K S Institute of Technology

Department of Computer Science and Engineering

Welcome to

2020-21 Odd Semester

Online Class

On

UNIX Programming

18CS56

UNIX
Operating System

Syllabus: Module 2 – File Attributes and permissions

File attributes and permissions: The ls command with options. Changing file permissions: the relative and absolute permissions changing methods. Recursively changing file permissions. Directory permissions.
The shells interpretive cycle: Wild cards. Removing the special meanings of wild cards. Three standard files and redirection. **Connecting commands:** Pipe. Basic and Extended regular expressions. The grep, egrep. Typical examples involving different regular expressions.
Shell programming: Ordinary and environment variables. The .profile. Read and readonly commands. Command line arguments. exit and exit status of a command. Logical operators for conditional execution. The test command and its shortcut. The if, while, for and case control statements. The set and shift commands and handling positional parameters. The here (<<) document and trap command. Simple shell program examples.

File attributes and permissions

Basic File Attributes

The UNIX file system allows the user to access other files not belonging to them and without infringing on security. A file has a number of attributes (properties) that are stored in the inode. In this chapter,

we discuss,

- ls –l to display file attributes (properties)
- Listing of a specific directory
- Ownership and group ownership
- Different file permissions

Listing File Attributes

ls command is used to obtain a list of all filenames in the current directory. The output in UNIX lingo is often referred to as the listing.

It lists seven attributes of all files in the current directory and they are:

- File type and Permissions
- Links
- Ownership
- Group ownership
- File size
- Last Modification date and time
- File name

Listing of Files : ls command

For example,

- $ ls –l

```
total 72
-rw-r--r--    1  kumar  metal  19514  may  10  13:45          chap01
-rw-r--r--    1  kumar  metal   4174  may  10  15:01          chap02
-rw-rw-rw-  1  kumar  metal     84   feb  12   12:30          dept.lst
-rw-r--r--    1  kumar  metal   9156  mar  12    1999          genie.sh
drwxr-xr-x  2  kumar  metal    512  may   9   10:31          helpdir
drwxr-xr-x  2  kumar  metal    512  may   9   09:57          progs
```

1st field :The file type and its permissions are associated with each file.

2nd Field:Links indicate the number of file names maintained by the system(Single copy, Many Names)

3rd Field:File is created by the owner.

4th Field:Every user is attached to a group owner.

5th Field: File size in bytes is displayed.

6th Field: Last modification time is the next field.

If you change only the permissions or ownership of the file, the modification time remains unchanged.

7th field: In the last field, it displays the file name.

Listing Directory Attributes: ls -d

- ls –d

will not list all subdirectories in the current directory

For example

$ ls –ld helpdir progs

drwxr-xr-x 2 kumar metal 512 may 9 10:31 helpdir

drwxr-xr-x 2 kumar metal 512 may 9 09:57 progs

Here

The First letter **d** indicates directory

NOTE: There is no command to display only directories

File Ownership

- User who creates a file, become its owner.
- Every owner is attached to a group owner.
- Several users may belong to a single group, but the privileges of the group are set by the owner of the file and not by the group members.
- When the system administrator creates a user account, he has to assign these parameters to the user:

✓ The user-id (UID) – both its name and numeric representation

✓ The group-id (GID) – both its name and numeric representation

File Permissions

Filetype owner (rwx) groupowner (rwx) others (rwx)

Three-tiered file protection system that determines a file's access rights.

EX

-rwxr-xr-- 1 kumar metal 20500 may 10 19:21 chap02

Filetype – indicates ordinary file(Not a directory)

- r w x r - x r

Owner/user group owner others

File Permissions - Continued

- First group has all three permissions.
- The file is readable, writable and executable by the owner of the file.
- The second group has a hyphen in the middle slot, which indicates the absence of write permission by the group owner of the file.
- The third group has the write and execute bits absent. This set of permissions is applicable to others.

Changing File Permissions

A file or a directory is created with a default set of permissions, which can be determined by umask.

Ex:

$umask

022

Indicate the difference i.e. 666-644(Default file permissions)=022

Let us assume that the file permission for the created file is -rw-r--r--.

chmod command-Can change the file permissions and allow the owner to execute his file.

