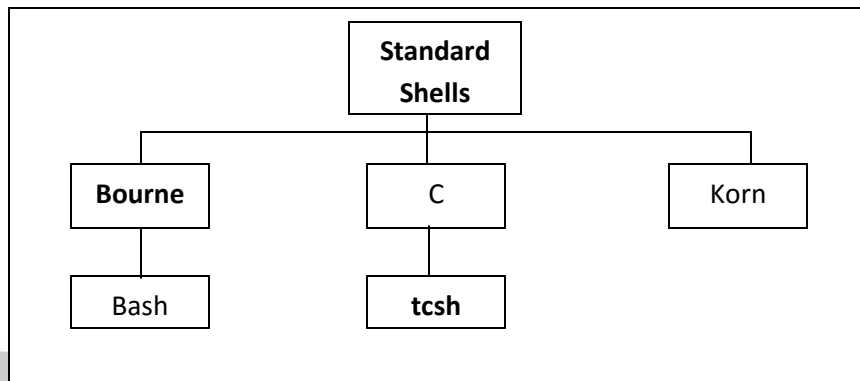


SHELLS:

The shell is the part of the UNIX that is most visible to the user. It receives and interprets the commands entered by the user. In many respects, this makes it the most important component of the UNIX structure.

To do anything in the system, we should give the shell a command. If the command requires a utility, the shell requests that the kernel execute the utility. If the command requires an application program, the shell requests that it be run. The standard shells are of different types as shown below:



There are two major parts to a shell. The first is the interpreter. The interpreter reads your commands and works with the kernel to execute them. The second part of the shell is a programming capability that allows you to write a shell (command) script.

A **shell script** is a file that contains shell commands that perform a useful function. It is also known as shell program.

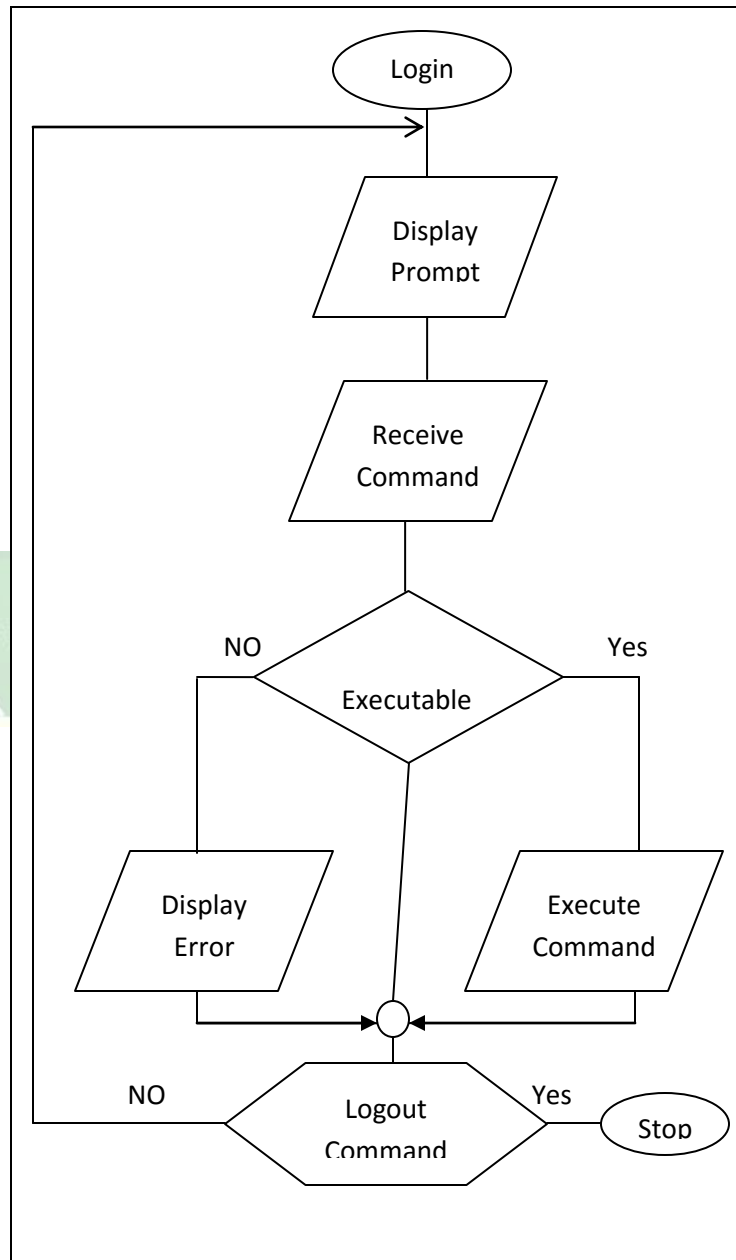
Three additional shells are used in UNIX today. The Bourne shell, developed by Steve Bourne at the AT&T labs, is the oldest. Because it is the oldest and most primitive, it is not used on many systems today. An enhanced version of Bourne shell, called Bash (Bourne again shell), is used in Linux.

The C shell, developed in Berkeley by Bill Joy, received its name from the fact that its commands were supposed to look like C statements. A compatible version of C shell, called **tcsh** is used in Linux.

The Korn shell, developed by David Korn also of the AT&T labs, is the newest and most powerful. Because it was developed at AT&T labs, it is compatible with the Bourne shell.

UNIX SESSION:

A UNIX session consists of logging in to the system and then executing commands to accomplish our work. When our work is done, we log out of the system. This work flow is shown in the following flowchart:



When you log in you are in one of the five shells. The system administrator determines which shell you start in by an entry in the password file (/etc/passwd). Even though your start up shell is determined by the system administrator, you can always switch to another shell. The following example shows how to move to other shells:

```
$ bash      # Move to Bash shell
$ ksh       # Move to Korn shell
$ csh       # Move to C shell
```

LOGIN SHELL VERIFICATION:

UNIX contains a system variable, SHELL that identifies the path to your login shell. You can check it with the command as follows:

```
$ echo $SHELL
/bin/ksh
```

Note: the variable name is all uppercase.

CURRENT SHELL VERIFICATION:

Your current shell may or may not be your login shell. To determine what your current shell is, you can use the following command. Note, however that this command works only with the Korn and Bash shells; it does not work with the C shell.

SHELL RELATIONSHIPS:

When you move from one shell to another, UNIX remembers the path you followed by creating a parent-child relationship. Your login shell is always the most senior shell in the relationship – the parent or grandparent depending on how many shells you have used.

Let us assume that your login shell is Korn shell. If you then move to the Bash shell, the Korn shell is the parent and Bash shell is the child. If later in the session you move to the C shell, the C shell is the child of Bash shell and the Bash shell is the child of Korn shell.

To move from child shell to a parent shell we use the **exit** command. When we move up to parent shell, the child shell is destroyed – it no longer exists. Should you create a child, an entirely new shell is created.

LOGOUT:

To quit the session – that is, to log out of the system – you must be at the original login shell. You cannot log out from a child. If you try to log out from a child, you will get an error message. The Korn shell and Bash shell both display a not-found message such as “logout not found”. The C shell is more specific: it reports that you are not in login shell.

The correct command to end the session at the login shell is **logout**, but the **exit** command also terminates the session.

STANDARD STREAMS:

UNIX defines three standard streams that are used by commands. Each command takes its input from a stream known as **standard input**. Commands that create output send it to a stream known as **standard output**. If an executing command encounters an error, the error message is sent to **standard error**. The standard streams are referenced by assigning a descriptor to each stream. The descriptor for standard input is 0 (zero), for standard input is 1, and for standard output is 2.

There is a default physical file associated with each stream: standard input is associated with the keyboard, standard output is associated with monitor and standard error is also associated with the monitor. We can change the default file association using pipes or redirection.

REDIRECTION:

It is the process by which we specify that a file is to be used in place of one of the standard files. With input files, we call it input redirection; with output files, we call it as output redirection; and with error file, we call it as error redirection.

Redirecting Input: we can redirect the standard input from the keyboard to any text file. The input redirection operator is the less than character (<). Think of it as an arrow pointing to a command, meaning that the command is to get its input from the designated file. There are two ways to redirect the input as shown below:

command 0< file1 or *command < file1*

The first method explicitly specifies that the redirection is applied to standard input by coding the 0 descriptor. The second method omits the descriptor. Because there is only-one standard input, we can omit it. Also note that there is no space between the descriptor and the redirection symbol.

Redirecting Output:

When we redirect standard output, the commands output is copied to a file rather than displayed on the monitor. The concept of redirected output appears as below:

command 1> file1 or *command > file1*

command 1>| file1 or *command >| file1*

command 1>> file1 or *command >> file1*

There are two basic redirection operators for standard output. Both start with the greater than character (>). Think of the greater than character as an arrow pointing away from the command and to the file that is to receive the output.

Which of the operators you use depends on how you want to output the file handled. If you want the file to contain only the output from this execution of the command, you use one greater than token (>). In this case when you redirect the output to a file that does not exist, UNIX creates it and writes the output.

