SYSTEM PROGRAMMING AND SYSTEM ADMINISTRATION

SEM-6TH

SECTION-C (NOTES)

UNIX ACCOUNTS:-

There are three types of accounts on a Unix system −

Root account

This is also called **superuser** and would have complete and unfettered control of the system. A superuser can run any commands without any restriction. This user should be assumed as a system administrator.

System accounts

System accounts are those needed for the operation of system-specific components for example mail accounts and the **sshd** accounts.

User accounts

User accounts provide interactive access to the system for users and groups of users. General users are typically assigned to these accounts and usually have limited access to critical system files and directories.

Managing Users and Groups

There are four main user administration files −

- **/etc/passwd** − Keeps the user account and password information. This file holds the majority of information about accounts on the Unix system.
- **/etc/shadow** − Holds the encrypted password of the corresponding account. Not all the systems support this file.
- **/etc/group** − This file contains the group information for each account.
- **/etc/gshadow** − This file contains secure group account information.

Create a Group

groupadd [-g gid [-o]] [-r] [-f] groupname

Modify a Group

To modify a group, use the **groupmod** syntax −

$ groupmod -n new_modified_group_name old_group_name

Delete a Group

We will now understand how to delete a group. To delete an existing group, all you need is the **groupdel command** and the **group name**. To delete the financial group, the command is −

$ groupdel developer

Create an Account

Let us see how to create a new account on your Unix system. Following is the syntax to create a user's account −

useradd -d homedir -g groupname -m -s shell -u userid accountname

S.No.	Option & Description
	-d homedir
1	Specifies home directory for the account
	-g groupname
2	Specifies a group account for this account
	-m
3	Creates the home directory if it doesn't exist
	-s shell
4	Specifies the default shell for this account
	-u userid
5	You can specify a user id for this account
	accountname
6	Actual account name to be created

Once an account is created you can set its password using the **passwd** command as follows −

PASSWORD:

$ passwd mcmohd20
Changing password for user mcmohd20.
New UNIX password:
Retype new UNIX password:
passwd: all authentication tokens updated successfully.

When you type ***passwd accountname***, it gives you an option to change the password, provided you are a superuser. Otherwise, you can change just your password using the same command but without specifying your account name.

Modify an Account

The **usermod** command enables you to make changes to an existing account from the command line. It uses the same arguments as the **useradd** command, plus the -l argument, which allows you

to change the account name.

For example, to change the account name ***mcmohd*** to ***mcmohd20*** and to change home directory accordingly, you will need to issue the following command −

$ usermod -d /home/mcmohd20 -m -l mcmohd mcmohd20

Files

- **ls** --- lists your files
  ls -l --- lists your files in 'long format', which contains lots of useful information, e.g. the exact size of the file, who owns the file and who has the right to look at it, and when it was last modified.
  ls -a --- lists all files, including the ones whose filenames begin in a dot, which you do not always want to see.
  There are many more options, for example to list files by size, by date, recursively etc.
- **more *filename*** --- shows the first part of a file, just as much as will fit on one screen. Just hit the space bar to see more or **q** to quit. You can use **/*pattern*** to search for a pattern.
- **emacs *filename*** --- is an editor that lets you create and edit a file.
- **mv *filename1 filename2*** --- moves a file (i.e. gives it a different name, or moves it into a different directory (see below)
- **cp *filename1 filename2*** --- copies a file
- **rm *filename*** --- removes a file. It is wise to use the option rm -i, which will ask you for confirmation before actually deleting anything. You can make this your default by making an alias in your .cshrc file.
- **diff *filename1 filename2*** --- compares files, and shows where they differ
- **wc *filename*** --- tells you how many lines, words, and characters there are in a file
- **chmod *options filename*** --- lets you change the read, write, and execute permissions on your files. The default is that only you can look at them and change them, but you may sometimes want to change these permissions. For example, **chmod o+r *filename*** will make the file readable for everyone, and **chmod o-r *filename*** will make it unreadable for others again. Note that for someone to be able to actually look at the file the directories it is in need to be at least executable.

Directories

Directories, like folders on a Macintosh, are used to group files together in a hierarchical structure.