The command can be used in two ways:

- In a relative manner by specifying the changes to the current permissions
- In an absolute manner by specifying the final permissions

Changing File Permissions - Relative Permission

chmod category operation permission filename(s)

chmod takes an expression as its argument which contains:

- user category (user, group, others)
- operation to be performed (assign or remove a permission)
- type of permission (read, write, execute)

Changing File Permissions - Relative Permission (Continued)

Category	**operation**	**permission**
u - user	+ assign	r - read
g - group	- remove	w - write
o - others	= absolute	x - execute
a - all (ugo)		

Changing File Permissions - Relative Permission

- **$chmod ugo+x xstart or**
- **$chmod a+x xstart or**
- **$chmod +x xstart**
-
- **-rwxr-xr-x 1 kumar metal 1906 sep 23:38 xstart**
-
- **chmod accepts multiple file names in command line**
-
- **$chmod u+x note note1 note3**
-
- **Let initially,**
-
- **-rwxr-xr-x 1 kumar metal 1906 sep 23:38 xstart**
-
- **$chmod go-r xstart**
-
- **Then, it becomes**
-
- **-rwx--x--x 1 kumar metal 1906 sep 23:38 xstart**

Absolute permission

- Need not to know the current file permissions. We can set all nine permissions explicitly.
- A string of three octal digits is used as an expression.
- The permission can be represented by one octal digit for each category.

For each category, we add octal digits.

Absolute permission

- Read permission – 4 (octal 100)
- Write permission – 2 (octal 010)
- Execute permission – 1 (octal 001)
-

Octal	Permissions	Significance
0	- - -	no permissions
1	- - x	execute only
2	- w -	write only
3	- w x	write and execute
4	r - -	read only
5	r - x	read and execute
6	r w -	read and write
7	r w x	read, write and execute

777 indicates read, write and execute permission for all category i.e. for user/owner, group and for others. But still we can prevent a file from being deleted

000 signifies absence of all permissions for all categories, but still we can delete a file. It is the directory permissions that determine whether a file can be deleted or not

Comparison of commands using relative and absolute permissions

- **Using relative permission, we have,**
- **$chmod a+rw xstart**

- **Using absolute permission, we have,**

- **$chmod 666 xstart**

Other examples of absolute permissions

- **$chmod 644 xstart**
- **$chmod 761 xstart**

*** Note: The default file permissions are 644(may vary)**

Directory Permissions

*** Note : The default directory permissions are 755(May vary from system to system)**

- It is possible that a file cannot be accessed even though it has read permission, and can be removed even when it is write protected. Ex: The default permissions of a directory are,

  rwxr-xr-x (755)

- A directory must never be writable by group and others
- Example:

$mkdir c_progs

- $ls –ld c_progs

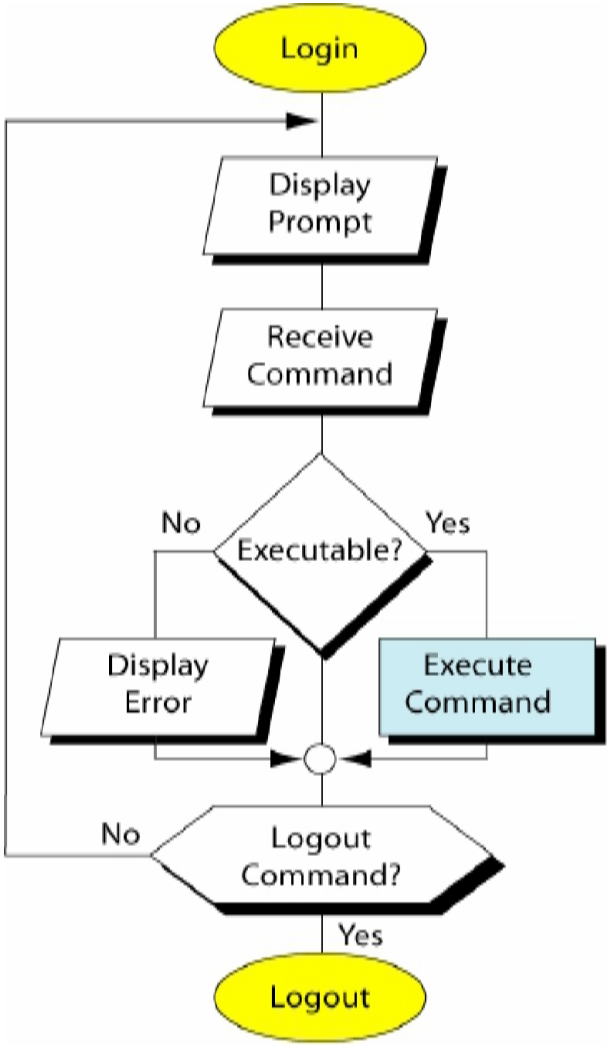
drwxr-xr-x 2 kumar metal 512 may 9 09:57 c_progs

The Shell

- **Introduction**
- Shell acts as both a command interpreter as well as a programming facility.