If the file already exists the action depends on the setting of a UNIX option known as **noclobber**. When the noclobber option is turned on, it prevents redirected output from destroying an existing file. In this case you get an error message which is as given in below example:

```
$ who > whoOct2
ksh: whoOct2: file already exists
```

If you want to override the option and replace the current file's contents with new output, you must use the redirection override operator, greater than bar (>|). In this case, UNIX first empties the file and then writes the new output to the file. The redirection override output is as shown in the below example:

```
$ who > whoOct2
$ more whoOct2
abu52408   ttyq3  Oct   2   15:24 (atc2west-171.atc.fhda.edu)
```

On the other hand if you want to append the output to the file, the redirection token is two greater than characters (>>). Think of the first greater than as saying you want to redirect the output and the second one as saying that you want to go to the end of the file before you start outputting.

When you append output, if the file doesn't exist, UNIX creates it and writes the output. If it is already exists, however, UNIX moves to the end of the file before writing any new output.

Redirecting errors:

One of the difficulties with the standard error stream is that it is, by default, combined with the standard output stream on the monitor. In the following example we use the long list (ls) command to display the permissions of two files. If both are valid, one displays after the other. If only one is valid, it is displayed but *ls* display an error message for the other one on the same monitor.

```
$ ls -l file1 noFile
```

```
Cannot access noFile: No such file or directory
```

```
-rw-r--r-- 1 gilberg staff 1234 Oct 2 18:16 file1
```

We can redirect the standard output to a file and leave the standard error file assigned to the monitor.

REDIRECTING TO DIFFERENT FILES:

To redirect to different files, we must use the stream descriptors. Actually when we use only one greater than sign, the system assumes that we are redirecting the output (descriptor 1). To redirect them both, therefore, we specify the descriptor (0, 1, or 2) and then the redirection operator as shown in below example:

```
$ ls -l file noFile 1> myStdOut 2> myStdErr
```

```
$ more myStdOut
```

```
-rw-r--r-- 1 gilberg staff 1234 Oct 2 18:16 file1
```

```
$ more myStdErr
```

```
Cannot open noFile: No such file or directory
```

The descriptor and the redirection operator must be written as consecutive characters; there can be no space between them. It makes no difference which one you specify first.

REDIRECTING TO ONE FILE:

If we want both outputs to be written to the same file, we cannot simply specify the file name twice. If we do, the command fails because the file is already open. This is the case as given in the following example:

```
$ ls -l file1 noFile 1> myStdOut 2> myStdOut
```

```
ksh: myStdOut: file already exists
```

If we use redirection override operator, the output file contains only the results of the last command output which is as given below:

```
$ ls -l file1 noFile 1> | myStdOut 2> | myStdOut
```

```
$ ls myStdOut
```

```
Cannot open file noFile: No such file or directory
```

To write all output to the same file, we must tell UNIX that the second file is really the same as the first. We do this with another operator called **and** operator (&). An example of the substitution operator is shown as follows:

```
$ ls -l file1 noFile 1> myStdOut 2>& 1
```

```
$ more myStdOut
```

```
Cannot open file noFile: No such file or directory
```

```
-rw-r--r-- 1 gilberg staff 1234 Oct 2 18:16 file1
```

The following table shows the redirection differences between the shells:

Type	Korn and Bash Shells	C Shell
Input	<i>0< file1</i> or <i>< file1</i>	<i>< file1</i>
Output	<i>1> file1</i> or <i>> file1</i> <i>1> file1</i> or <i>> file1</i> <i>1>> file1</i> or <i>>> file1</i>	<i>> file1</i> <i>> file1</i> <i>>> file1</i>
Error	<i>2> file2</i> <i>2> file2</i> <i>2>> file2</i>	Not Supported Not Supported Not Supported
Output & Error (different files)	<i>1> file1 2> file2</i> <i>> file1 2> file2</i>	Not Supported Not Supported
Output & Error (same files)	<i>1> file1 2> & 1</i> <i>> file1 2> & 1</i> <i>1> file1 2> & 1</i>	<i>>& file1</i> <i>>& file1</i> <i>>&! file1</i>

PIPES:

We often need to use a series of commands to complete a task. For example if we need to see a list of users logged in to the system, we use the **who** command. However if we need a hard copy of the list, we need two commands. First we use **who** command to get the list and store the result in a file using redirection. We then use the **lpr** command to print the file. This sequence of commands is shown as follows:

```
who > file1
```

```
lpr file1
```

We can avoid the creation of the intermediate file by using a pipe. Pipe is an operator that temporarily saves the output of one command in a buffer that is being used at the same time as the input of the next command. The first command must be able to send its output to standard output; the second command must be able to read its input from standard input. This command sequence is given as follows:

```
$ who | lpr
```

Think of the pipe as a combination of a monitor and a keyboard. The input to the pipe operator must come from standard output. This means that the command on the left that sends output to the pipe must write its output to standard output.

A pipe operator receives its input from standard output and sends it to the next command through standard input. This means that the left command must be able to send data to standard output and the right command must be able to receive data from standard input.

The token for a pipe is the vertical bar (|). There is no standard location on the keyboard for the bar. Usually you will find it somewhere on the right side, often above the return key.

Pipe is an operator not a command. It tells the shell to immediately take the output of the first command, which must be sent to the standard output, and turn it into input for the second command, which must get its input from standard input.

TEE COMMAND:

The **tee** command copies standard input to a standard output and at the same time copies it to one or more files. The first copy goes to standard output, which is usually the monitor. At the same time, the output is sent to the optional files specified in the argument list.

The format of tee command is **tee options file-list**

The tee command creates the output files if they do not exist and overwrites them if they already exist. To prevent the files from being overwritten, we can use the option **-a**, which tells tee to append the output to existing files rather than deleting their current content.

Note however, that the append option does not apply to the standard output because standard output is always automatically appended. To verify the output to the file, we use more to copy it to the screen.

COMMAND EXECUTION:

Nothing happens in a UNIX shell until a command executed. When a user enters a command on the command line, the interpreter analyzes the command and directs its execution to a utility or other program.

Some commands are short and can be entered as a single line at the command prompt. We have seen several simple commands such as **cal** and **date**, already. At other times, we need to combine several commands.

There are four syntactical formats for combining commands in to one line: sequenced, grouped, chained, and conditional.

SEQUENCED COMMANDS:

A sequence of commands can be entered on one line. Each command must be separated from its predecessor by a semicolon. There is no direct relationship between the commands; that is one command does not communicate with the other. They simply combined in to one line and executed.

Example of a command sequence assumes that we want to create a calendar with a descriptive title. This is easily done with a sequence as shown below:

```
$ echo "\n Goblins & Ghosts\n      Month" > Oct2000; cal 10 2000 >> Oct2000
```

GROUPED COMMANDS:

We redirected the output of two commands to the same file. This technique gives us the intended results, but we can do it more easily by grouping the commands. When we group commands, we apply the same operation to the group. Commands are grouped by placing them in parentheses.

CHAINED COMMANDS:

In the previous two methods of combining commands into one line, there was no relationship between the commands. Each command operated independently of the other and only shared a need to have their output in one file. The third method of combining commands is to pipe them. In this case however, there is a direct relationship between the commands. The output of the first becomes the input of the second.

CONDITIONAL COMMANDS:

We can combine two or more commands using conditional relationships. There are two shell logical operators, *and* (&&) and *or* (||). In general when two commands are combined with a logical *and*, the second executes only if the first command is successful.

Conversely if two commands are combined using the logical *or*, the second command executes only if the first fails.

Example:

```
$ cp file1 tempfile && echo " Copy successful"
```

```
Copy successful
```

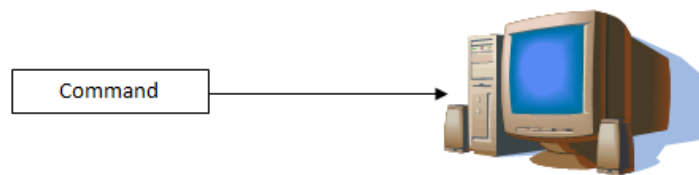
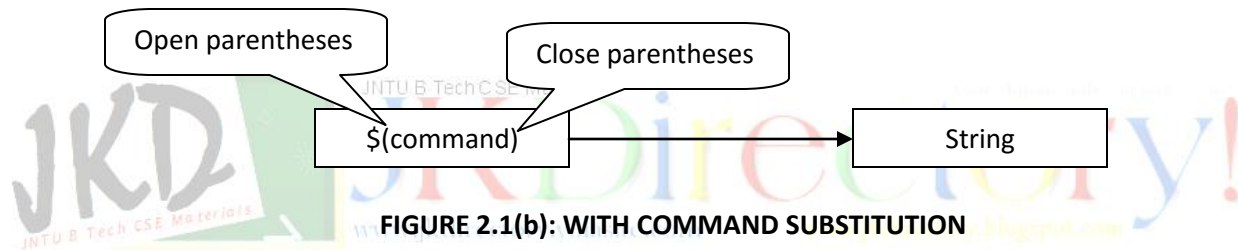
```
$ cp noFile tempfile || echo "Copy failed"
```

```
Copy failed
```

COMMAND SUBSTITUTION:

When a shell executes a command, the output is directed to standard output. Most of the time standard output is associated with the monitor. There are times, however such as when we write complex commands or scripts, which we need to change the output to a string that we can store in another string or a variable.

Command substitution provides the capability to convert the result of a command to a string. The command substitution operator that converts the output of a command to a string is a dollar sign and a set of parentheses (Figure 2.1).

**FIGURE 2.1(a): WITHOUT COMMAND SUBSTITUTION****FIGURE 2.1(b): WITH COMMAND SUBSTITUTION**

As shown in the figure 2.1, to invoke command substitution, we enclose the command in a set of parentheses preceded by a dollar sign (\$). When we use command substitution, the command is executed, and its output is created and then converted to a string of characters.