- **mkdir *dirname*** --- make a new directory
- **cd *dirname*** --- change directory. You basically 'go' to another directory, and you will see the files in that directory when you do 'ls'. You always start out in your 'home directory', and you can get back there by typing 'cd' without arguments. 'cd ..' will get you one level up from your current position. You don't have to walk along step by step - you can make big leaps or avoid walking around by specifying pathnames.
- **pwd** --- tells you where you currently are.
- **ff** --- find files anywhere on the system. This can be extremely useful if you've forgotten in which directory you put a file, but do remember the name. In fact, if you use **ff -p** you don't

even need the full name, just the beginning. This can also be useful for finding other things on the system, e.g. documentation.

- **grep *string filename(s)*** --- looks for the string in the files. This can be useful a lot of purposes, e.g. finding the right file among many, figuring out which is the right version of something, and even doing serious corpus work. grep comes in several varieties (**grep**, **egrep**, and **fgrep**) and has a lot of very flexible options.
- **w** --- tells you who's logged in, and what they're doing. Especially useful: the 'idle' part. This allows you to see whether they're actually sitting there typing away at their keyboards right at the moment.
- **who** --- tells you who's logged on, and where they're coming from. Useful if you're looking for someone who's actually physically in the same building as you, or in some other particular location.
- **finger *username*** --- gives you lots of information about that user, e.g. when they last read their mail and whether they're logged in. Often people put other practical information, such as phone numbers and addresses, in a file called **.plan**. This information is also displayed by 'finger'.
- **last -1 *username*** --- tells you when the user last logged on and off and from where. Without any options, **last** will give you a list of everyone's logins.
- **talk *username*** --- lets you have a (typed) conversation with another user
- **write *username*** --- lets you exchange one-line messages with another user
- **elm** --- lets you send e-mail messages to people around the world (and, of course, read them).
- **whoami** --- returns your username. Sounds useless, but isn't. You may need to find out who it is who forgot to log out somewhere, and make sure *you* have logged out.
- **passwd** --- lets you change your password, which you should do regularly.
- **ps -u *yourusername*** --- lists your processes. Contains lots of information about them, including the process ID, which you need if you have to kill a process.
- **kill *PID*** --- kills (ends) the processes with the ID you gave. This works only for your own processes, of course. Get the ID by using **ps**. If the process doesn't 'die' properly, use the option -9.
- **quota -v** --- show what your disk quota is (i.e. how much space you have to store files)
- **du *filename*** --- shows the disk usage of the files and directories in *filename* (without argument the current directory is used). **du -s** gives only a total.
- **last *yourusername*** --- lists your last logins. Can be a useful memory aid for when you were where, how long you've been working for, and keeping track of your phonebill if you're making a non-local phonecall for dialling in.

PATH:

A path is a unique location to a file or a folder in a file system of an OS. A path to a file is a combination of / and alpha-numeric characters.

What is an absolute path?

An absolute path is defined as the specifying the location of a file or directory from the root directory(/). In other words we can say absolute path is a complete path from start of actual filesystem from / directory.

Some examples of absolute path:

/var/**ftp**/pub
/etc/samba.smb.conf
/boot/grub/grub.conf

What is the relative path?

Relative path is defined as path related to the present working directory(pwd). Suppose I am located in /var/log and I want to change directory to /var/log/kernel. I can use relative path concept to change directory to kernel

changing directory to /var/log/kernel by using relative path concept.

pwd/var/logcd kernel

Every file in Unix has the following attributes −

- **Owner permissions** − The owner's permissions determine what actions the owner of the file can perform on the file.
- **Group permissions** − The group's permissions determine what actions a user, who is a member of the group that a file belongs to, can perform on the file.
- **Other (world) permissions** − The permissions for others indicate what action all other users can perform on the file.

The Permission Indicators

While using **ls -l** command, it displays various information related to file permission as follows −

```
$ls -l /home/amrood
-rwxr-xr--  1 amrood   users 1024  Nov 2 00:10  myfile
drwxr-xr--- 1 amrood   users 1024  Nov 2 00:10  mydir
```

The permissions are broken into groups of threes, and each position in the group denotes a specific permission, in this order: read (r), write (w), execute (x) −

- The first three characters (2-4) represent the permissions for the file's owner. For example, **-rwxr-xr--** represents that the owner has read (r), write (w) and execute (x) permission.
- The second group of three characters (5-7) consists of the permissions for the group to which the file belongs. For example, **-rwxr-xr--** represents that the group has read (r) and execute (x) permission, but no write permission.
- The last group of three characters (8-10) represents the permissions for everyone else. For example, **-rwxr-xr--** represents that there is **read (r)** only permission.

