to determine the hard drive's and CD-ROM drive's addresses. Most Unix systems use a SCSI bus to access storage devices.
3. Install SCSI devices if necessary.
4. Insert the CD-ROM in the drive.
5. Boot the Unix system from the CD-ROM device. Using the boot monitor to specify the CD-ROM drive as the target.
6. Each vendor provides an automated way of installing their operating system, though, the tasks that are performed are very similar.
    1. Set up the hard drive partitioning and create one or multiple filesystems to hold the system files.
    2. Choose the install packages that you want to install. Besides installing the base operating system, you can select many optional software packages to load.
    3. Select and set the hostname and IP address.
    4. Specify the default route, or gateway, for the network.
    5. Set the local time zone.
7. Reboot the system from the hard disk drive.
8. Edit system configuration files to customize the operation of the machine.

5.5 What is a [UNIX System] Administrator?

Noun:

1. One who administers; *one who manages*, carries on, or *directs the affairs* of any establishment or institution.

2. One who has the *faculty of managing or organizing*?

3. One who *executes or performs the official duties* of religion, justice, etc.; one who dispenses or ministers to the public in such matters.

4. One to whom *authority is given* to manage estates, etc. for the legal owner during his minority, incapacity, etc.; a trustee, a steward. esp. in Sc. Law. `A person *legally empowered* to act for another whom the law presumes incapable of acting for himself' (Encyc.Brit.), as the father of children under age. The bottom line: Administration of a Unix system on the Internet -- or any other system, for that matter -- is a full-time responsibility, not a part-time, casual activity.

5.5.1 Some Common System Administration Tasks.

- Performing backups
- Adding and removing users
- Adding and removing hardware
- Restoring files from backups that users have accidentally deleted
- Installing new software
- Answering users' questions
- Monitoring system activity, disc use and log files
- Figuring out why a program has stopped working since yesterday, even though the user didn't change anything; honest!
- Monitoring system security

- Adding new systems to the network
- Talking with vendors to determine if a problem is yours or theirs, and getting them to fix it when it *is* their problem
- Figuring out why "the network" (or "Pine", or "the computer") is so slow
- Trying to free up disc space
- Rebooting the system after a crash (usually happens when you are at home or have to catch a bus)
- Writing scripts to automate as many of the above tasks as possible

5.6 Types of Unix users.

- root (a.k.a. "super user"--guard this account!)
- everybody else

Becoming root

- Log in as root from the system console (avoid)
- Use "/bin/su -" command from your regular account (better method)

5.7 Things to be aware of when using root account

- $HOME is / (Watch what you delete!)
- Change the password frequently and use "good" passwords (more on this later)
- Remove the current working directory (a.k.a., ".") from your PATH
- Never leave the terminal unattended, even for "just a minute"
- Limit who has the root password to as few people as possible
- Never execute any regular user's program as root (possible Trojan Horse)
- Never let anyone else run a command as root, even if you are watching them

- You risk having your password stolen by "sniffers" if you use programs like "telnet", "ftp", access email remotely with POP/IMAP (anything protocol involving "clear-text" passwords)

5.7.1 Some Useful Tools.

Essential system administration tools.

- man
- su
- ps
- find
- egrep
- df/du
- vi (or emacs) editor
- Bourne Shell (/bin/sh) or Korn Shell (/bin/ksh)
- make

5.7.2 ps command (System V)

Common options:

- -e print all processes
- -f print full listing
- -l long listing (more info than -f)

Meaning of full listing columns:

- S state

- PRI priority
- SZ total size (in 4096 byte pages) of the process
- RSS total resident size (in pages) of the process
- STIME starting time
- TIME cumulative execution time

5.7.3 ps command (BSD)

Common options:

- -a print all processes involving terminals
- -e print environment and arguments
- -l long listing
- -u print user information
- -xi nclude processes with no terminals

Meaning of user information columns:

- %CPU percentage use of CPU
- SZ total size (in 1024 byte pages) of the process
- RSS total resident size (in pages) of the process
- STAT state of the process
- TIME time, including both user and system time

Here is an example of the output of ps under SunOS (System V style).