COMMAND LINE EDITING:

The phrase “to err is human” applies with a harsh reality to all forms of computing. As we enter commands in UNIX command line, it is very easy to err. As long as we haven’t keyed Return and we notice the mistake, we can correct it. But what if we have keyed Return?

There are two ways we can edit previous commands in the Korn and Bash shells and one way in C shell.

In the Korn and Bash shells we can use the history file or we can use command-line editing. The history file is a special UNIX file that contains a list of commands use during a session. In the C shell, we can use only the history file. The following table summarizes the command line editing options available.

Method	Korn Shell	Bash Shell	C Shell
Command line	✓	✓	
History File	✓	✓	✓

COMMAND LINE EDITING CONCEPT:

As each command is entered on the command line, the Korn shell copies it to a special file. With command line editing, we can edit the commands using either vi or emacs without opening the file. It's as though the shell keeps the file in a buffer that provides instant access to our commands. Whenever a command completes, the shell moves to the next command line and waits for the next command.

EDITOR SELECTION:

The system administrator may set the default command line editor, most likely in */etc/profile*. You can switch it, however by setting it yourself. If you set it at command line, it is set for only the current session. If you add it to your login file, it will be changed every time you log in. During the session you can also change it from one to another and back whenever you like.

To set the editor, we use the **set** command with the editor as shown in the next example; you would use only one of the two commands.

```
$ set -o vi          # Turn on vi editor
$ set -o emacs      # Turn on emacs editor
```

VI COMMAND LINE EDITOR:

We cannot tell which editor we are using at the command line simply by examining the prompt. Both editors return to the shell command line and wait for response. If quickly becomes obvious however, by the behavior of the command line.

The vi command line editor opens in the insert mode. This allows us to enter commands easily. When we key Return, the command is executed and we return to the vi insert mode waiting for input.

VI EDIT MODE

Remember that the vi editor treats the history file as though it is always open and available. Because vi starts in the insert mode, however to move to the vi command mode we must use the Escape key. Once in the vi command mode, we can use several of the standard vi commands. The most obvious commands that are not available are the read, write and quit commands.

The basic commands that are available are listed in the table below:

Category	Command	Description
Adding Text	I	Inserts text before the current character.
	l	Inserts text at the beginning of the current line.
	a	Appends text after the current character.
	A	Appends text at the end of the current line.
Deleting Text	X	Deletes the current character.
	Dd	Deletes the command line.
Moving Cursor	H	Moves the cursor one character to the left.
	l	Moves the cursor one character to the right.
	O	Moves the cursor to the beginning of the current line.
	\$	Moves the cursor to the end of the current line.
	k	Moves the cursor one line up.
	j	Moves the cursor one line down.
	-	Moves the cursor to the beginning of the previous line.
+	Moves the cursor to the beginning of the next line.	
Undo	u	Undoes only the last edit.
	U	Undoes all changes on the current line.
Mode	<esc>	Enters command mode.
	i, l, a, A	Enters insert mode.

Only one line is displayed at any time. Any changes made to the line do not change the previous line in the file. However, when the edited line is executed, it is appended to the end of the file.

USING THE COMMAND LINE EDITOR:

There are several techniques for using the command line editor.

Execute a Previous Line: To execute a previous line, we must move up the history file until we find the line. This requires the following steps:

1. Move to command mode by keying Escape (esc).
2. Move up the list using the Move-up key (k)
3. When the command has been located, key Return to execute it.

After the command has been executed, we are back at the bottom of the history file in the insert mode.

Edit and Execute a Previous Command: Assume that we have just executed the **more** command with a misspelled file name. In the next example, we left off the last character of the filename, such as "file" rather than "file1".

1. Move to the command mode by keying Escape (esc).
2. Using the Move-up key (k), recall the previous line.
3. Use the append-at-end command (A) to move the cursor to the end of the line in the insert mode.
4. Key the missing character and Return.

After executing the line, we are in the insert mode.

JOB CONTROL:

One of the important features of a shell is job control.

Jobs:

In general a job is a user task run on the computer. Editing, sorting and reading mail are all examples of jobs. However UNIX has a specific definition of a job. *A job is a command or set of commands entered on one command line.* For example:



Foreground and Background Jobs:

Because UNIX is a multitasking operating system, we can run more than one job at a time. However we start a job in the foreground, the standard input and output are locked. They are available exclusively to the current job until it completes. This means only one job that needs these files can run at a time. To allow multiple jobs, therefore, UNIX defines two types of jobs: foreground and background.

FOREGROUND JOBS:

A foreground job is any job run under the active supervision of the user. It is started by the user and may interact with the user through standard input and output. While it is running, no other jobs may be started. To start a foreground job, we simply enter a command and key Return. Keying Return at the end of the command starts it in the foreground.

Suspending a foreground job While a foreground job is running it can be suspended. For example, while you are running a long sort in the foreground, you get a notice that you have mail.

To read and respond to your mail, you must suspend the job. After you are through the mail you can then restart the sort. To suspend the foreground job, key ctrl+z. To resume it, use the foreground command (**fg**).

Terminating a foreground job If for any reason we want to terminate (kill) a running foreground job, we use the cancel meta-character, *ctrl+c*. After the job is terminated, we key Return to activate the command line prompt. If the job has been suspended, it must first be resumed using the foreground command.

BACKGROUND JOBS:

When we know a job will take a long time, we may want to run it in the background. Jobs run in the background free the keyboard and monitor so that we may use them for other tasks like editing files and sending mail.

Note: Foreground and Background jobs share the keyboard and monitor.

Any messages send to the monitor by the background job will therefore be mingled with the messages from foreground job.

Suspending, Restarting and Terminating Background jobs To suspend the background job, we use the **stop** command. To restart it, we use the **bg** command. To terminate the background job, we use the **kill** command. All three commands require the job number, prefaced with a percent sign (%).

Example:

```
$ longjob.scr&
[1] 1795841
$ stop %1
[1] + 1795841 stopped (SIGSTOP) longjob.scr&
$ bg %1
[1] longjob.scr&
$ kill %1
[1] + Terminated longjob.scr&
```

Moving between Background and Foreground To move a job between the foreground and background, the job must be suspended. Once the job is suspended, we can move it from the suspended state to the background with the **bg** command. Because job is in the foreground, no job number is required. To move a background job to a foreground job, we use the **fg** command.

MULTIPLE BACKGROUND JOBS:

When multiple background jobs are running in the background, the job number is required on commands to identify which job we want to affect.

Jobs command

To list the current jobs and their status, we use the **jobs** command. This command lists all jobs. Whether or not they are running or stopped. For each job, it shows the job number, currency, and status, running or stopped.

JOB STATES:

At any time the job may be in one of the three states: foreground, background or stopped. When a job starts, it runs the foreground. While it is running, the user can stop it, terminate it, or let it run to completion. The user can restart a stopped job by moving it to either the foreground or background state. The user can also terminate a job. A terminated job no longer exists. To be terminated, a job must be running.

While job is running it may complete or exit. A job that completes has successfully finished its assigned tasks. A job that exits has determined that it cannot complete its assigned tasks but also cannot continue running because some internal status has made completion impossible. When a job terminates either because it is done or it must exit, it sets a status code that can be checked by the user. The following figure 2.2 summarizes the job states:

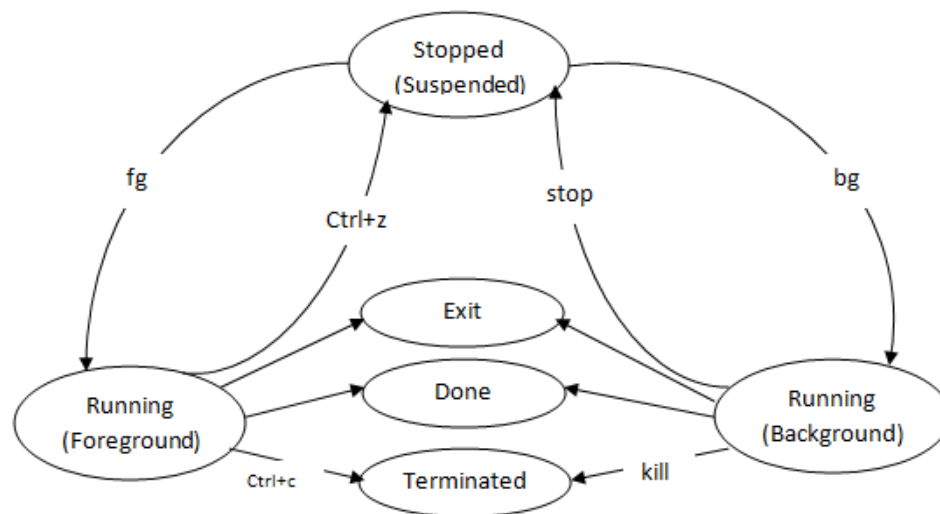


FIGURE 2.2 JOB STATES

Process ID

Job numbers are related to user session and the terminal; they are not global. UNIX assigns another identification, which is global in scope, to jobs or processes. It is called the process identifier, or PID. The **ps** command displays the current PIDs associated with the terminal which it is entered.

ALIASES:

An alias provides means of creating customized commands by assigning a name to a command. Aliases are handled differently in each shell.