File Access Modes

The permissions of a file are the first line of defense in the security of a Unix system. The basic building blocks of Unix permissions are the **read**, **write**, and **execute** permissions, which have been described below −

Read

Grants the capability to read, i.e., view the contents of the file.

Write

Grants the capability to modify, or remove the content of the file.

Execute

User with execute permissions can run a file as a program.

Pipes and Filters

To make a pipe, put a vertical bar (|) on the command line between two commands.

When a program takes its input from another program, it performs some operation on that input, and writes the result to the standard output. It is referred to as a ***filter***.

The grep Command

The grep command searches a file or files for lines that have a certain pattern. The syntax is −

```
$grep pattern file(s)
```

The name **"grep"** comes from the ed (a Unix line editor) command **g/re/p** which means "globally search for a regular expression and print all lines containing it".

A regular expression is either some plain text (a word, for example) and/or special characters used for pattern matching.

The sort Command

The **sort** command arranges lines of text alphabetically or numerically. The following example sorts the lines in the food file −

```
$sort food
Afghani Cuisine
Bangkok Wok
Big Apple Deli
Isle of Java
```

The pg and more Commands

A long output can normally be zipped by you on the screen, but if you run text through more or use the **pg** command as a filter; the display stops once the screen is full of text.

```
$ls -l | grep "Aug" | sort +4n | more
-rw-rw-r--  1 carol doc      1605 Aug 23 07:35 macros
-rw-rw-r--  1 john  doc      2488 Aug 15 10:51 intro
-rw-rw-rw-  1 john  doc      8515 Aug  6 15:30 ch07
-rw-rw-r--  1 john  doc     14827 Aug  9 12:40 ch03
```

```
        .
        .
        .
-rw-rw-rw-  1 john  doc     16867 Aug  6 15:56 ch05
--More--(74%)
```

The screen will fill up once the screen is full of text consisting of lines sorted by the order of the file size. At the bottom of the screen is the **more** prompt, where you can type a command to move through the sorted text.

UNIX PROCESSES:

The operating system tracks processes through a five-digit ID number known as the **pid** or the **process ID**. Each process in the system has a unique **pid**.

Pids eventually repeat because all the possible numbers are used up and the next pid rolls or starts over. At any point of time, no two processes with the same pid exist in the system because it is the pid that Unix uses to track each process.

When you start a process (run a command), there are two ways you can run it −

- Foreground Processes
- Background Processes

Foreground Processes

By default, every process that you start runs in the foreground. It gets its input from the keyboard and sends its output to the screen.

You can see this happen with the **ls** command. If you wish to list all the files in your current directory, you can use the following command −

```
$ls ch*.doc
```

Background Processes

A background process runs without being connected to your keyboard. If the background process requires any keyboard input, it waits.

The advantage of running a process in the background is that you can run other commands; you do not have to wait until it completes to start another!

The simplest way to start a background process is to add an ampersand (**&**) at the end of the command.

```
$ls ch*.doc &
```

Listing Running Processes

It is easy to see your own processes by running the **ps** (process status) command as follows −

```
$ps
PID       TTY      TIME        CMD
18358     ttyp3    00:00:00    sh
18361     ttyp3    00:01:31    abiword
18789     ttyp3    00:00:00    ps
```

Stopping Processes

Ending a process can be done in several different ways. Often, from a console-based command, sending a CTRL + C keystroke (the default interrupt character) will exit the command. This works when the process is running in the foreground mode.

