

UNIX – Basic Commands

UNIX – Basic Commands

Introduction

Overview

- An Operating System (OS) is the software that manages the sharing of the resources of a computer and provides programmers with an interface that is used to access those resources.
- An Operating System processes **system data** and **user input**, and responds by allocating and managing tasks and internal system resources as a service to users and programs of the system.
- At the foundation of all system software, an Operating System performs basic tasks such as controlling and allocating memory, prioritizing system requests, controlling input and output devices, facilitating networking and managing file systems.

Overview

- The most commonly used contemporary desktop and laptop (notebook) OS is **Microsoft Windows**. More powerful servers often employ Linux, FreeBSD, and other Unix-like systems. However, these Unix-like operating systems, especially Mac OS X, are also used on personal computers.

•Following are some of the important functions of an OS:

- ❖ **Process Management** : It involves creation and deletion of user and system processes, deadlock handling, and so on.
- ❖ **Main-Memory Management** : It involves keeping track of the parts of memory that are being used, allocating/deallocating memory space as required, etc.
- ❖ **Secondary-Storage Management** : It involves free-space management, disk scheduling, storage allocation.
- ❖ **I/O System Management** : It deals with hardware specific drivers for devices, keeps it all hidden from the rest of the system.
- ❖ **File Management** : It involves creating/deleting files and directories, backup, and so on.

- ❖ **Protection System** : It involves controlling access to programs, processes, or users
- ❖ **Networking** : It generalizes the network access.
- ❖ **Command-Interpreter System**: It provides an interface between the user and the OS.

History

- UNIX evolved at AT&T Bell Labs in the late sixties.
- The writers of Unix are Ken Thomson, Rudd Canaday, Doug McIlroy, Joe Ossanna, and Dennis Ritchie.
- It was originally written as OS for PDP-7 and later for PDP-11.
- Liberal licensing: the source code was made available to universities, industries, and government organizations. This led to development of various versions of Unix as everybody added their own utilities. The important ones amongst them include BSD (Berkeley Software Distribution from University of California, Berkeley), SunOS (from Sun Microsystems). Various versions.
- System V in 1983 - Unification of all variants.

History

- It had an elegant file system, a command interpreter, and a set of utilities. Initially called UNICS (as a pun on MULTICS), by 1970, it came to be known as Unix.
- Unix was originally written in Assembly language. In 1973, Ritchie and Thompson rewrote the Unix kernel in “C”. This was a revolutionary step, as earlier the operating systems were written in assembly languages. The idea of writing it in “C” was so that the Operating system could now be ported (the speed, in effect, was traded off). It also made the system easier to maintain and adapt to particular requirements.

- UNIX OS exhibits the following features:
 - ❖ It is a simple User Interface.
 - ❖ It is Multi-User and Multiprocessing System.
 - ❖ It is a Time Sharing Operating System.
 - ❖ It is written in “C” (HLL).
 - ❖ It has a consistent file format - the Byte Stream.
 - ❖ It is a hierarchical file system.
 - ❖ It supports Languages such as FORTRAN, BASIC, PASCAL, Ada, COBOL, LISP, PROLOG, C, C++, and so on.

Services Provided by UNIX:

Process Management:

- ❖ It involves Creation, Termination, Suspension, and Communication between processes.

File Management:

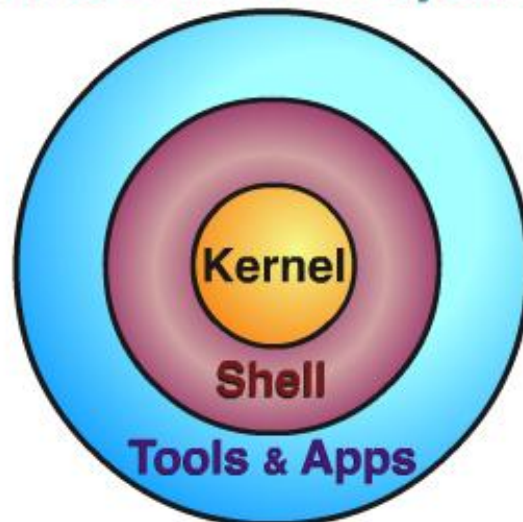
- ❖ It involves aspects related to files like creation and deletion, file security, and so on.

Block Diagram of Unix OS

The UNIX system is functionally organized at three levels:

- The **kernel**, which schedules tasks and manages storage
- The **shell**, which connects and interprets users' commands, calls programs from memory, and executes them
- The **tools** and **applications** that offer additional functionality to the operating system

Parts of the UNIX System



- Kernel:
 - The heart of the operating system, the kernel controls the hardware and turns part of the system on and off at the programmer's command. Suppose you ask the computer to list all the files in a directory. Then the kernel tells the computer to read all the files in that directory from the disk and display them on your screen.
- Shell
 - The **shell** is technically a **UNIX** command that interprets the user's requests to execute commands and programs. The **shell** acts as the main interface for the system, communicating with the **kernel**. The shell is also programmable in UNIX. There are many different types of shells like **Bourne Shell**, **Korn Shell**, and **C Shell**, most notably the command driven **Bourne Shell** and the **C Shell**, and menu-driven shells that make it easier for

- beginners to use. Whatever shell is used, its purpose remains the same -- to act as an interpreter between the user and the computer.
- Tools and Applications
 - There are hundreds of tools available to UNIX users, although some have been written by third party vendors for specific applications. Typically, tools are grouped into categories for certain functions, such as word processing, business applications, or programming.