Aliases in Korn and Bash Shells:

In the Korn and Bash shells, an alias is created by using the **alias** command. Its format is **alias name=command-definition** where alias is the command keyword, name is the alias being created, and command-definition is the code.

Example:

Renaming Commands One of the common uses of aliases is to create a more intuitive name for a command.

```

$ alias dir=ls
$ dir
TheRaven  file1 longJob.scr
TheRaven1 fileOut  loop.scr

```

Alias of command with Options An even better example is the definition of a command with options. We define a directory command with the long list option.

Example:

```

$ alias dir=`ls -l`
$ dir
Total 6
-rw- - - - - 1 gilberg staff 5782 10 16:19 TheRaven
....
-rw-r- - r - - 1 gilberg staff 149 Apr 18 2000 loop.scr

```


Alias of Multiple Command Lines Often a process requires more than one command. As long as the commands are considered one line in UNIX, they can be assigned as alias. Because some list output can be larger one screen of output, it is good idea to pipe the output to **more**.

```
$ alias dir="ls -l | more"
```

```
$ dir
```

```
Total 6
```

```
-rw- - - - - 1      gilberg staff  5782  10    16:19  TheRaven
```

```
....
```

```
-rw-r- - r - - 1      gilberg staff   149   Apr 18 2000  loop.scr
```

Using an Alias in and Alias Definition It is possible to create an alias definition using a definition. There is one danger in this usage; however, if a definition uses multiple aliases and one of them refers to another one, the definition may become recursive and it will bring down the shell. For this reason, it is recommended that only one definition can be used in defining a new alias definition.

Example:

```
$ alias dir=ls
```

```
$ alias Indir=`dir -l | more`
```

```
$ Indir
```

```
Total 6
```

```
-rw- - - - - 1      gilberg staff  5782  10    16:19  TheRaven
```

```
....
```

```
-rw-r- - r - - 1      gilberg staff   149   Apr 18 2000  loop.scr
```

ARGUMENTS TO ALIAS COMMANDS:

An argument may be passed to an alias as long as it is not ambiguous. Arguments are added after the command. The arguments may contain wildcards as appropriate. For example, we can pass a file to the list command so that it lists only the file(s) requested.

```
$ alias fl="ls -l"
```

```
$ fl f*
```

```
+ ls -l fgLoop.scr      file1  fileOut
-rwx- - - - - 1      gilberg      staff  175   13    10:38 fgLoop.scr
-rw-r - - r - - 1      gilberg      staff  15    17    2000  file1
-rw-r - - r - - 1      gilberg      staff  395   9     20:00 fileOut
```

Expanded commands are displayed starting with a plus followed by the command with its arguments filled in. Sometimes arguments can be ambiguous. This usually happens when multiple commands are included in one alias.

Listing aliases:

The Korn and Bash shells provide a method to list all aliases and to list a specific alias. Both use the alias command. To list all aliases, we use the alias command with no arguments. To list a specific command, we use the alias command with one argument, the name of the alias command. These variations are shown below:

\$ alias

autoload = 'typeset -fu'

cat = /sbin/cat

dir = 'echo '\ 'listing for gilberg'\ ' ; ls -l | more'

fl = 'ls -l | more'

...

Stop = 'kill -STOP'

suspend = 'kill -STOP \$\$'

\$ alias dir

dir = 'echo '\ 'listing for gilberg'\ ' ; ls -l | more'

Removing Aliases:

Aliases are removed by using the **unalias** command. It has one argument, a list of aliases to be removed. When it is used with the all option (-a), it deletes all aliases. You should be very careful, however with this option: It deletes all aliases, even those defined by the system administrator. For this reason, some system administrators disable this option. The following example demonstrates only the remove name argument.

\$ alias dir

dir = 'echo '\ 'listing for gilberg'\ ' ; ls -l | more'

\$ unalias dir

\$ alias dir

dir: alias not found

Aliases in the C Shell:

C shell aliases differ from Korn shell aliases in format but not in function. They also have more powerful set of features, especially for argument definition. The syntax for defining a C shell alias differs slightly in that there is no assignment operator. The basic format is:

alias name definition

Example:

```
% alias dir "echo Gilbergs Directory List; ls -l | more"
% dir
Gilbergs Directory List
total 30
-rw- - - - - 1    gilberg    staff  5782  Sep   10   16:19  TheRaven
...
-rw- r - - r - - 1    gilberg    staff   149   Apr   18   2000   teeOut1
-rw- r - - r - - 1    gilberg    staff   149   Apr   18   2000   teeOut2
```

Arguments to Alias Command:

Unlike Korn shell arguments are positioned at the end of the generated command, the C shell allows us to control the positioning. The following table contains the position designators used for alias arguments.

Designator	Meaning
\!*	Position of the only argument.
\!^	Position of the first argument.
\!\$	Position of the last argument.
\!:n	Position of the n th argument.

When we wrote the file list alias in the Korn shell, the argument was positioned at the end, where it caused a problem.

Listing Aliases Just like the Korn shell, we can list a specific alias or all aliases. The syntax for the two shells is identical.

Example:

```
% alias
cpto cp    \!:1  \!:$
dir  echo  Gilbergs Directory List; ls -l | more
f1   ls -l \!* | more
```

Removing Aliases:

The C shell uses the **unalias** command to remove one or all aliases. The following example shows the unalias command in C shell:

```
% unalias f1
% alias
cpto cp \!:1 \!:$
dir echo Gilbergs Directory List; ls -l | more
```

The following table summarizes the use of aliases in the three shells.

Feature	Korn and Bash	C
Define	\$ alias x=command	% alias x command
Argument	Only at the end	Anywhere
List	\$ alias	% alias
Remove	\$ unalias x y z	% unalias x y z
Remove All	\$ unalias -a	% unalias *

VARIABLES TYPES AND OPTIONS:**VARIABLES:**

A variable is a location in memory where values can be stored. Each shell allows us to **create**, **store** and **access** values in variables. Each shell variable must have a name. The name of a variable must start with an alphabetic or underscore (`_`) character. It then can be followed by zero or more alphanumeric or underscore characters.

There are two broad classifications of variables: (TYPES)

1. User-Defined
2. Predefined

User-Defined Variables:

User variables are not separately defined in UNIX. The first reference to a variable establishes it. The syntax for storing values in variables is the same for the Korn and Bash shells, but it is different for the C shell.

Predefined Variables:

Predefined variables are used to configure a user's shell environment. For example a system variable determines which editor is used to edit the command history. Other systems variables store information about the home directory.

Storing Data in Variables:

All three shells provide a means to store values in variables. Unfortunately, the C shell's method is different. The basic commands for each shell are presented in the table below:

Action	Korn and Bash	C Shell
Assignment	variable=value	set variable = value
Reference	\$variable	%variable

Storing Data in the Korn and Bash Shells:

The Korn and Bash shells use the assignment operator, = , to store values in a variable. Much like an algebraic expression, the variable is coded first, on the left, followed by the assignment operator and then the value to be stored. There can be no spaces before and after the assignment operator; the variable, the operator, and the value must be coded in sequence immediately next to each other as in the following example:

varA=7

In the above example varA is the variable that receives the data and 7 is the value being stored in it. While the receiving field must always be a variable, the value may be a constant, the contents of another variable, or any expression that reduces to a single value. The following shows some examples of storing values in variables.

```

$ x=23
$ echo $x
23
$ x=Hello
$ echo $x
Hello
$ x="Go Don's"
$ echo $x
Go Don's

```

Storing Data in the C Shell:

To store the data in a variable in the C Shell, we use the **set** command. While there can be no spaces before and after the assignment operator in the Korn and Bash shells, C needs them. C also accepts the assignment without spaces before and after the assignment operator.

Example:

```
% set x = 23
```

```
% echo $x
23
% set x = hello
% echo $x
hello
```

Accessing a Variable:

To access the value of a variable, the name of the variable must be preceded by a dollar sign. We use the echo command to display the values. Example:

```
$ x=23
$ echo The variable x contains $x
The variable x contains 23
$ x=hello
$ echo The variable x contains $x
The variable x contains hello
```

Predefined Variables:

Predefined variables can be divided into two categories: shell variable and environment variables. The shell variables are used to customize the shell itself. The environment variables control the user environment and can be exported to subshells. The following table lists the common predefined variables. In C shell, the shell variables are in lowercase letters, and the corresponding environmental variables are in uppercase letters.

Korn and Bash	C ^a	Explanation
CDPATH	cdpath	Contains the search path for cd command when the directory argument is a relative pathname.
EDIROT ^b	EDITOR	Path name of the command line editor
ENV		Pathname of the environment file
HOME ^b	home (HOME)	Pathname for the home directory
PATH ^b	path (PATH)	Search path for commands.
PS1	prompt	Primary prompt, such as \$ and %
SHELL ^b	shell (SHELL)	Pathname of the login shell
TERM ^b	term (TERM)	Terminal type
TMOUT	autologout	Defines idle time, in seconds, before shell automatically logs you off.
VISUAL ^b	VISUAL	Pathname of the editor for command line editing. See EDITOR table entry.

^aShell variables are in lowercase; environmental variables are in uppercase

^bBoth a shell and an environmental variable.

CDPATH:

The CDPATH variable contains a list of pathnames separated by colons (:) as shown in the example below:

:\$HOME: /bin/usr/files

There are three paths in the preceding example. Because the path starts with a colon, the first directory is the current working directory. The second directory is our home directory. The third directory is an absolute pathname to a directory of files.