- **Objectives**
- The Shell and its interpretive cycle
- Pattern Matching – The wild-cards
- Escaping and Quoting
- Redirection – The three standard files
- Filters – Using both standard input and standard output
- /dev/null and /dev/tty – The two special files
- Pipes
- tee – Creating a tee
- Command Substitution
- Shell Variables

The shell and its interpretive cycle



Activities in Interpretive cycle

1. The shell issues the prompt and waits for you to enter a command.
2. After a command is entered, the shell scans the command line for metacharacters and expands abbreviations (like the * in rm *) to recreate a simplified command line.
3. It then passes on the command line to the kernel for execution.
4. The shell waits for the command to complete and normally can't do any work while the command is running.
5. After the command execution is complete, the prompt reappears and the shell returns to its waiting role to start the next cycle. You are free to enter another command.

Pattern Matching – The Wild-Cards

- **A pattern is framed using ordinary characters and a metacharacter (like *) using well -defined rules**
- **pattern can then be used as an argument to the command, and the shell will expand it suitably before the command is executed**

Wild-Card	Matches
*	Any number of characters including none
?	A single character
[ijk]	A single character – either an i, j or k
[x-z]	A single character that is within the ASCII range of characters x and z
[!ijk]	A single character that is not an i,j or k (Not in C shell)
[!x-z]	A single character that is not within the ASCII range of the characters x and z (Not in C Shell)
{pat1,pat2…}	Pat1, pat2, etc. (Not in Bourne shell)

Wild cards examples

- **To list all files that begin with *modu*, use**

  $ ls Modu*

- **To list all files whose filenames are six character long and start with modu, use**

  $ ls modu??

- **Note: Both * and ? operate with some restrictions. for example,**

  the * doesn't match all files beginning with a . (dot) or the / of a pathname.

  If you wish to list all hidden filenames in your directory having at least three characters after the dot, the dot must be matched explicitly.

  $ ls .???*

- **However, if the filename contains a dot anywhere but at the beginning, it need not be matched explicitly.**

- **Similarly, these characters don't match the / in a pathname. So, you cannot use**

  $ cd /usr?local to change to /usr/local.

The character class

The character class comprises a set of characters enclosed by the rectangular brackets, [and], but it matches a single character in the class. The pattern [abd] is character class, and it matches a single character – an a,b or d.

- **Examples:**
- $ls chap0[124] Matches chap01, chap02, chap04 and lists if found.
- $ ls chap[x-z] Matches chapx, chapy, chapz and lists if found.
- You can negate a character class to reverse a matching criteria. For example,
- - To match all filenames with a single-character extension but not the .c or .o files,

use *.[!co]

- - To match all filenames that don't begin with an alphabetic character,

use [!a-zA-Z]*

Matching totally dissimilar patterns

- This feature is not available in the Bourne shell. To copy all the C and Java source programs from another directory, we can delimit the patterns with a comma and then put curly braces around them.

  $ cp $HOME/prog_sources/*.{c,java} .
- The Bourne shell requires two separate invocations of cp to do this job.

  $ cp /home/srm/{project,html,scripts}/* .
- The above command copies all files from three directories (project, html and scripts) to the current directory.