5.7.4 find command

# find starting-dir(s) matching-criteria-and-actions

Matching criteria

- -atime n file was accessed n days ago

- -mtime n file was modified n days ago
- -size n file is exactly n 512-byte blocks
- -type c file type (e.g., f=plain, d=dir)
- -name nam file name (e.g., `*.c')
- -user usr file's owner is usr
- -perm p file's access mode is p

Actions

- -print display pathname
- -exec cmd execute command ({} expands to file)

find examples

```
# find . -name \*.c -print
# find / -size +1000 -mtime +30 \
 -exec ls -l {} \;
# find / \( -name a.out -o -name core \
 -o -name '#*#' \) -type f -atime +14 \
 -exec rm -f {} \; -o -fstype nfs -prune
```

(Removes unnecessary files that are older than two weeks old, but doesn't descend NFS mounted file systems while searching)

```
# find / \( -perm 2000 -o -perm 4000 \) \
-print | diff - files.secure
```

5.8 Users, Groups and Passwords

- Users

- Groups
- */etc/passwd*
- */etc/shadow*
- Creating user accounts
- Deleting user accounts
- Making changes

One thing that Unix is frequently accused of not doing well is security. In terms of system administration, this means that the way user accounts are added to a system is of particular importance. If this is done wrong the security of the system can be severely compromised.

Users

Each user on a system must have a login account identified by a unique username and UID number. Unique means that within a domain, like *indiana.edu*, there can only be one user with the same username. Login names are typically all lowercase and a maximum of eight characters.

Most organizations with many users will establish a set way to form usernames. IU usernames are now last name based. This hasn't always been the case, so there are IU users whose username may be their first name or something else.

Each user must also have a unique UID (user ID) number. This is an integer between 0 and 32767, although some systems permit numbers up to 65534. In networks using NFS the UID must be unique across the network, as well as being attached to the same username and person. UID numbers are conventionally assigned to human users beginning with 100. Numbers less than 100 are reserved for system accounts.

1. Is it important that the username as well as UID always refer to the same person?

- No, the username is less important than the UID.

In terms of mail and programs such as *login*, the username is more important than the UID. However, the kernel is much more concerned with UID and GID numbers than with usernames.

- Yes, this is important for system security.

This particularly important in a networked computing environment where there the probability of having multiple users with the same name is greater. If the same username refers to more than one person under certain conditions those users would be able to access each others files.

Having a single username refer to more than one user also has potential to cause problems for the mail system, particularly if mail is delivered to one central machine.

It is recommended that UID numbers not be reused, even if the user has left the system. If system files are ever restored from tape reusing UID numbers can cause problems as users are identified by UID number on tape.

Groups

Each user in a system belongs to at least one group. Users may belong to multiple groups, up to a limit of eight or 16. A list of all valid groups for a system are kept in */etc/group*. This file contains entries like:

```
work:*:15:trsmith,pmayfiel,arushkin
```

Each entry consists of four fields separated by a colon. The first field holds the name of the group. The second field contains the encrypted group password and is frequently not used. The third field contains the GID (group ID) number. The fourth field holds a list of the usernames of group members separated by commas.

- GID's, like UID's, must be distinct integers between 0 and 32767. GID's of less then 10 are reserved for system groups. These default GID's are assigned during the installation of the operating system.

/etc/passwd

The */etc/passswd* file contains a list of users that the system recognizes. Each user's login name must be in this file. Entries in */etc/passwd* look something like this:

```
arushkin:Igljf78DS:132:20:Amy Rushkin:/usr/people/arushkin:/bin/csh
trsmith:*:543:20:Trent Smith, sys adm:/usr/people/trsmith/:/bin/tcsh
```

Although these entries differ in terms of the way the information is presented within the fields, they are both valid */etc/passwd* entries. The first field contains the user's login name. The second contains the user's password. In the entry for *arushkin* the password has been encrypted by the system and appears as a nonsensical string of characters. In the entry for *trsmith* the password field is occupied by a placeholder. This can mean that the user does not have a password. In the */etc/group* file, if a group does not use a password a placeholder is put in the password field rather than leaving the field blank. A blank field constitutes a security hole through which an unauthorized user could gain access to the system. A placeholder in the password field can also indicate that a shadow password file is in use. In that case, the actual password is kept in */etc/shadow*.