If a process is running in the background, you should get its Job ID using the **ps** command. After that, you can use the **kill** command to kill the process as follows −

```
$ps -f
UID      PID  PPID C STIME    TTY   TIME CMD
amrood   6738 3662 0 10:23:03 pts/6 0:00 first_one
amrood   6739 3662 0 10:22:54 pts/6 0:00 second_one
amrood   3662 3657 0 08:10:53 pts/6 0:00 -ksh
amrood   6892 3662 4 10:51:50 pts/6 0:00 ps -f
$kill 6738
Terminated
```

Parent and Child Processes

Each unix process has two ID numbers assigned to it: The Process ID (pid) and the Parent process ID (ppid). Each user process in the system has a parent process.

Most of the commands that you run have the shell as their parent. Check the **ps -f** example where this command listed both the process ID and the parent process ID.

VI EDITOR

There are many ways to edit files in Unix. Editing files using the screen-oriented text editor **vi** is one of the best ways. This editor enables you to edit lines in context with other lines in the file.

An improved version of the vi editor which is called the **VIM** has also been made available now. Here, VIM stands for **Vi IM**proved.

vi is generally considered the de facto standard in Unix editors because −

- It's usually available on all the flavors of Unix system.
- Its implementations are very similar across the board.
- It requires very few resources.
- It is more user-friendly than other editors such as the **ed** or the **ex**.

Starting the vi Editor

The following table lists out the basic commands to use the vi editor −

S.No.	Command & Description
	vi filename
1	Creates a new file if it already does not exist, otherwise opens an existing file.
	vi -R filename
2	

	Opens an existing file in the read-only mode.
	view filename
3	Opens an existing file in the read-only mode.

Following is an example to create a new file **testfile** if it already does not exist in the current working directory −

```
$vi testfile
```

Operation Modes

While working with the vi editor, we usually come across the following two modes −

- **Command mode** − This mode enables you to perform administrative tasks such as saving the files, executing the commands, moving the cursor, cutting (yanking) and pasting the lines or words, as well as finding and replacing. In this mode, whatever you type is interpreted as a command.
- **Insert mode** − This mode enables you to insert text into the file. Everything that's typed in this mode is interpreted as input and placed in the file.

Getting Out of vi

The command to quit out of vi is **:q**. Once in the command mode, type colon, and 'q', followed by return. If your file has been modified in any way, the editor will warn you of this, and not let you quit. To ignore this message, the command to quit out of vi without saving is **:q!**. This lets you exit vi without saving any of the changes.

The following points need to be considered to move within a file −

- vi is case-sensitive. You need to pay attention to capitalization when using the commands.
- Most commands in vi can be prefaced by the number of times you want the action to occur. For example, **2j** moves the cursor two lines down the cursor location.

There are many other ways to move within a file in vi.

Deleting Characters

Here is a list of important commands, which can be used to delete characters and lines in an open file −

S.No.	Command & Description
	x
1	Deletes the character under the cursor location
	X
2	

	Deletes the character before the cursor location
	dw
3	Deletes from the current cursor location to the next word
	d^
4	Deletes from the current cursor position to the beginning of the line
	d$
5	Deletes from the current cursor position to the end of the line
	D
6	Deletes from the cursor position to the end of the current line
7	**Dd** Deletes the line the cursor is on

Change Commands

You also have the capability to change characters, words, or lines in vi without deleting them. Here are the relevant commands −

S.No.	Command & Description
	cc
1	Removes the contents of the line, leaving you in insert mode.
	cw
2	Changes the word the cursor is on from the cursor to the lowercase **w** end of the word.
	r
3	Replaces the character under the cursor. vi returns to the command mode after the replacement is entered.
	R
4	Overwrites multiple characters beginning with the character currently under the cursor. You must use **Esc** to stop the overwriting.
	s
5	Replaces the current character with the character you type. Afterward, you are left in the insert mode.

	S
6	Deletes the line the cursor is on and replaces it with the new text. After the new text is entered, vi remains in the insert mode.

Copy and Paste Commands

You can copy lines or words from one place and then you can paste them at another place using the following commands −

S.No.	Command & Description
	yy
1	Copies the current line.
	yw
2	Copies the current word from the character the lowercase w cursor is on, until the end of the word.

TEXT MANIPULATION:

The `cat` command is one of the most basic commands. It is used to create, append, display, and concatenate files.