Escaping and Quoting (Removing the special meanings of wild cards)

- Escaping is providing a \ (backslash) before the wild-card to **remove (escape)** its special meaning.
- For instance, if we have a file whose filename is chap* (Remember a file in UNIX can be names with virtually any character except the / and null), to remove the file, it is dangerous to give command as *rm chap**, as it will remove all files beginning with chap. Hence to suppress the special meaning of *, use the command *rm chap**
- To list the contents of the file chap0[1-3], use $ cat chap0\[1-3\]
- A filename can contain a whitespace character also. Hence to remove a file named
- My Documend.doc, which has a space embedded, a similar reasoning should be followed: $ rm My\ Document.doc

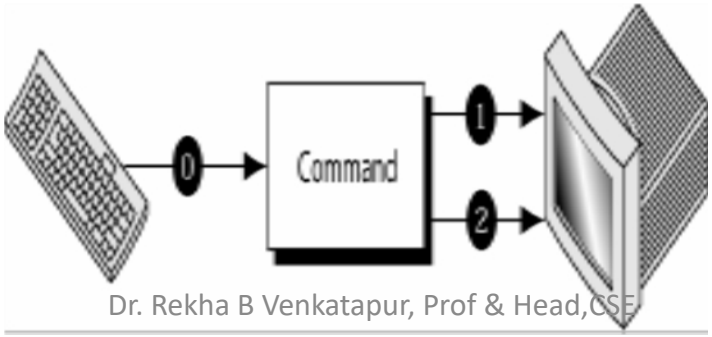
- **Quoting** is enclosing the wild-card, or even the entire pattern, within quotes. Anything within these quotes (barring a few exceptions) are left alone by the shell and not interpreted.
- When a command argument is enclosed in quotes, the meanings of all enclosed special characters are turned off.
- Examples:

$ rm ‘chap*’ //Removes fil *chap**

$ rm “My Document.doc”//Removes file *My Document.doc*

Redirection : The three standard files

- The shell associates three files with the terminal – two for display and one for the keyboard.
- These files are streams of characters which many commands see it as input and output. When a user logs in, the shell makes available three files representing three streams. Each stream is associated with a default device:
- **Standard input:** The file (stream) representing input, connected to the keyboard.
- **Standard output:** The file (stream) representing output, connected to the display.
- **Standard error:** The file (stream) representing error messages that emanate from the command or shell, connected to the display**.**



- **The standard input can represent three input sources:**

1. The keyboard, the default source.
2. A file using redirection with the < symbol.
3. Another program using a pipeline.

Example:

$wc

Abcdes

7 1 1

$wc < abc.c

12 3 1

$ cat – abc.c // First from standard input and then from abc.c

$ cat emp1.lst – abc.c // First from emp1.lst, then standard input and then abc.c

- **The standard output can represent three possible destinations:**

1. The terminal, the default destination.
2. A file using the redirection symbols > and >>(append).
3. As input to another program using a pipeline.

```
$ wc emp1.lst > newdata
$ cat newdata
    1  1 16 emp1.lst
$ wc xyz.c >> newdata
$ cat newdata
 1  1 16 emp1.lst
 6  8 65 xyz.c
```

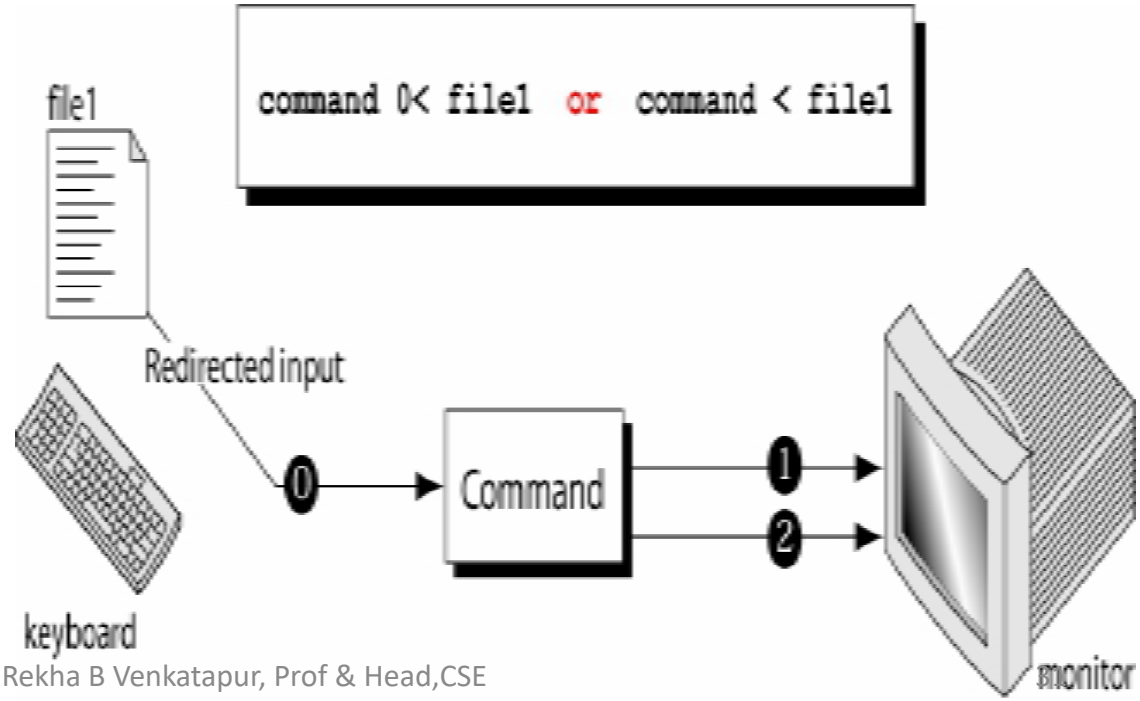
- Concatenated output stream :
- $ (ls *.c ; echo ; cat *.c) > c_progs_all.txt

