Introduction to the Unix Operating System

Scott Dickson
Penn State University
Center for Academic Computing
814 865 0829
dickson@iris.psu.edu

16 March 1992

1 Introduction

This document is designed to accompany the CAC Introduction to the Unix Operating Systems seminar. The seminar is a two session introduction to many of the concepts behind the Unix system, as well as to its basic workings. Much of the information contained in it may not be able to stand alone and requires the seminar to really make sense. Software to allow the user to create an environment similar to that in the seminar, thereby allowing the user to use this as a tutorial is in the works. Hopefully it will be useful to you.

While no knowledge of any other particular computing system is required, it is helpful to understand in advance some of the basic concepts of computing before continuing. In this document, we will assume basic understanding of what a command is, what files and directories (or folders) are, and basically how one would interact with a computer. Knowledge of DOS is also helpful, though by no means required.

The following is a summary of some of the topics that will be covered in the seminar and in this document:

- Overview of the major features of Unix.
- A short history of Unix.
- Major design goals of Unix.
- Structure of the Unix filesystem.
- A sample login session.
- Command format.
- Filesystem navigation and manipulation.
- Standard utilities and commands.
- Processes and jobs.
- Pipelines and data redirection.
- Getting more information.

There is no attempt to cover all material completely in this document or seminar. Time and space just do not allow for this, so many times you will be referred to other documentation or manuals. Become familiar with the online help facility under Unix. You *will* find yourself using it often.

2 A Little Background

2.1 Basic Characteristics of Unix

So, what is Unix anyway? Unix is, at its base level, a multi-user, multi-tasking, 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 installled. 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. 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 priviledges 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.

Unix is a multi-tasking 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. Multi-tasking, 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 time-sharing 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.

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 multi-tasking 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. For more information on virtual memory in general, see any text on operating system design. For more information about the details of the structure of Unix, see [?] and [?].

2.2 Genealogy of Modern Unix Systems

Unix was born at AT&T Bell Laboratories in 1968 and Dennis Ritchie and Ken Thompson were its parents. While it may seem like Unix is a newcomer on the computer scene, it has had its cadre of followers for over twenty years. Much of the early history is based in mythology, but it is true that one of the forces that led to Unix' development was the desire to play a game called Space Travel. Ken Thompson, Dennis Ritchie, and other staff at Bell Labs Computer Research Group put together the first version of Unix on a DEC PDP-7 in assembly language. After the first version, and the addition of useful tools, such as a text editor and a text formatting system, the popularity of Unix spread. It was rewritten

almost entirely in C and ported to a PDP 11/20. This was a major step, since until that time, all operating systems had been written in assembly language. As Unix' popularity spread, new versions were produced and other groups got into the act. The Computer Systems Research Group at Berkeley came out with their own variant of of the Bell Labs Unix known as BSD, the Berkeley Software Distribution. AT&T also moved Unix from the research lab into its regular product line. As a result, there are a number of different variations of Unix available. Primarily, versions of Unix available from various vendors are either based on 4.3 BSD or on AT&T's System V, though other versions exist. Interestingly enough, one of the major movements within the Unix community today is a movement to bring the various Unix versions back together into a single package. Perhaps ironically, there are multiple movements at unifying all of the various Unix variants, each looking not a lot like the others. For a nice history of Unix, see [?].

2.3 Basic Design Goals of Unix

While Unix has evolved over the last twenty years, there have been several guiding forces that have not been lost in this evolution. These forces are ideas about how operating systems should be built and how you should use the system that have carried through to the present day. A few of these are particularly germaine to the way one uses Unix and keeping them in mind will help make sense of much of the rest of the system.

Probably the most important of these ideas is the idea of rather than building large programs that do a lot of different tasks sort-of well, it is better to build small programs and small commands that do exactly one thing but do it very well. Combine this with the idea of having a consistent interface to all (in so far as it makes sense) programs and having a way of passing data between programs and you have the ability to use the small programs as building blocks. By combining smaller programs into combinations, synergy is acheived with the end result being larger than any of the parts taken separately.