- **man:**
 - It is possible to get a brief description about a command including all its options as well as some suitable examples.
- **cal :**
 - Any calendar from the year 1 to 9999 (either for a month or complete year), can be displayed using this command.
- **date:**
 - Unix has an internal clock, which actually stores the number of seconds elapsed from 1 Jan. 1970; and it is used for time stamping. A number of options are separately available to retrieve various components of the date.
 - ❖ **date “+%T”**
 - ❖ **date "+ %d %h“**

- **lp:**

- Jobs can be queued up for printing by using the spooling facility of Unix. Several users can print their files without any conflict. This command can be used to print the file, it returns the job number that can later be used to check the status of job.

❖ **lp myfile.txt**

- **tty:**

- Unix treats terminal also as a file. In order to display the device name of a terminal, the command used is **tty** (**teletype**).
- In order to display the current terminal related settings, the **stty** command can be used.
- The output of the **stty** command depends on the Unix implementation. It can also be used to change some settings.

- **Who**

- Unix maintains an account of all current users of system, the list of which can be printed using this command. The command can also be used to get one's own login details.

- ❖ **Who**

- ❖ **Who am I**

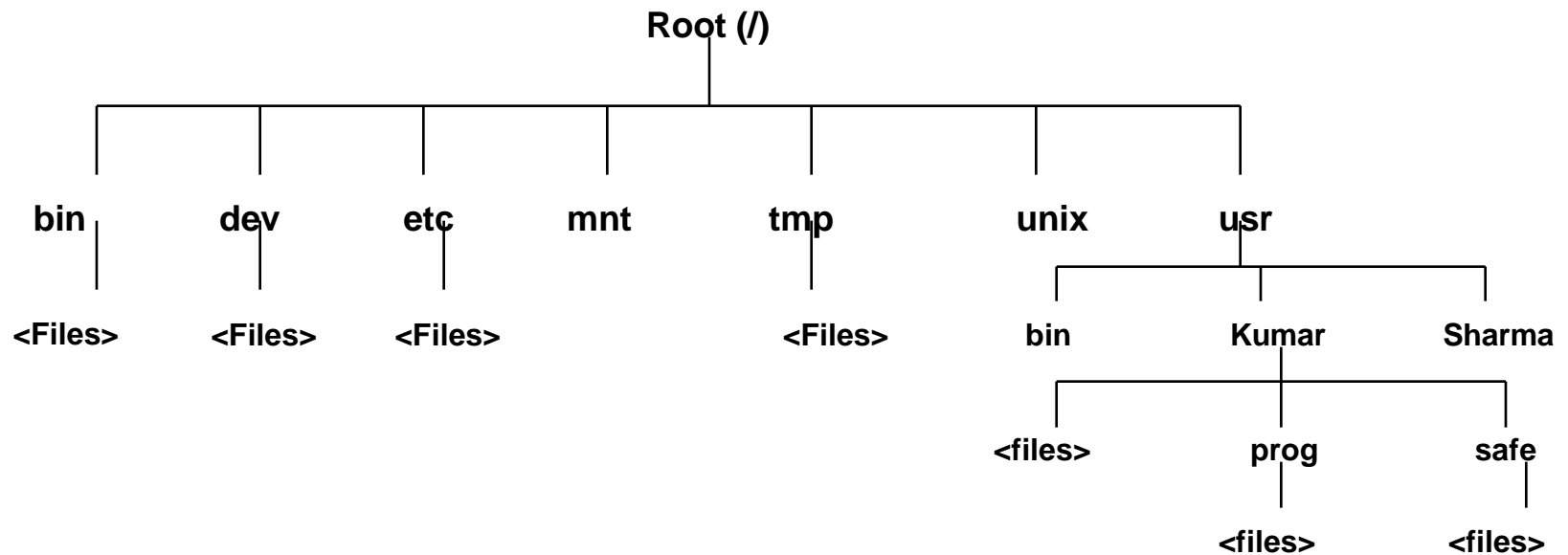
- **Logname:**

- command can be used to display the name of login directory, irrespective of what is the current working directory.

UNIX – Basic Commands

Unix File System

- UNIX works with a large number of files, which can belong to several different users.
- It becomes imperative for UNIX to organize files in a systematic fashion. The simple file system of UNIX is designed as an elaborate **Storage System** with separate compartment becoming available to store files. The system is widely adopted by different systems including DOS.
- The file system in UNIX is hierarchical. The **root**, a directory file represented by /, is at the top of the hierarchy and has several subdirectories (branches) under it.



- **/ bin** : commonly used UNIX Commands like who, ls
- **/usr/bin** : cat, wc etc. are stored here
- **/dev** : contains device files of all hardware devices
- **/etc** : contains those utilities mostly used by system administrator e.g. passwd, chmod, chown
- **/tmp** : used by some UNIX utilities especially vi and by user to store temporary files
- **/usr**: contains all the files created by user, including login directory
- **/unix** : kernel
- **Release V**:
 - It does not contain / bin.
 - It contains / home instead of /usr.

- **Pwd:**
 - The **pwd** command prints the present working directory
- **Cd:**
 - The **cd** command changes directories to specified directory .The directory name can be specified by using absolute path (Full Path) or relative path.
 - ❖ E.g. **cd** : Switches to home directory
 - ❖ **cd /** : Swicthes to root directory.