- Every file is identified by file descriptor.
- A file is opened by referring to its pathname, but subsequent read and write operations identify the file by a unique number called a file descriptor.
- The kernel maintains a table of file descriptors for every process running in the system.
- The first three slots are generally allocated to the three standard streams as,

0 – Standard input

1 – Standard output

2 – Standard error



Examples

```
$wc 0< abc.c
    4 7 21
$wc 0< abc.c 1> new
$cat new
    4 7 21
$cat foo 2>error
$cat error
   cat: foo: No such file or directory
```

- Assuming file2 doesn't exist, the following command redirects the standard output to file *myOutput* and the standard error to file *myError*.

$ ls –l file1 file2 1>myOutput 2>myError

Chapter 13: Filters Using Regular Expression—grep and sed

Filters: Using both standard input and standard output

- UNIX commands can be grouped into four categories viz.,

1. **Directory-oriented commands** like mkdir, rmdir and cd, and basic file handling commands like cp, mv and rm use neither standard input nor standard output.
2. Commands like ls, pwd, who etc. **don't read standard input but they write to standard output.**
3. Commands like lp that **read standard input but don't write to standard output.**
4. Commands like cat, wc, cmp etc. that use **both standard input and standard output.**

- Commands in the fourth category are called filters. Note that filters can also read directly from files whose names are provided as arguments.
- Example: To perform arithmetic calculations that are specified as expressions in input file calc.txt and redirect the output to a file result.txt, use **calc. txt**

$cat >calc.txt

3*4

9-6

8/2

^d

$ bc < calc.txt > result.txt

$cat result.txt

12 3 4

Pipes: Connecting Commands

- With piping, the output of a command can be used as input (piped) to a subsequent command.

  $ command1 | command2

- Output from command1 is piped into input for command2.
- This is equivalent to, but more efficient than:
- **$ command1 > temp**
- **$ command2 < temp**
- **$ rm temp**
-
- **Examples**
- **$ ls | wc**
- **$ who | sort**

When a command needs to be ignorant of its source

- If we wish to find total size of all C programs contained in the working directory, we can use the command,
- **$ wc –c *.c**
- However, it also shows the usage for each file(size of each file).
- We are not interested in individual statistics, but a single figure representing the total size. To be able to do that, we must make wc ignorant of its input source. We can do that by feeding the concatenated output stream of all the .c files to wc –c as its input:
- **$ cat *.c | wc –c**

Basic and Extended regular expressions

- We often need to search a file for a pattern, either to see the lines containing (or not containing) it or to have it replaced with something else.
- This chapter discusses two important filters that are especially suited for these tasks –
- grep and sed.
- grep takes care of all search requirements we may have.
- sed goes further and can even manipulate the individual characters in a line.

grep – searching for a pattern

- Grep- global regular expression print
- It scans the file / input for a pattern and displays lines containing the pattern, the line numbers or filenames where the pattern occurs.
- It's a command from a special family in UNIX for handling search requirements.

$cat emp.lst

2233 |a.|jai sharma |director |production|12/03/50 | 7000
5678 |sumit |d.g.m |marketing |19/04/43 | 7800
2365 |barun sengupta|director |personnel |11/04/47 | 5400
1265 |s.n. dasgupta|manager |sales |12/09/63 | 5600

- **grep *options pattern filename(s)***

- $grep "sales" emp.lst
- will display lines containing sales from the file emp.lst. Patterns with and without quotes is possible. It's generally safe to quote the pattern. Quote is mandatory when pattern involves more than one word. It returns the prompt in case the pattern can't be located.