Portability is the other major goal in Unix. When Unix was converted from assembler to C, the idea was that it would be possible to move Unix to many different computers. We see this today, with Unix being available on machines ranging from PC's to workstations to mainframes to supercomputers. This portability is acheived by writing programs that are not specific to any particular machine. A result of this is that programs developed under Unix can be moved from system to system with remarkably little modification.

3 Getting Started

3.1 Connecting to the System

Now, we want to connect to a Unix system and begin to explore. First, you will need an account on some Unix machine. An account consists of a username and a password. The username is the way that you identify yourself to the computer and the way that other users of the system can identify you as well. The password is used to authenticate that you are actually who you claim to be. Like everything in Unix, both the username and the password are case-sensitive and must be typed exactly.

How you go about getting this account depends upon what machine you want to use. At Penn State, many departments and colleges have Unix workstations available for students, faculty, and staff in their area. Likewise, many researchers have their own Unix systems. Within the CAC, both Sun Microsystems machines and NeXT Computers are available. Applications for accounts on these systems may be obtained from the documentation racks outside Room 215 Computer Building or by contacting a member of the

CAC's Workstations and Visualization Group.

Gaining access to the actual system may be as simple as sitting down in front of a terminal or workstation, but it also might involve connecting via some sort of terminal server or dataswitch. For now, we will assume that you are seated in front of a Unix workstation. If you need assistance connecting to your system via another route, check with your local guru or local system administrator.

Once you have connected to the Unix machine, you will be presented with a login prompt. This can take many forms, but will generally look something like Figure ??. To log in, enter your username and press return. You will be prompted for your password. Type it in, and if it is correct, you will be presented with a prompt.

```
Penn State Center for Academic Computing (wilbur)
SunOS 4.1.1

** This system is restricted to uses approved by Penn State University **

login:
```

Figure 1: An Example Unix Login Prompt

3.2 Changing Your Password

Once you have typed your password, you will be logged in and given a prompt. The prompt is Unix's way of signalling that it is ready to receive commands from you. The prompt may vary, but the default prompt for all Unix systems is the percent sign (You may customize the prompt however you want. This is covered in the seminar on customizing your Unix environment.

If this is the first time you have logged on, the first thing you will want to do is to change your password. This is done with the `passwd` command, as shown below.

```
% passwd
Changing password for dickson
New Password:
Retype new password:
Changing password for dickson.
```

Figure 2: Changing a Password

3.3 Selecting a Good Password

The need for good passwords may not be immediately obvious to you if you have not used many multi-user systems, but they are important. The password is what keeps people from masquerading as other users and from accessing data and services to which they are not entitled. Misuse of other people's passwords and

using other accounts without authorization is generally a criminal offense at the state level, but in some cases can be considered a Federal crime as well. So, select your password carefully, reveal your password to no one, and change it regularly.

Passwords are limited to eight characters and may contain virtually any key that you can type. Some control keys may not be good choices for inclusion in passwords, but other keys are all good choices. The following list gives some good guidelines for the selection of good passwords [?].

- **Don't** user your login name in any form (as-is, reversed, capitalized, doubled, etc.).
- **Don't** use your first, middle, or last name in any form.
- **Don't** use your spouse's or child's name.
- **Don't** use other information easily obtained about you. This includes license plate numbers, telephone numbers, social security numbers, the make of your automobile, the name of the street you live on, etc.
- **Don't** use a password of all digits, or all the same letter.
- **Don't** use a word contained in English or foreign language dictionaries, spelling lists, or other lists of words.
- **Don't** use a password shorter than six characters.
- **Do** use a password with mixed-case alphabetics.
- **Do** use a password with non-alphabetic characters (digits or punctuation).
- **Do** use a password that is easy to remember, so you don't have to write it down.
- **Do** use a password that you can type quickly, without having to look at the keyboard.