- **Unix Commands are categorized into :**
 - **External commands**
 - ❖ A new process will be set up
 - ❖ The file for external command should be available in BIN directory
 - E.g - cat, ls , Shell scripts
 - **Internal commands**
 - ❖ shell's own built in statements, and commands
 - ❖ No process is set up for such commands.
 - E.g cd , echo

- **Ls:**

- is used to list all the contents of the specified directory (in case no argument is specified, current directory is assumed).
- The ls command comes with several options, which are listed in the table. Many of these options can be combined to get relevant output. The command can also be used with more than one file name that is specified.
 - ❖ **X** :Displays multi columnar output (prior to Release 4)
 - ❖ **-F**:Marks executables with *and directories with /
 - ❖ **-r**:Sorts files in reverse order (ASCII collating sequence by default)
 - ❖ **-l**:The long listing showing seven attributes of a file
 - ❖ **-d**:Forces listing of a directory
 - ❖ **-a**:Shows all files including ., .. And those beginning with a dot

- ❖ **-t**: Sorts files by modification time
- ❖ **-R**: Recursive listing of all files in sub-directories
- ❖ **-u**: Sorts files by access time (when used with the **-t** option)
- ❖ **-i**: Shows i-node number of a file
- ❖ **-s**: Displays number of blocks used by a file

- **Cat :**

- The **cat** command can be used to display one or more files (if there is more than one file, then contents of the remaining immediately follow without any header information).
- To create a file using >
- To append to a file using >>

- **Cp:**

- The **cp** (copy file) command copies a file or group of files.

- **Rm:**

- This command is used to delete files.
 - ❖ **interactive (-i) delete and**
 - ❖ **recursive (-r) delete: The -r option will delete files from subfolders also.**

- **mv:**
 - The **mv** command is used to rename file or group of files as well as directories(belonging to the same parent).The destination file, if existing, gets overwritten.
- **wc:**
 - This command can be used to count the number of **lines (-l), words (-w), or characters (-c)** for one or more files.
 - If we specify multiple files, then the list of files should be separated by space. If no file name is specified, then it will accept data from standard i/p, that is from the keyboard.
- **nl:**
 - It is used to print file contents along with line numbers.
 - ❖ **-w** : width of the number
 - ❖ **-v** : Indicate first line number
 - ❖ **-i** : increment line number by

- **mkdir**




- A single directory, or a number of subdirectories, can be created using the mkdir command. A directory will not get created if there is already another directory by the same name under the parent directory.
- Besides, appropriate permissions will be required.

- **rmdir:**

- It is used to remove directory / directories.
- Only empty dir can be deleted.
- Command should be executed from at least one level above in the hierarchy.

- A UNIX file consists of a sequence of characters, and there are no restrictions on the file structure. The file consists only the actual bytes – and no extra information like its own size, attributes, or even an end of file marker. Extra information is stored in a structure called as **inode (index node)**. In UNIX, for every file, an inode is stored irrespective of its type.
- We have the following file types in UNIX:
 - Regular File/ Ordinary File:
 - ❖ These are also called as regular files. This is the “traditional” file, which can contain programs, data, object, and executable code, as well as all the UNIX programs and other user created files.
 - ❖ All text files belong to this category.
 - Directory File
 - Device File

- Every file in UNIX has a set of permissions using which it is determined “who can do what” with the file. This is stored as part of **inode** information, and is useful for maintaining file security.
- There are three categories of users: **Owner** (the user), **Group** (user is a member of which group), and **Others** (everybody else on the system). Access permissions of read (examining contents), write (changing contents) and execute (running program) are assigned to a file. These permissions are assigned to file owner, group and others.
- It is the file permissions based on which reading, writing or executing a file is decided for a particular user. It helps in giving restricted access to a file.

r w x	r w x	r w x
		
user(u)	group(g)	others (o)

- Permissions are interpreted as follows:
 - For a directory:
 - ❖ The “r” permission implies that one can find out the contents of the directory using commands like ls.
 - ❖ The “w” permission implies that it is possible to create and delete files in this directory. It is possible to remove even the files that are write-protected.
 - ❖ The “x” permission means search rather than execute. That is, it implies that the directory can be searched for a file. Also, when ‘x’ permission is set for a directory, one can do change dir to that directory and not otherwise.
 - Example: Using “r- -” will ensure that users can see the directory contents using ls, but cannot really use the contents.

- **Chmod:**

- In UNIX operating system every file is owned by the user who created that file. Only owner or Superuser can change the file permissions using chmod command.
- Permissions can be specified in two ways:
 - ❖ **by using symbolic notation, or**
 - ❖ **by using octal numbers :describes both category and permission**
 - » **Read(4),write (2) and execute (1)**
- **+ symbol, the previous permissions will be retained and new permissions will be added.**
- **= symbol, previous permissions will be overwritten.**
 - ❖ **chmod u-x, go+r note**
 - ❖ **chmod 741 note**

- Commands work with **character streams**.
- The default is the keyboard for input (standard input, file number 0), and terminal for the output (standard output, file number 1). In case of any errors, the system messages get written to standard error (file number 2), which defaults to a terminal.
- UNIX treats each of these streams as files, and these files are available to every command executed by the shell.
- It is the shell's responsibility to assign sources and destinations for a command. The shell can also replace any of the standard files by a physical file, which it does with the help of **metacharacters** for redirection.