The third field holds the user's UID and the fourth contains the default GID. The GID is the number of the group that the user is a member of when they log in. The fifth field is the GCOS field. It has no defined syntax and is generally used for personal information about the user; full name, phone number, etc. Frequently this field is not used at all. The sixth field contains information about where the users home directory is located. The last field contains the login shell. This is the shell that *login* starts for the user when they log in.

/etc/shadow

The */etc/passwd* file is world readable, but */etc/shadow* is readable only by root. SVR4 based systems support *pwconv*, which creates and updates */etc/shadow* with information from */etc/passwd*. When */etc/shadow* is used an 'X' is placed in the password field of each entry in */etc/passwd*. This tells *pwconv* not to modify this field because the passwords are kept in */etc/shadow*.

If */etc/shadow* doesn't exist *pwconv* will create it using the information in the */etc/passwd* file. Any password aging controls found in */etc/passwd* will be copied to */etc/shadow*. If the */etc/shadow* file already exists, *pwconv* adds entries in */etc/passwd* to it as well as removing entries that are not found in */etc/passwd*.

Entries in */etc/shadow* look something like this:

```
trsmith:56HnkldsOI2Z:543:14:180:10:60::
```

The first field contains the user's login name and the second holds the user's encrypted password. The third contains the date that the user's password was last changed. The fourth sets the minimum number of days that a password must be in existence before it can be changed. The fifth field sets password's life span. This is the maximum number of days that a password can remain unchanged. If this time elapses and the user does not change the password, the system administrator must change it for them. The sixth field is used to dictate how long before the password's expiration the user will begin receiving messages about changing the password. The seventh contains the number of days that an account can remain inactive before before it is disabled and the user can no longer log in. The eighth field can be used to specify an absolute date after which the account can no longer be used. This is useful for setting up temporary accounts. The last field is the flag field and is not used.

Not all flavors of Unix use all of these controls. In addition, the syntax of aging controls varies from platform to platform. To find out which aging controls can be set on a particular system it is best to consult the man page for *passwd*, *usermod*, etc. On some systems aging controls can also be added to an account at the time it is created using graphic tools.

5.8.1 Creating user accounts

Different "flavors" of Unix provide different tools or scripts to use when creating and deleting user accounts. HP-UX uses *SAM*, the System Administration Manager. IRIX has *User Manager*, found in the *System* pull down menu. Solaris provides the *admintool* and *useradd* utilities. Under Solaris 2.5, the graphic interface to the *admintool* looks very much like *SAM*. With these tools creating a user account is simply a matter of entering information when prompted. The tool takes that information and creates the account. There are also scripts to create accounts, such as *adduser*, that can be started from the command line. This script is available from a variety of sources, including the CD that comes with the second edition of the *UNIX System Administration Handbook*. Linux comes with an *adduser* program.

It is a good idea to understand how a new account is created so that it can be done manually if necessary. Creating a new account involves:

- Editing the */etc/passwd* file
- Editing */etc/group*
- Creating a home directory for the user
- Copying the startup files into that home directory
- Setting an initial password for the account

If at all possible is a good idea not to edit the */etc/passwd* file manually. Modifying this file by hand will eventually cause it to become corrupted and a corrupt */etc/passwd* file is security risk. If a situation arises in which the file must be modified by hand it is crucial that a copy of the file be made before it is modified. This way if the file does become corrupt, an uncorrupted copy still exists. When the file is modified manually, the sequence of opening the file, writing to it, and closing it should occur on three consecutive command lines. This way the file is not open any longer than absolutely necessary.

BSD derivative systems, like SunOS, use *vipw* to edit the password file. Some distributions of Linux also supply this command. *vipw* is a special version of the vi editor. It makes a copy of */etc/passwd* named *ptmp* that is then edited. When this copy is made the password file is locked so that no one can execute it while it is being edited. This is done to avoid a race condition. If a user executes the file while it is being edited a race begins to see which process writes to the file first. After changes to *ptmp* are complete *vipw* checks the consistency of the root entry before writing *ptmp* back to */etc/passwd*. *vipw* is available for other systems, like Solaris, as part of the SunOS/BSD Compatibility Package Commands.