- $grep director emp.lst

- When grep is used with multiple filenames, it displays the filenames along with the output

  $grep “director” emp1.lst emp2.lst

- Where it shows filename followed by the contents

grep options

- grep is one of the most important UNIX commands, and we must know the options that POSIX requires grep to support. Linux supports all of these options.
- -i ignores case for matching
- -v doesn’t display lines matching expression
- -n displays line numbers along with lines
- -c displays count of number of occurrences
- -l displays list of filenames only
- -e exp specifies expression with this option
- -x matches pattern with entire line
- -f file takes patterns from file, one per line
- -E treats pattern as an extended RE
- -F matches multiple fixed strings

Basic Regular Expressions (BRE) – An Introduction

- It is tedious to specify each pattern separately with the -e option.
- grep uses an expression of a different type to match a group of similar patterns.
- If an expression uses meta characters, it is termed a regular expression.
- Some of the characters used by regular expression are also meaningful to the shell.

BRE Character subset

Symbols or Expression	Matches
*	Zero or more occurrences of the previous character
g*	Nothing or g, gg, ggg, gggg, etc.
.	A single character
.*	Nothing or any number of characters
[pqr]	A single character p, q or r
[c1-c2]	A single character withing ASCII range shown by c1 and c2
[0-9]	A digit between 0 and 9
[^pqr]	A single character which is not a p, q or r
[^a-zA-Z]	A non-alphabetic character
^pat	Pattern pat at beginning of line
pat$	Pattern pat at end of line
^bash$	A bash as the only word in line
^$	Lines containing nothing

The character class

- grep supports basic regular expressions (BRE) by default and extended regular expressions (ERE) with the –E option.
- A regular expression allows a group of characters enclosed within a pair of [ ], in which the match is performed for a single character in the group.
    - $grep “[aA]g[ar][ar]wal” emp.lst

- **The ***

- The asterisk refers to the immediately preceding character. * indicates zero or more occurrences of the previous character.

- g* nothing or g, gg, ggg, etc.

- $grep “[aA]gg*[ar][ar]wal” emp.lst

- Notice that we don’t require to use –e option three times to get the same output!!!!!

- **The dot**

- A dot matches a single character. The shell uses ? Character to indicate that.

- .* signifies any number of characters or none

- $grep “j.*saxena” emp.lst

- **Specifying Pattern Locations (^ and $)**
- Match a pattern at the beginning or end of a line. Anchoring a pattern is often necessary when it can occur in more than one place in a line, and we are interested in its occurrence only at a particular location.
- ^ for matching at the beginning of a line
- $ for matching at the end of a line

$grep “^2” emp.ls

- Selects lines where emp_id starting with 2

$grep “7…$” emp.lst

- Selects lines where emp_salary ranges between 7000 to 7999

$grep “^[^2]” emp.lst

- Selects lines where emp_id doesn’t start with 2

- **When meta characters lose their meaning**
- It is possible that some of these special characters actually exist as part of the text. Sometimes, we need to escape these characters. For example, when looking for a pattern g*, we have to use \
- To look for [, we use \[
- To look for .*, we use \.*

Extended Regular Expression (ERE) and grep

- If current version of grep doesn't support ERE, then use egrep but without the –E option. -E option treats pattern as an ERE.
- + matches one or more occurrences of the previous character
- ? Matches zero or one occurrence of the previous character
- b+ matches b, bb, bbb, etc.
- b? matches either a single instance of b or nothing
- These characters restrict the scope of match as compared to the *

$grep –E "[aA]gg?arwal" emp.lst
?include +<stdio.h>

The ERE set

Expression	Significance
ch+	Matches one or more occurrences of character ch
ch?	Matches zero or one occurrence of character ch
exp1 | exp2	Matches exp1 or exp2
GIF | JPEG	Matches GIF or JPEG
(x1|x2)x3	Matches x1x3 or x2x3
(hard|soft)ware	Matches hardware or software