- Redirection operators:
 - < : Input Redirection
 - > : Output Redirection
 - 2> : Error Redirection
 - >> : Append Redirection
- **Input redirection:** Instead of accepting i/p from standard i/p(keyboard) we can change it to file.
 - ❖ **Example:** \$cat < myfile will work same as \$cat myfile
 - < indicates, take i/p from myfile and display o/p on standard o/p device.

- **Output redirection:** To redirect o/p to some file use >
 - ❖ **Example:** `$cat < myfile > newfile`
 - The above command will take i/p from myfile and redirect o/p to new file instead of standard o/p (monitor).
 - If outfile does not exist, it is first created; otherwise, the contents are overwritten.
 - ❖ `$ cat < file1.txt > result` is same as `$cat file1.txt > result`
 - ❖ `cat < file1.lst >> result`

- **Error Redirection:** To redirect the standard error, 2> is used.
 - ❖ `cat abc.txt > pqr.txt 2> errfile.txt`
 - ❖ If file abc.txt exists: Then contents of the file will be sent to pqr.txt. Since no error has occurred nothing will be transferred to errfile.txt.
 - ❖ If abc.txt file does not exist: Then the error message will be transferred to errfile.txt and pqr.txt will remain empty.

- **Cmp :**

- The cmp command can be used to compare if two files are matching or not.
- The comparison is done on a byte by byte basis.
- In case files are identical, no message is displayed. Otherwise, locations of mismatches are echoed.

- **Comm:**

- It compares two files and gives a 3 columnar output:
 - ❖ First column contains lines unique to the first file.
 - ❖ Second column contains lines unique to the second file.
 - ❖ Third column displays the common lines.
- Selective column output can be obtained by using options -1, -2 or -3. It would drop the column(s) specified from the output.

- **Diff:**
 - The **diff** command is used to display the file differences. It tells the lines of one file that need to be changed to make the two files identical.
- **Tr:**
 - The **tr** command accepts i/p from standard input and it takes two arguments which specify two character sets.
 - ❖ The first character set is replaced by the equivalent member in the second character set.
 - The **-s** option is used to squeeze several occurrences of a character to one character.
 - ❖ `tr "[a-z]" "[A-Z]" < file1.txt`
 - ❖ `tr -s " " < file1.txt`

- **More:**
 - It is a paging tool – it can be used to view one page at a time.
 - It is particularly useful for viewing large files.

- Metacharacters are characters to which the shell attaches special meaning. The shell interprets these metacharacters and the command is rebuilt before it is passed on to the kernel.
- Following are the Bourne Shell metacharacters:
 - ❖ *** : To match any number of characters**
 - **emp* : This would match all patterns that begin with emp, and may be followed by any number of characters (like emp, emppune, empttc, empseepz etc).**
 - ❖ **? : To match with a single character**
 - **emp? : This would match all patterns that begin with emp and are followed by exactly one more character, which could be anything (like emp1, empa etc).**

- ❖ **[]** : Character class; Matching with any single character specified within []
 - **emp[abc]**: This would match with patterns that begin with emp followed by a or b or c any one of it. (like emp`a`, emp`b` or emp`c`)
- ❖ **!** : To reverse matching criteria of character class
- ❖ **** : To remove special meaning attached to metacharacters
 - **emp*** : This would match only with the pattern emp`*`.
- ❖ **;** : To give more than one command at the same prompt
 - **ls -l file2.txt ; chmod u+x file2.txt ; ls -l file2.txt**
- ❖ All redirection operators **>**, **<**, **>>** are also shell metacharacters

- The shell has a special operator called pipe (`|`), using which the output of one command can be sent as an input to another.
- The shell sets up the interconnection between commands. It eliminates the need of temporary files for storing intermediate results.
- Pipe - allows stream of data to be passed between reader & writer process.
 - ❖ `$ who | wc -l`
 - ❖ `$ ls | wc -l`
- Any number of commands can be combined together to make a single command.

Command Substitution?

- Shell allows the argument of a command to be obtained from the output of another command. This is done by using a pair of backquotes.
- The backquoted command is executed first. The output of command is substituted in place of command. Then outer command will get executed.
 - `$ cal `date "+%m 20%y"``

UNIX – Basic Commands

vi Editor

- Three Modes of Vi Editor are: Input, Command and ex mode.

Input Mode : The Input mode is used to insert, append, replace or change text. A summary of input mode commands are given below:

- i : inserts text to left of cursor
- I Inserts text at beginning of line
- a Appends text to right of cursor
- A Appends text at end of line
- o Opens line below
- O Opens line above

- From input mode to command mode press <Esc>
- From command mode:
 - To Save : w
 - To Quit : q
 - To Quit without saving : q!
 - To save & quit : wq or : x
- The most commonly used operators are:
 - d – delete

Using set command

- Set command is used to customize the behavior of the VI editor
- Some of the useful commands

Sr no.	Command	Description
1.	:set autoindent or :set ai	To set autoindent on
2	:set numbers or :set nu	To Displays lines with line numbers on the left side
3	:set smd or :set showmode	To show the actual mode of the editor that you are in at the bottom line.
4.	:set wm=x or :set wrapmargin=x	To automatically wrap the word on next line, x will be any nonzero value. (:set wm=2 sets the wrap margin to 2 characters)

- To get the list of all options in set command use **:set all**

UNIX – Basic Commands

Unix Filters

What is a Filter?

- Filters are central tools of the UNIX tool kit.
- Commands work as follows:
 - Accept some data as input.
 - Perform some manipulation on the inputted data.
 - Produce some output.
- They normally work on set of records, with each field of a record delimited by a suitable delimiter.