3.4 Getting More Information

Now that you are logged on, you need to find out how to use Unix. On virtually all Unix systems, the complete reference manual is kept online and at your fingertips. The manual is accessed with the `man` command. The format for this command is

`man` *commandname*

For example: `man passwd` will tell you about the `passwd` command. `man` will look up the pages in the manual on the command you asked for. If you are not sure what command to look for, there is a keyword index of all of the commands on the system. Use the command

`man -k` *keyword*

to search for all of the commands related to your keyword. For example, `man -k password` will list all of the manual entries that deal with passwords.

Online manual pages are very nice to have, but are not a complete subsitutute for printed manuals. If you are using the CAC's Sun or NeXT lab, complete sets of vendor documentation are available in the lab, along with documentation for other packages installed on the systems.

Many other Unix books are available in the trade sections at bookstores. Both the Penn State Bookstore, on campus, and Svoboda's bookstore in downtown State College have moderately large computer sections

including several Unix books. One publisher in particular, O'Reilly & Associates, carries a large selection of extremely good Unix publications. Some of their titles include "Learning the Unix Operating System" [?], "Unix in a Nutshell" [?, ?], "DOS Meets Unix" [?], and "Unix for FORTRAN Programmers" [?]. In addition, another particularly good and entertaining book is "Life with Unix" [?]. This book gives, in addition to the basic information on using Unix, an interesting look at its history, development, and culture.

4 Dealing with Unix Files

4.1 So, What's the Big Deal About Files?

It has been said that once you 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. We will talk more about these later. For now, remember that everything is a file to Unix.

4.2 What Files Do I Have?

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.

4.3 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 that you can never be expected to remember them all (that's what the `man` command is for).

4.4 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 modifer `-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
-rw-r--r--  1 dickson      2639 Jan 30 18:02 baby.1
drwxr-xr-x  2 dickson       512 Jan 30 18:02 bin
-rw-r--r--  1 dickson      9396 Jan 30 18:03 ig.discography
-rw-------  1 dickson      5239 Jan 30 18:10 mbox
drwxrwxr-x  3 dickson       512 Jan 30 18:06 src
-rw-r--r--  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.

4.5 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.

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`, that they have execute permissions set. Execution permission is interpreted differently for directories than for plain files and will be discussed shortly.

4.6 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 o-r 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 o-r unix.refer
% chmod g+w unix.refer
% ls -l unix.refer
-rw-rw----  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.

4.7 Manipulating files with `cp` and `mv`

What if we wanted to make a copy of a file? The `cp` command will do this. 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
-rw-r-----  1 dickson      1311 Jan 30 19:20 NewBibligraphy.refer
-rw-rw----  1 dickson      1311 Jan 30 18:03 unix.refer
```

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

```
% mkdir bibs
% cp *.refer bibs
% ls -l bibs
total 4
-rw-r-----  1 dickson      1311 Jan 30 19:42 NewBibligraphy.refer
-rw-r-----  1 dickson      1311 Jan 30 19:42 unix.refer
```

If you want to change the name of a file, `mv` will rename the file. Actually, tt 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
-rw-r-----  1 dickson      1311 Jan 30 19:42 NewBibligraphy.refer
-rw-r-----  1 dickson      1311 Jan 30 19:42 unix.refer
```

4.8 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.
```

5 Directories and the Unix File System

5.1 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 DOS, 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 a 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.

5.2 Your 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 DOS, 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.

5.3 Naming Unix Files and Directories

5.3.1 Absolute Naming

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

5.3.2 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
```

5.3.3 Short-cuts for File Naming

Since certain file naming patterns are used over and over, Unix provides some short-cuts 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.

5.3.4 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 used mixed-case names to add clarity. While long, meangingful 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.