The script will automatically select a UID and GID for the new account, defaulting to the next highest unused integer. The system administrator can change both of these values if a specific number is desired. The script may also set a default home directory name. By convention home directories are given the name of the user's login. The default login shell on most systems is the Bourne shell. If the user has another shell preference or the system administrator wants the user to use a different shell this can be changed.

The default settings used by the script that creates user accounts are referred to as skeleton files. They are kept in */usr/skel*, */etc/skel* or */etc/security*.

A script can also be used to edit */etc/group*. Solaris supports the *groupadd* command for adding users to existing groups or adding new groups to the system. BSD uses *set group*. BSD also differentiates between the real group and group set. A users real group is the group specified in */etc/passwd*. The group set is entire set of groups to which the user belongs. If */etc/group* is modified manually it is a good idea to take the same precautions as when editing */etc/passwd* by hand.

A home directory for the new user can be created using the *mkdir* command. This directory must have the same name as the home directory specified in */etc/passwd*. At the time it is created the directory will be owned by root. The system administrator must use the *chown* and *chgrp* command to change the directories ownership. The *adduser* script includes the option to have the home directory created automatically. Additional information is needed to create accounts with *useradd* on Solaris systems.

5.8.2 Deleting user accounts

Most systems provide a script, such as *userdel* for deleting user accounts. It can also be done with the graphical tools provided by the system. If the need arises to do it manually the process of deleting a user consists of :

- Removing the */etc/passwd* entry
- Removing the */etc/shadow* entry
- Removing the user's login name from groups in */etc/group*
- Deleting the home directory and startup files

5.8.3 Making changes

Once a user's initial account information is established it can be changed manually or with various scripts. Solaris provides the *usermod* for changing users status as well as *groupadd*, *groupmod*, and *groupdel* for modifying groups. BSD supports the *groups* command for changing the group set or the designated real group. If *groups* is used with login name as an option it will list all the groups to which that user belongs.

In addition many systems support commands such as *passwd* that allow users to change their own passwords. After a password has been changed the *netpass* command can be used to change the password across the network.

The status of a user account can also be changed with the tools provided by various Unix "flavors". Under IRIX the *User Manager* can be used. In HP-UX *SAM* can be used to change information about a current user as well as adding new users.

5.9 UNIX file system

A file system is a logical method for organizing and storing large amounts of information in a way which makes it easy manage. The file is the smallest unit in which information is stored. The UNIX file system has several important features.

- Different types of file
- Structure of the file system
- Access permissions

Different types of file

To you, the user, it appears as though there is only one type of file in UNIX - the file which is used to hold your information. In fact, the UNIX filesystem contains several types of file.

- **Ordinary files**

This type of file is used to store your information, such as some text you have written or an image you have drawn. This is the type of file that you usually work with.

Files which you create belong to you - you are said to "own" them - and you can set access permissions to control which other users can have access to them. Any file is always contained within a directory.

- **Directories**

A directory is a file that holds other files and other directories. You can create directories in your home directory to hold files and other sub-directories.

Having your own directory structure gives you a definable place to work from and allows you to structure your information in a way that makes best sense to you.

Directories which you create belong to you - you are said to "own" them - and you can set access permissions to control which other users can have access to the information they contain.

- **Special files**

This type of file is used to represent a real physical device such as a printer, tape drive or terminal.

It may seem unusual to think of a physical device as a file, but it allows you to send the output of a command to a device in the same way that you send it to a file. For example:

```
cat scream.au > /dev/audio
```

This sends the contents of the sound file scream.au to the file /dev/audio which represents the audio device attached to the system. Guess what sound this makes?

The directory /dev contain the special files which are used to represent devices on a UNIX system.

- **Pipes**

Unix allows you to link two or more commands together using a pipe. The pipe takes the standard output from one command and uses it as the standard input to another command.

```
command1 | command2 | command3
```

The | (vertical bar) character is used to represent the pipeline connecting the commands.

With practice you can use pipes to create complex commands by combining several simpler commands together.

The pipe acts as a temporary file which only exists to hold data from one command until it is read by another.

Examples:

To pipe the output from one command into another command:

```
who | wc -l
342
```

This command tells you how many users are currently logged in on the system: a lot!

The standard output from the who command - a list of all the users currently logged in on the system - is piped into the wc command as its standard input. Used with the -l option this command counts the numbers of lines in the standard input and displays the result on the standard output.