- **Head:**

- These are simple horizontal filters.
- Using head, it is possible to display beginning of one or more lines from files.
- By default, the first 10 lines are displayed. Incase a numeric line count argument is specified, the command would display those many lines from the beginning of the file.

- **Tail:**

- The **tail** command is useful to display last few lines or characters of the file.

- **Cut:**

- The **cut** command can be used to retrieve specific column information from a file.
- In case of fixed record formats, the
 - ❖ **-c** (columns) option can be used to specify column positions.
 - ❖ **-f** (field)
 - ❖ **-d** (delimiter) options

- **Paste:**

- It is used for horizontal merging of files.

- **Sort:**

- The sort command sorts a file (which may or may not contain fixed length records) on line by line basis in the ascending ASCII order.
- Sorting can be done on one or more fields or on character positions within fields.
- Since the sorting is done on the basis of ASCII collating sequence, incase of sorting of numbers, -n option needs to be used.
- Options are:
 - ❖ -r : Reverse order
 - ❖ -n : Numeric sort
 - ❖ -f : Omit the difference between Upper and lower case alphabets
 - ❖ -t : Specify delimiter
 - ❖ -k : to specify fields as primary or secondary key

- **Uniq:**

- The **uniq** command fetches only one copy of redundant records from a sorted file and writes the same to standard output.
 - ❖ **-u option:** It selects only non-repeated lines.
 - ❖ **-d option:** It selects only one copy of repeated line.
 - ❖ **-c option:** It gives a count of occurrences.
- To find unique values, the file has to be sorted on that field.

- **Tee:**

- The **tee** command copies the standard input to the standard output and also to the specified file.
- If it is required to see the output on screen as well as to save output to a file, the **tee** command can be used.

- **Find:**

- The **find** command is used to find files matching a certain set of selection criteria. The **find** command searches recursively in a hierarchy, and also for each pathname in the pathname-list (a list of one or more pathnames specified for searching).

- **Grep:**

- The **grep** command is used to locate a pattern / expression in a file / set of files.

- ❖ E.g: `grep 'Unix' books.lst`

- The **grep** command scans the file(s) specified for the required pattern, and outputs the lines containing the pattern. Depending on the options used, appropriate output is printed.

- The grep command compulsorily requires a pattern to be specified, and the rest of the arguments are considered as file names in which the pattern has to be searched.
- Options of grep:
 - ❖ **c** : It displays count of lines which match the pattern.
 - ❖ **n** : It displays lines with the number of the line in the text file which match the pattern.
 - ❖ **v** : It displays all lines which do not match pattern.
 - ❖ **i** : It ignores case while matching pattern.
 - ❖ **w** : It forces grep to select only those lines containing matches that form whole words

- Regular Expression:

Expression	Description
<code>^</code> (Caret)	match expression at the start of a line, as in <code>^A</code> .
<code>\$</code> (Question)	match expression at the end of a line, as in <code>A\$</code> .
<code>\</code> (Back Slash)	turn off the special meaning of the next character, as in <code>\^</code> .
<code>[]</code> (Brackets)	match any one of the enclosed characters, as in <code>[aeiou]</code> . Use Hyphen <code>-</code> for a range, as in <code>[0-9]</code> .
<code>[^]</code>	match any one character except those enclosed in <code>[]</code> , as in <code>[^0-9]</code> .
<code>.</code> (Period)	match a single character of any value, except end of line.
<code>*</code> (Asterisk)	match zero or more of the preceding character or expression.
<code>\{x,y\}</code>	match x to y occurrences of the preceding.
<code>\{x\}</code>	match exactly x occurrences of the preceding.
<code>\{x,\}</code>	match x or more occurrences of the preceding.

- Examples of Regular Expression:

Example	Description
grep "smile" files	search <i>files</i> for lines with 'smile'
grep '^smile' files	'smile' at the start of a line
grep 'smile\$' files	'smile' at the end of a line
grep '^smile\$' files	lines containing only 'smile'
grep '\^s' files	lines starting with '^s', "\" escapes the ^
grep '[Ss]mile' files	search for 'Smile' or 'smile'
grep 'B[oO][bB]' files	search for BOB, Bob, BOb or BoB
grep '^\$' files	search for blank lines
grep '[0-9][0-9]' file	search for pairs of numeric digits

- **fgrep**
 - This command is similar to **grep** command. The **fgrep** command can also accept multiple patterns from command line as well as a file. However, it does not accept regular expressions - only fixed strings can be specified
 - The **fgrep** command is faster than **grep** and **egrep**, and should be used while using fixed strings.
- The principal disadvantage of the **grep** family of filters is that there are no options available to identify fields. Also it is very difficult to search for an expression in a field.

UNIX – Basic Commands

Process Related Commands

- A process is an instance of a program in execution. When any executable file is executed, a process starts.
- It remains active while the program is executing. When the program terminates, the process dies. Generally the name of the process is the name of the executable.
- Since Unix is a multi-tasking system, there can be many processes that run at the same time.
- A unique number called as the Process Identifier, PID, identifies each of these processes. The kernel allocates this PID, which is a number from 0 to 32767.
- It is the responsibility of the kernel to manage the processes - in terms of time allocated to process, its associated priorities and swapping etc.