5.3.5 File Name Extensions

While Unix does not actually have the same notion of a file extension as is found in some other systems, notably DOS, 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 DOS 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 ?? 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 lanaguage source
`.tex`	TEXdocument
`.dvi`	TEXdevice 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

5.4 More About Listing Directories

The `ls` command has some additional options to it that deserve mentioning 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 issed 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
```

5.5 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 DOS, 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
```

5.6 Creating and Removing Directories

Also similar to DOS, 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.

5.7 Symbolic Links

Eariler, we mentioned symbolic links, but never really described them. A symbolic link is a file that is sort of a pointer to another file. Essentially, its contents are the name of the file that it really refers to. Unix knows that links are special, and when you try to acess a link, what you actually get is the file that is pointed to. A link may point to any file on the system. Links are used extensively by the system to make files available in multiple places in the filesystem without having to have duplicate files. However, users rarely create symbolic links on their own.

In order to create a link, you use the command `ln -s`. The following example shows how a link can be used to give an alias to a file, or to call a directory by another name. Notice that we create a link called `Bibliography` that is a link to the directory `bibs`. When we use `ls -F`, we see that a link has been created, and with `ls -lF` we can see what the link points to.

```
iris.1: ls -F
NewBibligraphy.refer    bibs/                   mbox
ShortFile               bin/                    src/
baby.1                  ig.discography          unix.refer
iris.2: ln -s bibs Bibliography
iris.3: ls -F
Bibliography@           baby.1                  ig.discography          unix.refer
```

```
NewBibligraphy.refer    bibs/                   mbox
ShortFile               bin/                    src/
iris.4: ls -lF
total 28
lrwxrwxrwx  1 dickson  cac-vhf         4 Feb  6 17:31 Bibliography -> bibs/
-rw-r-----  1 dickson  cac-vhf      1311 Jan 30 19:20 NewBibligraphy.refer
-rw-r--r--  1 dickson  cac-vhf       140 Feb  6 08:34 ShortFile
-rw-r--r--  1 dickson  cac-vhf      2639 Jan 30 18:02 baby.1
drwxr-sr-x  2 dickson  cac-vhf       512 Jan 30 19:42 bibs/
drwxr-xr-x  2 dickson  cac-vhf       512 Jan 30 18:02 bin/
-rw-r--r--  1 dickson  cac-vhf      9396 Jan 30 18:03 ig.discography
-rw-------  1 dickson  cac-vhf      5239 Jan 30 18:10 mbox
drwxrwxr-x  2 dickson  cac-vhf       512 Jan 30 19:36 src/
-rw-rw----  1 dickson  cac-vhf      1311 Jan 30 18:03 unix.refer
```

6 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
`/tmp`	Temporary files
`/usr`	User-related files
`/usr/local`	Local additions and modifications to Unix
`/home`	User home directories

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

6.1 Using Meta-Characters for File Matching

Sometimes, it is easier to specify some pattern of charcaters 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 DOS, 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 meta-characters

7 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 facilties 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 ther 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 new-rut | pr | more
```

Using pipes, you can connect as many different utilities as you might want to complete a particular job. DOS 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.

8 Processes and Jobs

Early in this document, we said that Unix is a multi-processing operating system. This allows the user to run many programs simultaneously. This is not the same as running many batch jobs, since at any point in the execution of programs under Unix, a job may become the foreground job, gaining access to the terminal and the keyboard. Programs may be started in the foreground, using the terminal and keyboard and then suspended, only to be resumed later. Jobs may also run in the background, taking their input from a file and outputting either to the terminal or to a file. If a job in the background requests input from the terminal, it is suspended and the user is notified. Under Unix, jobs may be easily moved from foreground to background, and vice versa, at any point during their execution.

Running a program in the foreground is something that you are already familiar with, since that is the way all programs are run, unless otherwise specified. Running a job in the background requires some action on your part. The simplest way to run a job in the background is to append the symbol & to the end of the command. The command will then run in the background, allowing you to continue with other things while it is running. This is particularly useful for potentially long-running commands that require little or no interaction with you. It is often useful to send the output from a background command to a file. Once the command is complete, you will be notified.