The contents of CDPATH are used by the **cd** command using the following rules:

1. If CDPATH is not defined, the **cd** command searches the working directory to locate the requested directory. If the requested directory is found, **cd** moves to it. If it is not found, **cd** displays an error message.
2. If CDPATH is defined as shown in the previous example, the actions listed below are taken when the following command is executed:

\$ cd reports

- a. The **cd** command searches the current directory for the *reports* directory. If it is found, the current directory is changed to *reports*.
- b. If the *reports* directory is not found in the current directory, **cd** tries to find it in the home directory, which is the second entry in CDPATH. Note that the home directory may be the current directory. Again if the *reports* directory is found in the home directory, it becomes the current directory.
- c. If the *reports* directory is not found in the home directory, **cd** tries to find in */bin/usr/files*. This is the third entry in CDPATH. If the *reports* directory is found in */bin/usr/files*, it becomes the current directory.
- d. If the *reports* directory is not found in */bin/usr/files*, **cd** displays an error message and terminates.

HOME:

The HOME variable contains the PATH to your home directory. The default is your login directory. Some commands use the value of this variable when they need the PATH to your home directory. For example, when you use the **cd** command without any argument, the command uses the value of the HOME variable as the argument. You can change its value, but we do not recommend you change it because it will affect all the commands and scripts that use it. The following example demonstrates how it can be changed to the current working directory. Note that because **pwd** is a command, it must be enclosed in back quotes.

```
$ echo $HOME
/mnt/diska/staff/gilberg
$ oldHOME=$HOME
$ echo $oldHOME
/mnt/diska/staff/gilberg
```

```
$ HOME=$(pwd)
$ echo $HOME
/mnt/diska/staff/gilberg/unix13bash
```

```
$ HOME=$oldHOME
$ echo $HOME
/mnt/diska/staff/gilberg
```

PATH

The PATH variable is used to search for a command directory. The entries in the PATH variable must be separated by colons. PATH works just like CDPATH.

When the SHELL encounters a command, it uses the entries in the PATH variable to search for the command under each directory in the PATH variable. The major difference is that for security reasons, such as Trojan horse virus, we should have the current directory last.

If we were to set the PATH variable as shown in the below example, the shell would look for the **date** command by first searching the /bin directory, followed by the /usr/bin directory, and finally the current working directory.

```
$ PATH=/bin: /usr/bin: :
```

Primary Prompt (PS1 Prompt)

The primary prompt is set in the variable *PS1* for the Korn and Bash shells and *prompt* for the C shell. The shell uses the primary prompt when it expects a command. The default is the dollar sign (\$) for the Korn and Bash shells and the percent (%) sign for the C shell.

We can change the value of the prompt as in the example below:

```
$ PS1="KSH> "
KSH> echo $PS1
KSH>
KSH> PS1="$ "
$
```


SHELL

The SHELL variable holds the path of your login shell.

TERM

The TERM variable holds the description for the terminal you are using. The value of this variable can be used by interactive commands such as **vi** or **emacs**. You can test the value of this variable or reset it.

Handling Variables:

We need to set, unset, and display the variables. Table below shows how this can be done for each shell.

Operation	Korn and Bash	C Shell
Set	var=value	set var = value (setenv var value)
Unset	unset var	unset var (unsetenv var)
Display One	echo \$var	echo \$var
Display All	set	set (setenv)

Korn and Bash Shells:

Setting and Unsetting: In the Korn and Bash shells, variables are set using the assignment operator as shown in the example:

```
$ TERM=vt100
```

To unset a variable, we use the **unset** command. The following example shows how we can unset the TERM variable.

```
$ unset TERM
```

Displaying variables: To display the value of an individual variable, we use the **echo** command:

```
$ echo $TERM
```

To display the variables that are currently set, we use the **set** command with no arguments:

```
$ set
```

C Shell

The C Shell uses the different syntax for changing its shell and environmental variables.

Setting and Unsetting: To set a shell variable it uses the **set** command; to set the environmental variable it uses the **setenv** command. These two commands demonstrated in the example below:

```
$ set prompt = `CSH % `
```

```
CSH % setenv HOME /mnt/diska/staff/gilberg
```

To unset the C shell variable, we use the **unset** command. To unset an environmental variable we use the **unsetenv** command.

Example:

```
CSH % unset prompt
unsetenv EDITOR
```

Displaying Variables: To display the value of the individual variable (both shell and environmental), we use the **echo** command. To list the variables that are currently set, we use the **set** command without an argument for the shell variables and the **setenv** command without an argument for the environmental variables. These commands are shown in the example below:

```
% echo $variable-name # display one variable
% set # display all shell variables
% setenv # display all environmental variables
```

OPTIONS:

The following table shows the common options used for the three shells.

Korn and Bash	C	Explanation
Noglob	noglob	Disables wildcard expansion.
Verbose	verbose	Prints commands before executing them.
Xtrace		Prints commands and arguments before executing them.
Emacs		Uses emacs for command-line editing.
Ignoreeof	ignoreeof	Disallows ctrl+d to exit the shell.
Noclobber	noclobber	Does not allow redirection to clobber existing file.
Vi		Users vi for command-line editing.

Global (noglob): The global option controls the expansion of wildcard tokens in a command. For example, when the global option is off, the list file (**ls**) command uses wildcards to match the files in a directory.

Thus the following command lists all files that start with 'file' followed by one character:

```
$ ls file?
```

On the other hand when the global option is on, wildcards become text characters and are not expanded. In this case only the file names 'file?' would be listed.

Print Commands (verbose and xtrace): There are two print options, verbose and xtrace that are used to print commands before they are executed. The verbose option prints the command before it is executed. The xtrace option expands the command arguments before it prints the command.

Command line Editor (emacs and vi): To specify that the **emacs** editor is to be used in the Korn shell, we turn on the emacs option. To specify that the **vi** editor is to be used in the Korn shell, we turn on the vi option. Note that these options are valid only in the Korn Shell.

Ignore end of file (ignoreeof): Normally, if end of file (ctrl+d) is entered at the command line, the shell terminates. To disable this action we can turn on the ignore end of file option, ignoreeof. With this option, end of file generates an error message rather than terminating the shell.

No Clobber Redirection (noclobber): when output or errors are directed to a file that already exists, the current file is deleted and replaced by a new file. To prevent this action we set the noclobber option.

Handling Options: To customize our shell environment we need to set, unset and display options; the following table shows the appropriate commands for each shell.

Operation	Korn and Bash	C
Set	set -o option	set option
Unset	set +o option	unset option
Display All	set -o	Set

Korn and Bash Shell Options:

Setting and Unsetting Options: To set and unset an option, we use the **set** command with -o and +o followed by the option identifier. Using the Korn shell format, we would set and unset the verbose option, as shown in the following example:

```
$ set -o verbose      # Turn print commands option on
$ set -o verbose      # Turn print commands option off
```

Display Options: To show all of the options (set or unset), we use the `set` command with an argument of `-o`. This option requests a list of all option names with their state, on or off.

```
$ set -o          # Korn Shell format: lists all options
```

C Shell Options:

Setting and Unsetting Options: In C shell, options are set with the `set` command and unset with the `unset` command, but without the minus sign in both cases. They are both shown in the following example:

```
$ set verbose    # Turn print commands option on
$ unset verbose  # Turn print commands option off
```

Displaying Options:

To display which options are set, we use the `set` command without an argument. However the C shell displays the setting of all variables including the options that are variables. The options are recognized because there is no value assigned to them: Only their names are listed. The next example shows the display options format:

```
$ set          # C shell format: lists all variables
```

SHELL / ENVIRONMENT CUSTOMIZATION:

UNIX allows us to customize the shells and the environment we use. When we customize the environment, we can extend it to include subshells and programs that we create.

There are four elements to customizing the shell and the environment. Depending on how we establish them, they can be temporary or permanent. Temporary customization lasts only for the current session. When a new session is started, the original settings are reestablished.

Temporary Customization:

It can be used to change the shell environment and configuration for the complete current session or for only part of a session. Normally we customize our environment for only a part of the session, such as when we are working on something special.

For example if we are writing a script it is handy to see the expanded commands as they are executed. We would do this by turning on the verbose option. When we are through writing the script, we would turn off the verbose option.

Any option changed during this session is automatically reset to its default when we log on the next time.

Permanent Customization:

It is achieved through the startup and shutdown files. Startup files are system files that are used to customize the environment when a shell begins. We can add customization commands and set customization variables by adding commands to the startup file. Shutdown files are executed at logout time. Just like the startup files, we can add commands to clean up the environment when we log out.

Korn Shell:

The Korn shell uses the three profile files as described below:

System Profile File: There is one system level profile file, which is stored in the /etc directory. Maintained by the system administrator, it contains general commands and variable settings that are applied to every user of the system at login time. It is generally quiet large and contains many advanced commands. The system profile file is read-only file; its permissions are set so that only the system administrator can change it.

Personal Profile File: The personal profile, ~/.profile contains commands that are used to customize the startup shell. It is an optional file that is run immediately after the system profile file. Although it is a user file, it is often created by the system administrator to customize a new user's shell.

Environment File: In addition, the Korn shell has an environmental file that is run whenever a new shell is started. It contains environmental variables that are to be exported to subshells and programs that run under the shell.