- **ps:**
 - The ps command can be used to display the characteristics of the processes. It has the knowledge of kernel built into it, using which it can read the process tables to get necessary information.
 - Options used are:
 - ❖ -f - full form
 - ❖ -u - details of only users processes
 - ❖ -a - all processes details
 - ❖ -l - detailed listing
 - ❖ -e - system processes

- Output of ps -l command: It displays Long format
 - **F** - Octal flags which are added together to give more information about the current status of a process(20 - process is loaded in primary memory: it has not been swapped out to disk)
 - **S** - State of the process (O - Process is running on a processor, R - Process is on run queue)
 - **UID** - The Userid of the process owner (login name is printed using -f option)
 - **PID** - The process ID of the process (this number is needed to kill a process)
 - **PPID** - The process ID of the parent process
 - **C** - CPU usage by the process; combination of this value with nice value is used to calculate the priority

- **PRI** - The priority of the process (lower number mean lower priority)
- **NI** - The nice value of the process
- **ADDR** - The virtual address of the process entry in the process table

- Processes can run in foreground or background mode.
 - Only one process can run in foreground mode but multiple processes can run in background mode.
 - The processes, which do not require user intervention can run in background mode, e.g. sort, find.
 - To run a process in background, use & operator
 - ❖ `$sort -O emp.lst emp.lst &`
 - Even if a command is run in the background, it is possible that the output as well as error messages are still coming to the standard output. Hence, care should be taken to suitably redirect this output.
 - However, too many jobs should not be run in the background as significant deterioration of CPU performance can occur.

- **nohup (no hangup) -**

- permits execution of process even if user has logged off.

- ❖ `$nohup sort emp.lst &` (sends output to nohup.out)

- **Kill**

- Used to terminate a process

- It is possible to send signals to processes. When a process receives a signal, it can ignore it, terminate or do something else.

- Signals in Unix are identified by a number - each signal notifying that an event has occurred.

- ❖ `$kill 1005` (default signal 15) - kills job with pid 1005

- ❖ `$kill -9 1005` - sure killing of job

- ❖ `$kill 0` - kills all background process or `kill -STOP`

- **Nice:**

- runs a program with modified scheduling priority.
- All processes on Unix are usually executed with equal priority. In order to reduce the priority of a job, the command needs to be prefixed with nice.
- By default, nice reduces the priority of any process by 10 units. The amount of reduction can also be specified as an argument (value from 0 to 19) to the nice command.
- In case of commands in a pipeline, to reduce the priority of all commands in the pipeline, nice needs to be used in each element in the pipeline.
 - ❖ **\$ nice cat chap?? | nice wc -l > wclist &**

- **Wait:**
 - waits for child process to complete.
 - The wait command can be used to wait for the completion of a background process.
 - This is a built-in shell command - no process is spawned for this command. It sends the shell into a wait state so that it can acknowledge the death of child processes.
 - ❖ `$wait 138` - waits for background job with pid 138
- **bg:**
 - Pushes a job to be executed in background
- **jobs:**
 - Lists the status of the jobs already running

- **Fg:**
 - Brings any of the background jobs to foreground
 - ❖ **Fg** : brings the most recent background job to foreground
 - ❖ **Fg %+1** : brings the first background job to foreground
 - ❖ **Fg %sort**: brings the sort job to foreground.
- **mail:**
 - Sending mail to a user
 - ❖ **mail trainer** : sends a mail to the trainer
 - ❖ **mail** : opens the mail box of the user
 - ❖ **&r** : reply to mails

- At:
 - Schedules a job to be executed at a given time
 - ❖ At 14:03 sh script1
 - If suffix am / pm is omitted it takes it to be in 24-hr format.
 - ❖ At noon : at 12:00 hrs
 - ❖ At now + 1 year : current time after 1 year
 - ❖ At now + 1 day : current time the next day
 - ❖ At 1:00 pm Decemeber 25,2010
 - ❖ At 1:00 pm Mon
 - - l: lists the job scheduled.
 - -r : removes the first job from the queue

- Batch:
 - Schedules job to be executed as soon as the system load permits.
 - It doesn't take any arguments and it uses an internal algorithm to decide when the job is going to be executed.
- cron:
 - Executes programs at different intervals
 - It searches for the control file in /usr/spool/cron/crontabs for the commands/instructions to be executed at that instance.

UNIX – Basic Commands

Shell Programming

- There are several variables set by the system - some during booting and some after logging in. These are called the system variables, and they determine the environment one is working in. The user can also alter their values.
- The set statement can be used to display list of system variables.
 - Set
- Shell Variables
 - PATH : Contains the search path string.
 - ❖ Determines the list of directories (in order of precedence) that need to be scanned while you look for an executable command.
 - ❖ Path can be modified as: `$ PATH=$PATH:/usr/user1/progs`
 - ❖ This causes the `/usr/user1/progs` path to get added to the existing PATH list.

- HOME : Specifies full path names for user login directory.
- TERM : Holds terminal specification information
- LOGNAME : Holds the user login name.
- PS1 : Stores the primary prompt string.
- PS2 : Specifies the secondary prompt string.

- Script:

echo "Good Morning!"

echo "Enter your name?"

read name

echo "HELLO \$name How are you?"