To connect several commands together:

```
ps -aux|grep joe|sort +5 -6|less
```

The first command ps -aux outputs information about the processes currently running. This information becomes the input to the command grep joe which searches for any lines that contain the username "joe". The output from this command is then sorted on the sixth field of each line, with the output being displayed one screenful at a time using the less pager.

5.10 Structure of the file system

The UNIX file system is organised as a hierarchy of directories starting from a single directory called *root* which is represented by a / (slash). Imagine it as being similar to the root system of a plant or as an inverted tree structure.

Immediately below the root directory are several system directories that contain information required by the operating system. The file holding the UNIX kernel is also here.

- **UNIX system directories**

The standard system directories are shown below. Each one contains specific types of file. The details may vary between different UNIX systems, but these directories should be common to all. Select one for more information on it.

```
                 /(root)
                   |
 ------------------------------------------------------------
 |     |     |     |     |     |     |     |
 /bin   /dev   /etc   /home   /lib   /tmp   /usr   kernel file
```

- **Home directory**

Any UNIX system can have many users on it at any one time. As a user you are given a home directory in which you are placed whenever you log on to the system.

User's home directories are usually grouped together under a system directory such as /home. A large UNIX system may have several hundred users, with their home directories grouped in subdirectories according to some schema such as their organizational department.

- **Pathnames**

Every file and directory in the file system can be identified by a complete list of the names of the directories that are on the route from the root directory to that file or directory.

Each directory name on the route is separated by a / (forward slash). For example:

    /usr/local/bin/ue

This gives the full pathname starting at the root directory and going down through the directories usr, local and bin to the file ue - the program for the MicroEMACS editor.

You can picture the full pathname as looking like this:

```
          /(root)
             |
             |
      ---------------------
        |           |
       tmp          usr
                    |
             ---------------- ... ----
              |      |          |
          /games   /local      /spool
                     |
               ---------------
                   |
                  /bin
                   |
                   ---------
                         |
                        ue
```

- **Relative pathnames**

You can define a file or directory by its location in relation to your current directory. The pathname is given as / (slash) separated list of the directories on the route to the file (or directory) from your current directory.

A .. (dot dot) is used to represent the directory immediately above the current directory.

In all shells except the Bourne shell, the ~ (tilde) character can be used as shorthand for the full pathname to your home directory.

5.11 Controlling access to your files and directories

Every file and directory in your account can be protected from or made accessible to other users by changing its access permissions.

You can only change the permissions for files and directories that you own.

- **Displaying access permissions**

To display the access permissions of a file or directory use the the command:

ls -l *filename* (*directory*)

This displays a one line summary for each file or directory. For example:

```
-rwxr-xr-x  1 erpl08   staff      3649 Feb 22 15:51 my.html
```

This first item -rwxr-xr-x represents the access permissions on this file. The following items represent the number of links to it; the username of the person owning it; the name of the group which owns it; its size; the time and date it was last changed, and finally, its name.

- **Understanding access permissions**

There are three types of permissions:

r read the file or directory

w	write to the file or directory
x	execute the file or search the directory

Each of these permissions can be set for any one of three types of user:

u	the user who owns the file (usually you)
g	members of the group to which the owner belongs
o	all other users

The access permissions for all three types of user can be given as a string of nine characters:

user	group	others
r w x	r w x	r w x

These permissions have different meanings for files and directories.

Summary of access permissions

Permission	File	Directory
r read	read a file	list files in ...
w write	write a file	create file in ...
		rename file in ...
		delete file ...
x execute	execute a	read a file in ...
	shell script	write to a file in ...

execute a file in ...

execute a shell script in ...

- Default access permissions

When you create a file or directory its access permissions are set to a default value. These are usually:

rw-------

gives you read and write permission for your files; no access permissions for the group or others.

rwx------

gives you read write and execute permission for your directories; no access permissions for the group or others.

Access permissions for your home directory are usually set to rwx--x--x or rwxr-xr-x.

- **Changing the group ownership of files**

Every user is a member of one or more groups. To find out which groups you belong to use the command:

```
groups
```

To find out which groups another user belongs to use the command:

groups *username*

Your files and directories are owned by the group (or one of the groups) that you belong to. This is known as their "group ownership".