The environment file does not have a predetermined name. We can give it any name we desire. It must be stored in home directory or in a subdirectory below the home directory. But it is recommended to store in the home directory.

To locate the environmental file, the Korn shell requires that it's absolute or relative pathname be stored in the predefined variable, ENV.

Bash Shell:

For the system profile file, the Bash shell uses the same file as the Korn shell (/etc/profile). However for the personal profile file, it uses one of the three files. First it looks for Bash profile file (~/.bash_profile). If it doesn't find a profile file, it looks for a login file (~/.bash_login).

If it does not find a login file, it looks for a generic profile file (`~/profile`). Whichever file the Bash finds, it executes it and ignores the rest. The Bash environmental file uses the same concept as the Korn shell, except that the filename is stored in the `BASH_ENV` variable.

The Korn shell does not have logout file, but the Bash shell does. When the shell terminates, it looks for the logout file (`~/bash_logout`) and executes it.

C Shell:

The C shell uses both startup and shutdown files: it has two startup files, `~/login` and `~/cshrc`, and one shutdown file, `~/logout`.

Login File: The C Shell login file (`~/login`) is the equivalent of the Korn and Bash user Profile file. It contains commands and variables that are executed when the user logs in. It is not exported to other shells nor is it executed if the C shell is started from another shell as a subshell.

Environmental File: The C shell equivalent of the environmental file is the `~/cshrc` file. It contains environmental settings that are to be exported to subshells. As an environmental file, it is executed whenever a new subshell is invoked.

Logout File: The C shell logout file `~/logout`, is run when we log out of the C shell. It contains commands and programs that are to be run at logout time.

Other C Shell Files: The C shell may have other system files that are executed at login and logout time. They found in the `/etc` directory as `/etc/csh.cshrc`, `/etc/csh.login` and `/etc/csh.logout`.

FILTERS AND PIPES – RELATED COMMANDS:

FILTERS:

In UNIX, a filter is any command that gets its input from the standard input stream, manipulates the input and then sends the result to the standard output stream. Some filters can receive data directly from a file.

We have already seen one filter, the **more** command. There are 12 more simple filters available. Three filters – **grep**, **sed** and **awk** – are so powerful. The following table summarizes the common filters:

FILTER	ACTION
more	Passes all data from input to output, with pauses at the end of the each screen of data.

FILTER	ACTION
cat	Passes all data from input to output.
cmp	Compares two files.
comm	Identifies common lines in two files.
cut	Passes only specified columns.
diff	Identifies differences between two files or between common files in two directories.
head	Passes the number of specified lines at the beginning of the data.
paste	Combines columns.
sort	Arranges the data in sequence.
tail	Passes the number of specified lines at the end of the data.
tr	Translates one or more characters as specified.
uniq	Deletes duplicate (repeated) lines.
wc	Counts characters, words, or lines.
grep	Passes only specified lines.
sed	Passes edited lines.
awk	Passes edited lines – parses lines.

FILTERS AND PIPES:

Filters work naturally with pipes. Because a filter can send its output to the monitor, it can be used on the left of a pipe; because a filter can receive its input from the keyboard, it can be used on the right of a pipe.

In other words a filter can be used on the left of a pipe, between two pipes, and on the right of the pipe. These relationships are presented in figure 2.3:

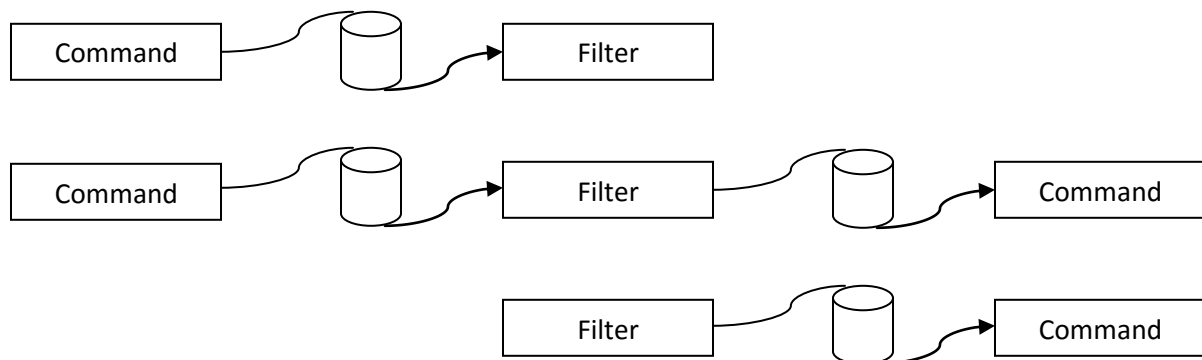
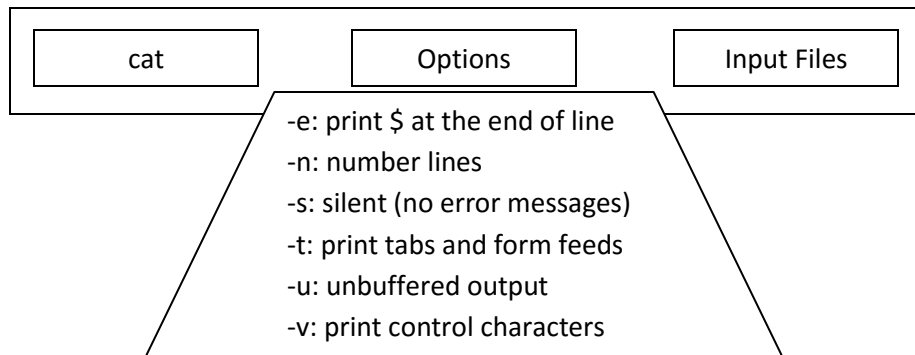


Figure 2.3: Using Filters and Pipes

CONCATENATING FILES:

UNIX provides a powerful utility to concatenate commands. It is known as the catenate command, or **cat** for short. It combines one or more files by appending them in the order they are listed in the command. The input can come from the keyboard; the output goes to the monitor.

The basic concept is shown as follows:



Catenate (cat) command:

Given one or more input files, the **cat** command writes them one after another to standard output. The result is that all of the input files are combined and become one output. If the output file is to be saved, standard output can be redirected to a specified output file. The basic **cat** command as shown below:

Example: In the below example if display the contents of three files each of which has only one line.

```
$ cat file1 file2 file3
```

```
This is file1.
This is file2.
This is file3.
```

Using cat to Display a File:

Its basic design makes **cat** a useful tool to display a file. When only one input is provided, the file is catenated with a null file. The result is that the input file is displayed on the monitor. The use of **cat** command to display a file is as shown below:

```
cat file1
```

The following example demonstrates the use of **cat** to display a file. The file, TheRavenV1 contains the first six lines of "TheRaven".

```
$ cat TheRavenV1
```


*Once up on a midnight dreary, while I pondered, weak and weary,
Over many a quaint and curious volume of forgotten lore
While I nodded, nearly napping, suddenly there came a tapping,
As of someone gently rapping, rapping at my chamber door.
‘Tis some visitor,’ I muttered, ‘tapping at my chamber door
Only this and nothing more’*

Using cat to create a File:

The second special application uses **cat** to create a file. Again there is only one input this time, however, the input comes from the keyboard. Because we want to save the file, we redirect standard output to a file rather than to the monitor.

Because all of the input is coming from keyboard, we need some way to tell the **cat** command that we have finished inputting data. In other words we need to tell the system that we have input all of the data and are at the end of the file. In UNIX, the keyboard command for **end of file** is the ctrl+d keys, usually abbreviated as ^d.

The following example demonstrates the **cat** command.

```
$ cat > goodStudents
```

```
Now is the time  
For all good students  
To come to the aid  
of their college.
```

cat Options:

There are six options available with **cat**. They can be grouped into four categories: visual characters, buffered output, missing files, and numbered lines.

Visual Characters: Sometimes when we display output, we need to see all of the characters. If the file contains unprintable characters, such as ASCII control characters, we can't see them. Another problem arises if there are space characters at the end of a line – we can't see them because they have no visual graphic.

The visual option **-v** allows us to see control characters, with the exception of the tab, new line, and form feed characters. Unfortunately the way they are printed is not intuitive and is beyond the scope of this text.

If we use the option **-ve** a dollar sign is printed at the end of the each line. If we use the option **-vt**, the tabs appear as **^I**. with both options nonprintable characters are prefixed with a caret (^).

Example:

```
$ cat -vet catExample
```

There is a tab between the numbers on the next line\$

```
1^I2^I3^I4^I5$
```

```
$
```

```
One two buckle my shoe$
```

Buffered Output:

When output is buffered, it is kept in the computer until the system has time to write it to a file. Normally **cat** output is buffered. You can force the output to be written to the file immediately by specifying the option **-u** for unbuffered. This will slows down the system

Missing Files:

When you catenate several files together, if one of them is missing, the system displays a message such as:



If you don't want to have this message in your output, you can specify that the **cat** is to be **silent** when it can't find the file. This option is **-s**.

Numbered Lines:

The numbered lines option (**-n**) numbers each line in each file as the line is written to standard output. If more than one file is being written, the numbering restarts with each file.