- To Execute it

➤ **sh script_name**

- To Debug it

➤ **sh -x script_name**

- Script:
 echo "Enter first Number"
 read no1
 echo "Enter second Number"
 read no2
 res=`expr \$no1 + \$no2`
 echo "The result is \$res"
- In above script **res=`expr \$no1 + \$no2`** can be replaced by **let res=no1+no2**
- Add *one* to variable *i*. Using let statement:
 - ❖ **let i=i+1** (If no spaces in expression)
 - ❖ **let "i = i + 1"** (enclose expression in "... " if expression includes spaces)
 - ❖ **((i = i + 1))**

- Example: Examine the difference between the two codes.

```
var=pwd  
echo $var
```

- Let's see this:

```
var=`pwd`  
echo $var
```

- Since `pwd` is enclosed in backquotes it will get replaced by present working directory. `echo` will display name of current working directory.

- You can pass values to shell programs while you execute shell scripts. These values entered through command line are called as *command line arguments*.
- Arguments are assigned to special variables \$1, \$2 etc called as positional parameters.
- *special parameters*
 - \$0 – Gives the name of the executed command
 - \$* - Gives the complete set of positional parameters
 - \$# - Gives the number of arguments
 - \$\$ - Gives the PID of the current shell
 - \$! – Gives the PID of the last background job
 - \$? – Gives the exit status of the last command
 - @\$ - Similar to \$*, but generally used with strings in looping constructs

- Example:

echo Program: \$0

echo Number of arguments are \$#

echo arguments are \$*

- **Conditional Execution using && and ||**

- The shell provides && and || operators to control the execution of a command depending on the success or failure of previous command.

- && ensures the second command executes only if the first has succeeded.
- || will ensure that the second command is executed only if the first has failed.

❖ **\$grep `director` emp.lst && echo “pattern found”**

❖ **\$grep `manager` emp.lst || echo “pattern not found”**

Shell Script

- Syntax:

```
if <condition is true>  
then  
    <execute commands>  
else  
    <execute commands>  
fi
```

- Example:

```
if grep "director" emp.lst  
then  
    echo "Pattern Found"  
else  
    echo "Pattern Not Found"  
fi
```

- Specify condition either using *test* or *[condition]*
 - Example: `test $1 -eq $2` same as `[$1 -eq $2]`
- Relational Operator for Numbers:
 - `eq`: Equal to
 - `ne`: Not equal to
 - `gt`: Greater than
 - `ge`: Greater than or equal to
 - `lt`: Less than
 - `le`: Less than or equal to

- **Example:**

```
if test $# -lt 2
then
    echo "Please key in the arguments"
else
    if [ $1 -gt 0 ]
    then
        echo "Number is positive"
    elif test $1 -lt 0; then
        echo "Number is negative"
    else
        echo "Zero"
    fi
fi
```


Relational Operator for strings and logical operators

- String operators used by *test*:
 - -n str True, if str not a null string
 - -z str True, if str is a null string
 - S1 = S2 True, if S1 = S2
 - S1 != S2 True, if S1 ≠ S2
 - str True, if str is assigned and not null

- Logical Operators
 - -a .AND.
 - -o .OR.
 - 81 ➤ ! Not

File related operators

- File related operators used by test command
 - -f <file> True, if file exists and it is regular file
 - -d<file> True, if file exist and it is directory file
 - -r <file> True, if file exist and it is readable file
 - -w <file> True, if file exist and it is writable file
 - -x <file> True, if file exist and it is executable file
 - -s <file> True, if file exist and it's size > 0
 - -e <file> True, if file exist

➤ Example:

```
echo "Enter File Name:\c "  
read source  
if [ -z "$source" ] ;then  
    echo "You have not entered file name"  
else  
    if test -s "$source";then    #file exists & size is > 0  
        if test ! -r "$source" ;then  
            echo "Source file is not readable"  
            exit  
        fi ;else  
            echo "Source file not present"  
            exit  
        fi  
    fi  
fi
```

- Select Case

```
case <expression> in
    <pattern 1> ) <execute commands> ;;
    <pattern 2> ) <execute commands> ;;
    <...>
esac
```

- Example:

```
echo "\n Enter Option : \c"
read choice
case $choice in
    1) ls -l ;;
    2) ps -f ;;
    3) date ;;
    4) who ;;
esac
```

- Example:

```
case `date | cut -d" " -f1` in
    Mon ) <commands> ;;
    Tue ) <commands> ;;
    :
esac
```

- Example:

```
echo "do you wish to continue?"
read ans
case "$ans" in
    [yY] [eE] [sS]) ;;
    [nN] [oO]) exit ;;
    *) "invalid option" ;;
esac
```

- While:

```
while <condition is true>  
do  
    <execute statements>  
done
```

- Example:

```
num=1  
while [ $num -le 10 ]  
do  
    echo $num  
    num=`expr $num + 1`  
done  
#end of script
```

- Continue:
 - Suspends statement execution following it.
 - Switches control to the top of loop for the next iteration.
- Break:
 - Causes control to break out of the loop.