- **Matching multiple patterns (|, (and))**

$grep –E ‘sengupta|dasgupta’ emp.lst

- We can locate both without using –e option twice, or

$grep –E ‘(sen|das)gupta’ emp.lst

Chapter 14: Essential Shell Programming

The Shell

- The UNIX shell is both an interpreter as well as a scripting language. An interactive shell turns noninteractive when it executes a script.
- **Bourne Shell** – This shell was developed by Steve Bourne. It is the original UNIX shell. It has strong programming features, but it is a weak interpreter.
- **C Shell** – This shell was developed by Bill Joy. It has improved interpretive features, but it wasn't suitable for programming.
- **Korn Shell** – This shell was developed by David Korn. It combines best features of the bourne and C shells. It has features like aliases, command history. But it lacks some features of the C shell.
- **Bash Shell** – This was developed by GNU. It can be considered as a superset that combined the features of Korn and C Shells. More importantly, it conforms to POSIX shell specification.

Environment Variables

- We already mentioned a couple of environment variables, such as PATH and HOME.
- Until now, we only saw examples in which they serve a certain purpose to the shell.
- What other information do programs need apart from paths and home directories?
- A lot of programs want to know about the kind of terminal you are using – TERM variable.
- The shell you are using --SHELL variable
- The operating system type -- OS and so on.
- A list of all variables currently defined for your session can be viewed entering the **env** command

- The environment variables are managed by the shell.
- Environment variables are inherited by any program you start, including another shell.
- The set statement display all variables available in the current shell, but env command displays only environment variables.
- env is an external command and runs in a child process.

The Common Environment Variables

Variable name	Stored information
HISTSIZE	size of the shell history file in number of lines
HOME	path to your home directory
HOSTNAME	local host name
LOGNAME	login name
MAIL	location of your incoming mail folder
MANPATH	paths to search for man pages
PATH	search paths for commands
PS1	primary prompt
PS2	secondary prompt
PWD	present working directory
SHELL	current shell
TERM	terminal type
UID	user ID
USER	Login name of user
MAILCHECK	Mail checking interval for incoming mail
CDPATH	List of directories searched by cd when used with a non-absolute pathname

Shell Programming

- **Shell Script**: A group of commands to be executed regularly are stored in a file and the file itself executed as a shell script or shell program.
- Shell programs are executed in separate child shell process.
- You can use another shell to run script.
- Your login shell may be korn and script shell may be bash.
- Special interpreter line in the first line of the script is used to specify different shell for your script.
- Use vi editor to create the shell script.

Sample Shell Script

```
#!/bin/sh
# Script : Sample Shell Script
echo "Today's date: “
date
echo "This month's calendar:"
cal
echo "My Shell : $SHELL“
```

Output:

```
$ sh first.sh
Today's date: Tue, Oct 13, 2020 10:27:38 AM
This month's calendar:
    October 2020
Su Mo Tu We Th Fr Sa
             1  2  3
 4  5  6  7  8  9 10
11 12 13 14 15 16 17
18 19 20 21 22 23 24
25 26 27 28 29 30 31

My Shell : /bin/bash
```

read: MAKING SCRIPT INTERACTIVE

- The read statement is the shell's internal tool for taking input from the user, i.e. making script interactive.
- **Example:**

```
#!/bin/sh
#Sample Shell Script - simple.sh
echo "Enter Your First Name:"
read fname
echo "Enter Your Last Name:"
read lname
echo "Your First Name is: $fname"
echo " Your Last Name is: $lname"
```

- **Example: Search a pattern in a file**

```
#!/bin/sh
#Sample Shell Script - simple.sh
echo "Enter Pattern to be searched:"
read pname
echo "Enter file to be used:"
read flname
echo "Searching for  $pname from file $flname"
grep “$pname” $flname
echo " The selected records are shown above"
```

USING COMMAND LINE ARGUMENTS

- Shell scripts also accept arguments from the command line.
- They can, therefore, run non-interactively and be used with redirection and pipelines.
- When arguments are specified with a shell script, they are assigned to positional parameters.
- The shell uses following parameters to handle command line arguments-

Shell parameter	Significance
$#	Number of arguments specified in command line
$0	Name of executed command
$1, $2, …	Positional parameters representing command line arguments
$*	Complete set of positional parameters as a single string
"$@"	Each quoted string is treated as a separate arguments, same as $*

- **Example: Using Command line arguments**

```
#!/bin/sh
#Shell Script to demonstrate command line arguments - sample.sh
echo "The Script Name is: $0"
echo "Number of arguments specified is: $#"
echo "The arguments are: $*"
echo "First Argument is: $1"
echo "Second Argument is: $2"
echo "Third Argument is: $3"
echo "Fourth Argument is: $4"
echo "The arguments are: $@"
```