We can create a file with `cat` using the '>' to redirect standard input (`stdin`) to a file. Using the '>' operator truncates the contents of the output file specified. Text entered after that is redirected to the file specified to the right of the '>' operator. The control-d signals an end-of-file, returning control to the shell.

```
$ cat > grocery.list
apples
bananas
plums
<ctrl-d>
$
```

Use the '>>' operator to append standard input into an existing file.

```
$ cat >> grocery.list
carrots
<ctrl-d>
```

Use of wc

The `wc` (wordcount) command counts the number of lines, words (separated by whitespace), and characters in specified files, or from `stdin`.

```
$wc grocery.list
```

```
       4       4      29 grocery.list
```

Using grep

The `grep` command searches specified files or `stdin` for patterns matching a given expression(s). Output from `grep` is controlled by various option flags

```
$cat grocery.list2
Apple Sauce
wild rice
black beans
kidney beans
dry apples
```

Streams, pipes, redirects, tee

In UNIX a terminal by default contains three streams, one for input, and two output-based streams. The input stream is referred to as `stdin`, and is generally mapped to the keyboard (other input devices may be used, or could be piped from another process). The standard output stream is referred to as `stdout`, and generally prints to the terminal, or output may be consumed by another process (as `stdin`). The other output stream `stderr` primarily used for status reporting usually prints to the terminal like `stdout`. As each of these streams has its own file descriptor, each may be piped or redirected separately from the other, even if they are all connected to the terminal. File descriptors for each of these streams are:

- stdin = 0
- stdout = 1
- stderr = 2

These streams can be piped and redirected to files or other processes. This construct is commonly referred to as building a pipeline. For example, a programmer may want to merge the `stdout` and `stderr` streams, and then display them on the terminal but also save the results to a file to examine build issues. Using `2>&1` the `stderr` stream with file descriptor `2` is redirected to `&1`, a 'pointer' the `stdout` stream. This effectively merges `stderr` into `stdout`. Using the '|' symbol indicates a pipe. A pipe links `stdout` from the left-hand process (`make`) to `stdin` of the right-hand process (`tee`). The `tee` command duplicates the (merged) `stdout` stream sending the data to the terminal and to a file, in this example, called build.log.

```
$ make –f build_example.mk 2>&1 | tee build.log
```

Example of redirection to make a backup file:

```
$ cat < grocery.list > grocery.list.bak
```

Use of comm, cmp, and diff

To demonstrate these commands, two new files are created.

```
cat << EOF > dummy_file1.dat
011 IBM 174.99
012 INTC 22.69
013 SAP 59.37
```

```
014 VMW 102.92
EOF
cat << EOF > dummy_file2.dat
011  IBM 174.99
012 INTC 22.78
013 SAP 59.37
014 vmw 102.92
EOF
```

Using sort

To arrange rows in `stdin` or a file in a particular order, such as alphabetic or numeric, the `sort` command may be used. By default output from `sort` is written on `stdout`. Environment variables such as LC_ALL, LC_COLLATE, or LANG can affect the output of `sort` and other commands.

```
$ cat << EOF > dummy_sort1.dat

014 VMW, 102.92
013 INTC, 22.69
012 sap,  59.37
011 IBM, 174.99
011 IBM, 174.99
```

Using split

A useful task for the `split` command is to break large datafiles into smaller files for processing. In this example, BigFile.dat is shown to have 165782 lines using the `wc` command. The -l option flag tells `split` the maximum number of lines for each output file. `split` allows for a prefix to be specified for output filenames, below BigFile_ is the specified prefix. Other options allow for suffix control, and on BSD, a -p option flag allows for splits to occur at a regular expression like the `csplit` (context split) command.

```
$ wc BigFile.dat
165782  973580 42557440 BigFile.dat

$ split -l 15000 BigFile.dat BigFile_
```

AWK UTILITY:

awk - Finds and Replaces text, database sort/validate/index

SYNOPSIS

awk

DESCRIPTION

awk command searches files for text containing a pattern. When a line or text matches, awk performs a specific action on that line/text. The Program statement tells awk what operation to do; Program statement consists of a series of "rules" where each rule specifies one pattern to search for,

and one action to perform when a particular pattern is found. A regular expression enclosed in slashes (/) is an awk pattern to match every input record whose text belongs to that set.

```
$ ls -l | awk '{print $2}'
13
3
17
7
```