Example:

```
$ cat -n goodStudents catExample
```

```
1: Now is the time
```

```
2: For all good students
```

```
3: To come to the aid
```

```
4: of their college.
```

```
1: There is a tab between the numbers on the next line
```

```
2: 1^I2^I3^I4^I5
```

3:

4: One two buckle my shoe

DISPLAYING BEGINNING AND END OF FILES:

UNIX provides two commands, **head** and **tail** to display the portions of files.

head Command

While the **cat** command copies entire files, the **head** command copies a specified number of lines from the beginning of one or more files to the standard output stream. If no files are specified it gets the lines from standard input. The basic format is given below:

head options inputfiles . . .

The option **-N** is used to specify the number of lines. If the number of lines is omitted, head assumes 10 lines. If the number of lines is larger than the total number of lines in the file, the total file is used.

Example:

```
$ head -2 goodStudents
```

```
Now is the time
```

```
For all good students
```

When multiple files are included in one **head** command, **head** displays the name of file before its output.

Example:

```
$ head -2 goodStudents TheRaven
```

```
= => goodStudents <= =
```

```
Now is the time
```

```
For all good students
```

```
= => TheRaven <= =
```

```
Once up on a midnight dreary, while I pondered, weak and weary,
```

```
Over many a quaint and curious volume of forgotten lore
```

tail Command:

The tail command also outputs data, only this time from the end of the file. The general format of tail command is **tail options inputfile**

Although only file can be referenced (in most systems), it has several options as shown in the table below:

Option	Code	Description
Count from beginning	+N	Skip N-1 lines
Count from end	-N	N lines from end
Count by lines	-l	Measured by lines (default)
Count by characters	-c	Measured by characters
Count by blocks	-b	Measured by blocks
Reverse order	-r	Output in reverse order

Example:

```
$ tail -2r goodStudents
```

of their college.

To come to the aid

We can combine head and tail commands to extract lines from the center of a file.

Example:

```
$ head -1 TheRaven | tail +2
```

Cut and Paste:

In UNIX **cut** command removes the columns of data from a file, and **paste** combines columns of data.

Because the **cut** and **paste** commands work on columns, text files don't work well. Rather we need a data file that organizes data with several related elements on each line. To demonstrate the commands, we created a file that contains selected data on largest five cities in United States according to 1990 census which is as shown in the table below:

Chicago	IL	2783726	3005072	1434029
Houston	TX	1630553	1595138	1049300
Los Angeles	CA	3485398	2968528	1791011
New York	NY	7322564	7071639	3314000
Philadelphia	PA	1585577	1688510	1736895

cut Command:

The basic purpose of cut command is to extract one or more columns of data from either standard input or from one or more file. The format of **cut** command is as shown below:

cut options file list

Since **cut** command looks for columns, we have some way to specify where the columns are located. This is done with one of two command options. We can specify what we want to extract based on character positions with a line or by a field number.

Specifying Character Positions:

Character positions work well when the data are aligned in fixed columns. The data in above table are organized in this way. City is string of 15 characters state is 3 characters (including trailing space), 1990 population is 8 characters, 1980 population is 8 characters and work force is 8 characters.

To specify that file is formatted with fixed columns, we use the character option **-c** followed by one or more column specification. A **column specification** can be one column or a range of columns in the format N-M, where N is the start column and M is the end column, inclusively. Multiple columns are separated by commas.

Example:

```
$ cut -c1-14,19-25 censusFixed
```

```
Chicago 2783726
Houston 1630553
Los Angeles 3485398
New York 7322564
Philadelphia 1585577
```

Field Specification:

While the column specification works well when the data are organized around fixed columns, it doesn't work in other situations. In the table below the city name ranges between columns 1-7 and columns 1-12. Our only choice therefore is to use delimited fields. We have indicated the locations of the tabs with the notation <tab> and have spaced the data to show how it would be displayed.

Chicago	IL<tab>	2783726<tab>	3005072<tab>	1434029
Houston	TX<tab>	1630553<tab>	1595138<tab>	1049300
Los Angeles	CA<tab>	3485398<tab>	2968528<tab>	1791011
New York	NY<tab>	7322564<tab>	7071639<tab>	3314000
Philadelphia	PA<tab>	1585577<tab>	1688510<tab>	736895

When the data are separated by tabs, it is easier to use fields to extract the data from the file. Fields are separated from each other by a terminating character known as a **delimiter**. Any character may be a delimiter; however, if no delimiter **cut** assumes it as a tab character.

To specify a field we use the **field** option (-f). Fields are numbered from the beginning of the line with the first field being field number one. Like the character option, multiple fields are separated by commas with no space after the comma. Consecutive fields may be specified as a range.


The **cut** command assumes that the delimiter is a tab. If it is not we must specify it in the delimiter option. When the delimiter has special meaning to UNIX, it must be enclosed in quotes. Because the space terminates an option, therefore we must enclose it in quotes.

The options of cut command are shown in the table below:

Option	Code	Results
Character	-c	Extracts fixed columns specified by column number
Field	-f	Extracts delimited columns
Delimiter	-d	Specifies delimiter if not tab (default)
Suppress	-s	Suppresses output if no delimiter in line.

Paste Command:

The paste command combines lines together. It gets the input from two or more files. To specify that the input is coming from the standard input stream, you use a hyphen (-) instead of a filename. The paste command is represented as follows:



paste options file list

The paste combines the first line of the first file with the first line of the second file and writes the combined line to the standard output stream. Between the columns, it writes a tab. At the end of the last column, it writes a newline character. It then combines the next two lines and writes them, continuing until all the lines have been written to standard stream. In other words paste treats each line of each file as a column.

Note: The *cat* and *paste* command are similar: The *cat* command combines files vertically (by lines). The *paste* command combines files horizontally (by columns).

Sorting:

When dealing with data especially a lot of data we need to organize them for analysis and efficient processing. One of the simplest and most powerful organizing techniques is sorting. When we sort data we arrange them in sequence.

Usually we use ascending sequence, an arrangement in which each piece of data is larger than its predecessor. We can also sort in descending sequence, in which each piece of data is smaller than its predecessor.

sort Command:

The sort utility uses options, field specifiers, and input files. The field specifiers tell it which fields to use for the sort. The sort command format is shown below:

sort options field specifiers input files

Sort by lines:

The easiest sort arranges the data by lines. Starting at the beginning of the line, it compares the first character in one line with the first character in another line. If they are the same, it moves to the second character and compares them. This character-by-character comparison continues until either all character in both lines have compared equal or until two unequal characters are found.

If lines are not equal, comparison stops and sort determines which line should be first based on the two unequal characters. In comparing characters, sort uses the ASCII values of each character.

Field specifiers:

When a field sort is required, we need to define which field or fields are to be used for the sort. Field specifiers are a set of two numbers that together identify the first and last field in a sort key. They have the following format:

+number₁ –number₂

number₁ specifies the number of fields to be skipped to get to the beginning of the sort field, whereas number₂ specifies the number of fields to be skipped, relative to the beginning of the line to get to the end of the sort key.

TRANSLATING CHARACTERS:

There are many reasons for translating characters from one set to another. One of the most common is to convert lowercase characters to uppercase, or vice versa. UNIX provides translate utility making conversions from one set to another.

tr command: The tr command replaces each character in a user-specified set of characters with a corresponding character in a second specified set. Each set is specified as a string. The first character in the first set is replaced by the first character in the second set; the second character in the first set is replaced by the second character in the second set and so forth until all matching characters have been replaced. The strings are specified using quotes.

The tr command is represented as follows: **tr options string1 string2**

Simple Translate:

Translate receives its input from standard input and writes its output to standard output. If no options are specified, the text is matched against the string1 set, and any matching characters are replaced with the corresponding characters in the string2 set. Unmatched characters are unchanged.

```
$ tr "aeiou" "AEIOU"
```

```
It is very easy to use TRANSLATE. #input
```

```
It Is vEry EAsy tO Use TRANSLATE. #output
```

Nonmatching Translate Strings:

When the translate strings are of different length, the result depends on which string is shorter. If string2 is shorter, the unmatched characters will all be changed to the last character in string2. On the other hand, if string1 is shorter, the extra characters in string2 are ignored.

```
$ tr "aeiou" "AE?" #case 1: string2 is shorter than string1
```

```
It is very easy to use TRANSLATE.
```

```
It ?s vEry EAsy t? ?sE trAnslAtE.
```

```
$ tr "aei" "AEIOU?" #case 1: string1 is shorter than string2
```

```
It is very easy to use TRANSLATE.
```

```
It Is vEry EAsy to usE trAnslAtE.
```

Delete Characters:

To delete matching characters in the translation we use the delete option (-d). In the following example we delete all vowels, both upper and lowercase. Note that the delete option does not use string2.

```
$ tr -d "aeiouAEIOU"
```

```
It is very easy to use TRANSLATE
```

```
t s vry sy t s TRNSLT
```

Squeeze Output:

The squeeze option deletes consecutive occurrences of the same character in the output. **Example:**

```
$ tr -s "ie" "dd"
```


The fiend did dastardly deeds

Thd fdnd d dastardly ds

Complement:

The complement option reverses the meaning of the first string. Rather than specifying what characters are to be changed, it says what characters are not to be changed.

Example:

```
$ tr -c "aeiou" "*"

```

It is very easy to use TRANSLATE.

```
***j***e***ea****o*u*e*****
```

FILES WITH DUPLICATE LINES:

We used a sort to delete duplicate lines (words). If the lines are already adjacent to each other, we can use the unique utility.

uniq command:

The **uniq** command deletes duplicate lines, keeping the first and deleting the others. To be deleted, the lines must be adjacent. Duplicate lines that are not adjacent are not deleted. To delete nonadjacent lines the file must be sorted.