To list the group ownership of your files:

```
ls -gl
```

You can change the group ownership of a file or directory with the command:

```
chgrp group_name file/directory_name
```

You must be a member of the group to which you are changing ownership to.

- **Changing access permissions**

To change the access permissions for a file or directory use the command

```
chmod mode filename
chmod mode directory_name
```

The "mode" consists of three parts: who the permissions apply to, how the permissions are set and which permissions to set.

Who means

This is specified as one of:

u	(user)	the owner of the file
g	(group)	the group to which the owner belongs
o	(other)	everyone else
a	(all)	u, g and o (the world)

How means

This is given as one of:

+	add the specified permission

- subtract the specified permission

= assign the specified permission, ignoring

whatever may have been set before.

Which means

These are specified by one or more from:

r read

w write

x execute

Remember that these permissions have different meanings for files and directories.

5.12 Backup and File Restoration

Disk failure, while much less prevalent now, still happens and as the system administration it is your responsibility to gaurentee you have taken the needed precautions to recover. It is *extremely* important to let people know what your strategy is and to verify you can meet the expectations of your organization. State in writing what you can and cannot do and make sure your boss and the users you support understand this.

Historical Context

Over the years the computer industry runs in cycles, as technology to produce mass storage improves, the cost of disk space drops. Whereas 10 years ago the cost of 1 MB of disk storage averaged above $10 a MB, today that cost has fallen to under $.25 a MB (e.g. a 1GB drive selling for $229). This drop in cost has resulted in much more disk space being bought and placed online. For example, this past year my group has ordered over 50GB of new disk storage, Five years ago we survived on under 10 GB.

Unfortunately archival capacity has not fallen in price as quickly. As administrators you can easily add disk space and out-strip your capacity to back up that disk space.

Types of Backup Media

Backup media has generally fallen into two categories. One category is tape media and the other is removal cartridge. Tape media is generally cheaper is cost and supports larger sizes; however, tape media does not easily support random access to information. Removal cartridge drives, whether optical or magnetic, do support random access to information; however they have lower capacity and a much higher cost per megabyte than tape media. Finally, optical drives generally retain information for a longer time period than tapes and may be used if a permanent archival is needed (side note: Nothing is permanent, if that is a requirement, you must consider ways to make additional backup copies and periodically check the media).

Among tape media, the two most common choices now are 4mm DAT (digital audio tape) and 8MM Video Tape. The 4mm tape supports up to 2GB of data on a single tape (with compression this can approach 8GB). The 8mm tape supports up to 7GB of data on a tape (with compression this can approach 25GB). Both of these technologies have been in existence since the late eighties and are relatively proven. However, while they changed the dynamic of backup at that their introduction they are now having problems keeping up with the growth in data to be backed up. One of the principal problems is speed. At their maximum they can backup about 300K bytes/sec or just under 1GB in an hour. That sounds wonderful till you have to backup 50 GB in a night!

Among cartridge media, optical is the most commonly used. WORM (write once read many) is the primary choice. Worm drives can store between 600MB and 10GB on a platter and have a throughput similar to cartridge tapes for writing data. Optical storage will last longer and supports random access to data. These features make optical storage useful for storing billing or record information. Coupled with a Jukebox for housing multiple optical platters this can be a good solution for certain types of data.

Many backup options out there:

tar

tape archive program, easy to use and transportable. has limit on file name size, won't backup special files, does not follow symbolic links, doesn't support multiple volumes.

advantage is that tar is supported everywhere. Also useful for copying directories

```
tar -cf . | ( cd /user2/directory/jack; tar -xBf -)
tar cvfb /dev/tape 20 .
tar xvf /dev/tape .
```

cpio

Copy in/out archive program, rarely used except on older System V type machines. Must be given a list of file names to archive. Use find to do this:

```
find . -print | cpio -ocBV > /dev/tape
```

reloading a archive from tape

```
cpio -icdv < /dev/tape
```

dump

dump (or rdump) reads the raw file system and copies the data blocks out to tape. dump is program that is generally used by most groups. One feature of dump is that dump (via rdump) can work across a network and dump a file system to a remote machine's tape drive. Dump cannot be used to back up just a specific directory. Thus for backup of a directory tree tar or cpio is better suited.

dump backs up all files in filesystem, or files changed after a certain date to magnetic tape or files. The key specifies the date and other options about the dump. Key consists of characters from the set 0123456789fuscdbWwn. Any arguments supplied for specific options are given as subsequent words on the command line, in the same order as that of the options listed.