- For Loop

```
for variable in list
do
    <execute commands>
done
```

- Example:

```
for x in 1 2 3
do
    echo "The value of x is $x"
done
```


- Example:

```
for (( i = 0 ; i <= 5; i++ ))  
do  
    echo "Welcome $i times"  
done
```

- Example:

```
for x in `cat first`  
do  
    echo $x  
done
```

- Example:

```
for x in *  
do  
    echo $x  
done
```

- Example:

```
for x in $*  
do  
    grep "$x" emp.lst || echo "Pattern Not Found"  
done
```

- Example:

```
for x in $@  
do  
    grep "$x" emp.lst || echo "Pattern Not Found"  
done
```

- Use shell functions to modularize the script.
 - These are also called as script module
 - Normally defined at the beginning of the script.
 - Syntax (Function Definition):

```
functionname(){  
    commands  
}
```

- Example:

```
Myfunction(){  
    echo "$*"  
    echo "The number should be between 1 and 20"  
    read num  
    if [ $num -le 1 ] -a [ $num -ge 20 ]; then  
        return 1;  
    else  
        return 0;  
    fi  
    echo "You will never reach to this line"  
}  
echo "Calling the function Myfunction"  
if Myfunction "Enter the number"  
then  
    echo "The number is within range"  
else  
    echo the number is out of range"
```

```
flowers[0]=Rose
flowers[1]=Lotus
flowers[2]=Mogra
i=0
while [ $i -lt 3 ]
do
    echo ${flowers[$i]}
    i=`expr $i+1`
done
```

To access all elements:

```
${array_name[*]}
${array_name[@]}
```

UNIX – Basic Commands

AWK

- Named after Aho, Weinberger, Kernigham.
- As powerful as any programming language.
- It can access, transform and format individual fields in a record - it is also called as a report writer. It can accept regular expressions for pattern matching, has “C” type programming constructs, variables and in-built functions. Based on *pattern matching* and *performing action*.
- Limitations of the *grep* family are:
 - ❖ No options to identify and work with fields.
 - ❖ Output formatting, computations etc. is not possible.
 - ❖ Extremely difficult to specify patterns or regular expression always.
- AWK overcomes all these drawbacks.

- Syntax:
 - `awk <options> 'line specifier {action}' <files>`
- Example:
 - `awk '{ print $0 }' emp.lst`
 - This program prints the entire line from the file *emp.data*.
 - `$0` refers to the entire line from the file *emp.data*.

- Variable List:
 - **\$0**: Contains contents of the full current record.
 - **\$1..\$n**: Holds contents of individual fields in the current record.
 - **NF**: Contains number of fields in the input record.
 - **NR**: Contains number of record processed so far.
 - **FS**: Holds the field separator. Default is *space* or *tab*.
 - **OFS**: Holds the output field separator.
 - **RS**: Holds the input record separator. Default is a new line.
 - **FILENAME**: Holds the input file name.
 - **ARGC**: Holds the number of Arguments on Command line
 - **ARGV**: The list of arguments

- Example
 - `awk -F "|" '{ print $1 $2 $3 }' emp.lst`
 - ❖ This prints the *first*, *second* and *third* column from file
 - `awk '{ print }' emp.data`
 - Prints all lines (all the fields) from file *emp.data*.
- Fields are identified by special variable \$1, \$2,;
- Default delimiter is a contiguous string of spaces.
- Explicit delimiter can be specified using -F option
 - Example: `awk -F "|" '/sales/{print $3, $4}' emp.lst`
 - `awk -F"|" '$1=="1002" {printf "%2d,%-20s",NR,$2}' emp.lst`
 - `awk -F "|" '$5 > 7000 { print $1, $2 * $3 }' emp.lst`

- Line numbers can be selected using NR built-in variable.
 - `awk -F "|" 'NR ==3, NR ==6 {print NR, $0}' emp.lst`
 - `awk '{ print NF, $1, NR }' emp.data`
 - `awk '$3 == 0' emp.data`
 - `awk '{ print NR, $0 }' emp.data`
 - `awk ' $1 == "Susie" ' emp.data`
- Relational and Logical Operators can also be used.
 - `$awk - F "|" '$3 == "director" || $3 == "chairman">{printf "%-20s", $2}' emp.lst`
 - `$awk -F "|" '$6>7500 {printf "%20s", $2}' emp.lst`
 - `$awk -F "|" '$3 == "director" || $6>7500 {print $0}' emp.lst`

- Computations can also be performed in AWK

```
$awk -F "|" '$3 == "director" || $6>7500 {
```

```
    kount = kount+1
```

```
    printf "%3d%-20s\n", kount,$2}' emp.lst
```

- Programming can be done in separate file.

➤ Example: awk1

```
$3 == "Director" {print $2,$4,$5}
```

➤ To execute the AWK command using the file

```
awk -F "|" -f awk1 emp.lst
```

- BEGIN and END Section:
- In case, something is to be printed before processing the first line begins, BEGIN section can be used.
 - Normally if you want to print any header lines or want to set field separator then use BEGIN section
- Similarly, to print at the end of processing, END section can be used.
 - And If you want to display total or any summarized information at the end of the report then use END section
- These sections are optional. When present, they are delimited by the body of the awk program.
 - Format: (i) BEGIN {action} (ii) END {action}

- Example:

```
BEGIN {  
    printf "\n\t Employee details \n\n"  
}  
$6>7500{  
    # increment sr. no. and sum salary  
    kount++; tot+=$6  
    printf "%d%-20s%d\n", kount, $2, $6  
}  
END {  
    printf "\n The Avg. Sal. Is %6d\n", tot/kount  
}
```

```
$awk -F "|" -f emp.awk emp.lst
```

- It is possible to store an entire awk command into a file as a shell script, and pass parameters as arguments to the shell script. The shell will identify these arguments as \$1, \$2 etc based on the order of their occurrence on the command line.
- Within awk, since \$1, \$2 etc. indicate fields of data file, it is necessary to distinguish between the positional parameters and field identifiers. This is done by using single quotes for the positional parameters used by awk.
 - Positional parameter should be single-quoted in an *awk* program.
 - Example: \$3 > '\$1'