Unless otherwise specified the whole line can be used for comparison. Options provide for the compare to start with a specified field or character. The compare whether line, field, or character, is to the end of the line. It is not possible to compare one field in the middle of the line. The unique command is shown as follows:

uniq options inputfile

All of the examples use a file with three sets of duplicate lines. The complete file is shown below:

```
5 completely duplicate lines
5 completely duplicate lines
5 completely duplicate lines
5 completely duplicate lines
5 completely duplicate lines
Not a duplicate - - next duplicates first 5
5 completely duplicate lines
Last 3 fields duplicate: one two three
Last 3 fields duplicate: one two three
```

```

Last 3 fields duplicate: one two three
The next 3 lines are duplicate after char 5
abcde Duplicate to end
fghij Duplicate to end
klmno Duplicate to end

```

There are three options: output format, skip leading fields and skip leading characters.

Output Format:

There are four output formats: nonduplicated lines and the first line of each duplicate series (default), only unique lines (-u), only duplicated lines (-d), and show count of duplicated lines (-c).

Default Output Format:

We use unique command without any options. It writes all of the nonduplicated lines and the first of a series of duplicated lines. This is the default result.

\$ **uniq uniqFile**

```

5 completely duplicate lines
Not a duplicate - - next duplicates first 5
5 completely duplicate lines
Last 3 fields duplicate: one two three
The next 3 lines are duplicate after char 5
abcde Duplicate to end
fghij Duplicate to end
klmno Duplicate to end

```

Nonduplicated Lines (-u):

The nonduplicated lines option is -u. It suppresses the output of the duplicated lines and lists only the *unique lines in the file*. Its output is shown in the example below:

\$ **uniq -u uniqFile**

```

Not a duplicate - - next duplicates first 5
5 completely duplicate lines
The next 3 lines are duplicate after char 5
abcde Duplicate to end
fghij Duplicate to end
klmno Duplicate to end

```

Only Duplicated Lines (-d):

The opposite of nonduplicated lines is to write only the duplicated lines (-d). Its output is shown in the example below:

```
$ uniq -d uniqFile
```

```
5 completely duplicate lines
Last 3 fields duplicate: one two three
```

Count Duplicate Lines (-c):

The count duplicates option (-c) writes all of the lines, suppressing the duplicates, with a count of number duplicates at the beginning of the line.

```
$ uniq -c uniqFile
```

```
5 5 completely duplicate lines
1 Not a duplicate - - next duplicates first 5
1 5 completely duplicate lines
3 Last 3 fields duplicate: one two three
1 The next 3 lines are duplicate after char 5
1 abcde Duplicate to end
1 fghij Duplicate to end
1 klmno Duplicate to end
```

Skip Leading Fields:

While the default compares the whole line to determine if two lines are duplicate, we can also specify where the compare is to begin. The skip duplicate fields option (-f) skips the number of fields specified starting at the beginning of the line and any spaces between them. Remember that a field is defined as a series of ASCII characters separated by either a space or by a tab. Two consecutive spaces would be two fields; that's the reason **uniq** skips leading spaces between fields.

Example:

```
$ uniq -d -f 4 uniqFile
```

```
5 completely duplicate lines
Last 3 fields duplicate: one two three
abcde Duplicate to end
```

Skipping Leading Characters:

We can also specify the number of characters that are to be skipped before starting the compare. In the following example, note that the number of leading characters to be skipped is separated from the option (-s). This option is represented as in below example:

```
$ uniq -d -s 5 uniqFile
```

```
5 completely duplicate lines
Last 3 fields duplicate: one two three
abcde Duplicate to end
```

The unique options are represented as in the table below:

Option ^a	Code	Results
Unique	-u	Only unique lines are output.
Duplicate	-d	Only duplicate lines are output.
Count	-c	Outputs all lines with duplicate count.
Skip field	-f	Skips leading fields before duplicate test.
Skip characters	-s	Skips leading characters before duplicate test.

COUNT CHARACTERS, WORDS, OR LINES:

Many situations arise in which we need to know how many words or lines are in a file. Although not as common, there are also situations in which we need to know a character count. The UNIX word count utility handles these situations easily.

wc command:

The **wc** command counts the number of characters, words, and lines in one or more documents. The character count includes newlines (/n). Options can be used to limit the output to only one or two of the counts. The word count format is shown below:

wc options inputfiles

The following example demonstrates several common count combinations. The default is all three options (clw). If one option is specified, the other counts are not displayed. If two are specified the third count is not displayed.

```
$ wc TheRaven
```

```
116          994          5782 TheRaven
```

```
$ wc TheRaven uniqFile
```

```
116      994      5782 TheRaven
14       72      445  uniqFile
130     1066     6227 total
```

```
$ wc -c TheRaven
```

```
5782 TheRaven
```

```
$ wc -l TheRaven
```

```
116 TheRaven
```

```
$ wc -w TheRaven
```

```
116      5782 TheRaven
```

The following table shows the word count options:

Option	Code	Results
Character count	-c	Counts characters in each file.
Line count	-l	Counts number of lines in each file.
Word count	-w	Counts number of words in each file.

COMPARING FILES:

There are three UNIX commands that can be used to compare the contents of two files: compare (**cmp**), difference (**diff**), and common (**comm**).

Compare (cmp) Command:

The **cmp** command examines two files byte by byte. The action it takes depends on the option code used. Its operation is shown below:

```
cmp options file1 file2
```

cmp without Options:

When the **cmp** command is executed without any options, it stops at the first byte that is different. The byte number of the first difference is reported. The following example demonstrates the basic operation, first with two identical files and then with different files.

```
$ cat cmpFile1
```

```
123456
```

```
7890
```

```
$ cat cmpFile1.cpy
```

```
123456
```

```
7890
```

```
$ cmp cmpFile1 cmpFile1.cpy
```

```
$ cat cmpFile2
```

```
123456
```

```
As9u
```

```
$ cmp cmpFile1 cmpFile2
```

```
cmpFile1 cmpFile2 differ: char 8, line2
```

cmp with List Option (-l)

The list option displays all of the differences found in the files, byte by byte. A sample output is shown in the following example:

```
$ cmp -l cmpFile1 cmpFile2
```

```
8 67 141
```

```
9 70 163
```

```
11 60 165
```

```
cmp with suppress list option (-s):
```

The suppress list option (-s) is similar to the default except that no output is displayed. It is generally used when writing scripts. When no output is displayed, the results can be determined by testing the exit status. If the exit status is 0, the two files are identical. If it is 1, there is at least one byte that is different. To show the exit status, we use the **echo** command.

```
$ cmp cmpFile1 cmpFile1.cpy
```

```
$ echo $?
```

```
0
```

```
$ cmp cmpFile1 cmpFile2
```

```
$ echo $?
```

```
1
```

Difference (diff) Command:

The **diff** command shows the line-by-line difference between the two files. The first file is compared to the second file. The differences are identified such that the first file could be modified to make it match the second file.

The command format is shown below:

diff options files or directories

The diff command always works on files. The arguments can be two files, a file and directory, or two directories. When one file and one directory are specified, the utility looks for a file with the same name in the specified directory. If two directories are provided, all files with matching names in each directory are used. Each difference is displayed using the following format:

```
range1 action range2
< text from file1
- - -
> text from file2
```

The first line defines what should be done at range1 in file1 (the file identified by first argument) to make it match the lines at range2 in file2 (the file identified by second argument). If the range spans multiple lines, there will be a text entry for each line in the specified range. The action can be change (c), append (a), or delete (d).

Change (c) indicates what action should be taken to make file1 the same as file2. Note that a change is a delete and replace. The line(s) in range1 are replaced by the line(s) in range2.

Append (a) indicates what lines need to be added to file1 to make it the same as file2. Appends can take place only at the end of the file1; they occur only when file1 is shorter than file2.

Delete (d) indicates what lines must be deleted from file1 to make it the same as file2. Deletes can occur only if file1 is longer than file2.

Example: Interpretation

```
6c6          change: Replace line 6 in file1 with line 6 in file2
< hello
- - -
> greeting
```

```
25a26,27    append: At the end of fil1 (after line 25), insert 26 and 27 from file2.
> bye bye.  Note that for append, there is no separator (dash) and no file1 (<) lines.
> good bye.
```

```
78,79d77    delete: the extra lines at the end of file1 should be deleted.
< line 78   The text of the lines to be deleted is shown.
```

text Note again that there is no separator line
 < line 79 and, in this case no file2 (>) lines.
 text

Common (comm) Command:

The **comm** command finds lines that are identical in two files. It compares the files line by line and displays the results in three columns. The left column contains unique lines in file 1; the center column contains unique lines in file 2; and the right column contains lines found in both files. the command format is shown below:

comm options file1 file2

The files (comm1 and comm2) used to demonstrate the **comm** command is shown in the table given below:

comm1	comm2
one same	one same
two same	two same
different comm1	different comm2
same at line 4	same at line 4
same at line 5	same at line 5
not in comm2	
same at line 7	same at line 7
same at line 8	same at line 8
	not in comm1
last line same	last line same

The output for comm utility for these two files is shown below:

\$ comm comm1 comm2

```

one same
two same
different comm1
different comm2
same at line 4
same at line 5
not in comm2
same at line 7
same at line 8
not in comm1
last line same
    
```