If no key is given, the key is assumed to be 9u and the filesystem specified is dumped to the default tape device /dev/tape.

Dump uses level numbers to determine whether a full or partial dump should be done. The levels can be between 0 and 9.

Designing a Backup Strategy

In a perfect world you will have the resources necessary to backup all data on all disks each night. However, the norm is that this is not the case, if that is so then you should Ask yourself these questions:

- What happens if I can't recover data ?
  If you are a bank, what happens if you lost the fact I made an atm withdrawl?
- How long can you be down if you have to recover data from archive? Could our organization function if we were down for a day or more without while this data was being recovered?

If you must have guaranteed access to data you might consider using some form of RAID disk storage. RAID storage, coupled with optical or tape backup can deliver near continuous uptime.

If you can live with some risk and don't have the tape capacity to back up all file systems each night then you must develop a backup rotation stragety. This strategy defines how you will use your tape resources to cover the data resources in your organization. For example, one form of backup rotation might be you completely backup a file system every other night, meaning in a worst case situation you could lose up to 2 days of work! Alternately, you may define a strategy where you do occasional complete file system backups (called a full backup) and each night backup information that has changed since some date (an incremental backup).

Under Unix, if you use the dump command to backup your file system you have alot of flexibility in defining this strategy. Dump gives you the ability to assign a level to each dump ranging from 0 through 9. A level 0 dump will backup all files on the filesystem. A level 1 dump will back up all files that have changed since the level 0 dump was last performed, a level 2 dump backs up all files since the last level 1 dump or level 0 dump.

(In class I will describe how this works pictorially). Understand Exhibit A on page 188.

The dump command has many options. A sample dump command would look like: dump 0usdf 6000 54000 /dev/tape /users Where
0 - represent the dump level (0 through 9)
u - signifies to update the /etc/dumpdates file
s - is the size of the tape in feet
d - represents the density of the tape
f - represents the device to use for the dump
6000 - is the size parameter in feet
54000 - is the density parameter
/dev/tape - is the device to use
/users - is the filesystem to dump

Tape Drive Basics

Most backup work is done to some form of magnetic tape media. As with disks or other media, to use a tape drive under Unix you must have device files defined for that device.

Most Unix systems will look for an environment variable named TAPE and use that. Also most systems create a hard link from /dev/tape to the actual device name also have a hard link for /dev/nrtape to the default device. If not, you will need to know which device you are using. Under SGI IRIX, tape devices are in /dev/mt/ and start with tps: The special files are named according to this convention:

/dev/{r}mt/tpsd{nr}{ns}{s}{v}{.density}{c}
/dev/rmt/jagd{nr}{ns}{s}{v}{stat}{.density}

Where {nr} is the no-rewind on close device, {ns} is the non-byte swapping device, {s} is the byte swapping device, and {v} is the variable block size device (supported only for

9 track, DAT, DLT and 8 mm devices as shipped). Density is tape density or type of device it is emulating (e.g. exabyte 8200 or exabyte 8500).

Unix creates multiple device files for the same tape device to handle different types of functions. NR means that the tape will not rewind when you close the tape device, this is needed if you want to place multiple files on a tape, one after another. NS means non-byte swap and is used by default, S is the byte swap device and is used when making tapes for machines such as sun systems. V allows variable block sizes to be written and requires a special tape device. Density is the type of tape device you are using and helps determine maximum size.

Restoring Files

Hopefully this won't have to happen but you need to be prepared. However, in practice something always arises to require a restore. To restore an entire file system you must remake the file system (mkfs, mount), then cd to that file system and run the command restore -r. If you have multiple levels of the file system restore will restore the files in reverse order of when they were dumped.

To restore just one file, use the restore command in interactive mode. This is done with the command restore -i. In interactive mode, you use the basic unix commands to navigate on the restore tape and then use the sub-command add filespec to add that file to the list to be extracted. When complete, use the command extract to have restore load in the files from tape. Add a directory to the list also causes all of the files within that directory to be added.