```
% ls -Rl > filelist \&
[1] 2108
%
[1]    Done                 ls -Rl > filelist
```

When a job is put into the background, it is given a number that can be used to identify it. These numbers are small integers, starting with 1. When a job goes to the background, it is given the next available number. This is the number shown in square brackets above. The number shown next to the square brackets is the process id (PID). This is a number that uniquely identifies a process that is running to the Unix operating system. At any time, many processes are running and these may be identified and manipulated using the process id.

The `jobs` command and the `ps` commands can be used to track what jobs and processes are running. `jobs` tells you what commands are running in the background, while `ps` tells you all of the processes that are running. The listing from `ps` may look a little imposing, with many mysterious processes in evidence. Don't worry – most of them are supposed to be there are and are part of the everyday operation of Unix.

In the following example, we will start some long running processes so that we can watch them. The first, using `find` will find all files on the system named `core` and print out their relative path. We are redirecting the output from this command to a file. The `nroff` command is used to format one of the manual pages, this one for the `tn3270` program, for display. The output from nroff is then sent to the `ul` program which reformats underscoring for the terminal. We're sending the output from this to a file as well, so that we can look at the manual page later on. Then, we use the `jobs` and `ps` commands to see what is running.

```
% find / -name core -print > /tmp/corefiles &
[1] 2121
%nroff -man /usr/man/manl/tn3270.l | ul > /tmp/tn3270.cat &
[2] 2129 2130
% jobs
[1]  + Running              find / -name core > /tmp/corefiles
[2]  - Running              nroff -man /usr/man/manl/tn3270.l | ul > /tmp/tn3270.cat
% ps
  PID TT STAT  TIME COMMAND
 1328 p1 S     0:02 - (csh)
 1526 p2 S     0:09 - (csh)
 2121 p2 U     0:17 find / -name core
 2129 p2 U     0:04 nroff -man /usr/man/manl/tn3270.l
 2130 p2 S     0:00 ul
%
[2]    Done                 nroff -man /usr/man/manl/tn3270.l | ul > /tmp/tn3270.cat
```

Jobs can be interrupted by typing the interrupt key, usually Control Z (written C-Z or $\hat{Z}$). This will stop a process, but not kill it. It may then be resumed in either the foreground or the background with either the `fg` or `bg` command and its job number.

```
% find / -user dickson > /tmp/my_files
\^Z
Stopped
% jobs
[1]  + Stopped              find / -user dickson > /tmp/my\_files
% bg %1
[1]    find / -user dickson > /tmp/my\_files &
% jobs
[1]    Running              find / -user dickson > /tmp/my\_files
% kill %1
[1]    Terminated           find / -user dickson > /tmp/my\_files
```

Jobs may also be killed or sent signals using the `kill` command. Generally `kill` is used to kill a process, but it can also signal other conditions to the process. For a better understanding of `kill`, see the man page. Sometimes a process will not die with just a regular kill. When this is the case, use the modifer -KILL to kill instead.

```
% kill -KILL %1
```

9 Some Other Commonly Used Commands

We have only begun to brush the surface of Unix and its utilities. There are literally hundreds of programs that come with a standard Unix system, plus thousands of others available freely on the various bulletin boards. We have covered some of the basic concepts and basic commands, but have left out many of the important utilities. The following is a list of some of the more important commands. It is well worth your while to use the manual pages online and read about these commands. Space and time do not allow much exposition on any of these, but they are used heavily.

`more`	Display files one screen at a time.
`pr`	Add a title line and paginate files for printing.
`users`	Show users on the system
`who`	Like `users`, but with more information
`w`	Like `who`, but with more information.
`find`	Finds files that meet given criteria on the system
`lpr`	Print files
`grep`	Match patterns in files
`diff`	Show differences between files
`tar`	Create archives
`compress`	Compress files
`uncompress`	Uncompress files
`du`	Monitor disk usage