- **Execution & Output:**
- $ sh sample.sh welcome to hit nidasoshi [Enter]

```
The Script Name is: sample.sh
Number of arguments specified is: 4
The arguments are: welcome to hit nidasoshi
First Argument is: welcome
Second Argument is: to
Third Argument is: hit
Fourth Argument is: nidasoshi
The arguments are: welcome to hit nidasoshi
```

exit AND EXIT STATUS OF COMMAND

- C program and shell scripts have a lot in common, and one of them is that they both use the same command (or function in c) to terminate a program.It has the name exit in the shell and exit( ) in C.
- The command is usually run with numeric arguments:
- **exit 0 #Used when everything went fine**
- **exit 1 #Used when something went wrong**
-

Exit....

- The shell offers a variable $? and a command test that evaluates a command's exit status.
- The parameter $? stores the exit status of the last command.
- It has the value 0 if the command succeeds and a non-zero value if it fails.
- This parameter is set by exit's argument.
- Examples:
- $ grep director emp.lst >/dev/null; echo $?
- 0 #Success
- $ grep director emp.lst >/dev/null; echo $?
- 1 #Failure – in finding pattern

THE LOGICAL OPERATORS && AND || - CONDITIONAL EXECUTION

- The shell provides two operators that allow conditional execution – the && and | |.

Examples:

$ date && echo “Date Command Executed Successfully!”
Sun Jan 13 15:40:13 IST 2013
Date Command Executed Successfully!

$ grep 'director' emp.lst && echo “Pattern found in File!”
1234 | Henry Ford | director | Marketing | 12/12/12 | 25000
Pattern found in File!

$ grep 'manager' emp.lst | | echo “Pattern not-found in File!”
Pattern not-found in File!

USING test AND [ ] TO EVALUATE EXPRESSIONS

- When you use if to evaluate expressions, you need the test statement because the true or false values returned by expression's can't be directly handled by if.
- test uses certain operators to evaluate the condition on its right and returns either a true or false
- exit status, which is then used by if for making decision.
- test works in three ways:
  1. Compares two numbers
  2. Compares two strings or a single one for a null value.
  3. Checks a file's attributes
- test doesn't display any output but simply sets the parameter $?.

Numeric Comparison

- Numerical Comparison operators used by test:

Operator	Meaning
-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

- The numerical comparison operators used by test always begins with a - (hyphen), followed by a two-letter string, and enclosed on either side by whitespace.

- **Examples:**

$ x=5, y=7, z=7.2

$ test $x -eq $y; echo $?

1 # Not Equal

$ test $x -lt $y; echo $?

0 #True

$ test $y -eq $z

0 #True- 7.2 is equal to 7

- The last example proves that numeric comparison is restricted to integers only.
- The [ ] is used as shorthand for test.
- Hence, above example may be re-written as test
- $x -eq $y or [$x -eq $y] #Both are equivalent

THE if CONDITIONAL

- The if statement makes two-way decisions depending on the fulfillment of a certain condition.
- In the shell, the statement uses the following forms

Form 1	Form 2	Form 3
if command is successful then execute commands else execute commands fi	If command is successful then execute commands fi	If command is successful then execute commands elif command is successful then execute commands else execute commands fi

- **Example:**

```
#!/bin/sh
#Shell script to illustrate if conditional
if grep 'director' emp.lst >/dev/null
then
echo “Pattern found in File!”
else
echo “Pattern not-found in File!”
fi
```

String Comparison

- test can be used to compare strings with yet another set of operators. The below table shows string tests used by test-

Test	True if
s1 = s2	String s1 = s2
s1 != s2	String s1 is not equal to s2
-n stg	String stg is not a null string
-z stg	String stg is a null string
stg	String stg is assigned and not a null string
s1 == s2	String s1 = s2 (Korn and Bash only)

Example:

```
#!/bin/sh
#Shell script to illustrate string comparison using test – strcmp.sh
if [ $# -eq 0 ]; then
echo "Enter Your Name: \c"; read name;
if [-z $name ]; then
echo "You have not entered your name! "; exit 1;
else
echo "Your Name From Input line is : $name "; exit 1;
fi
else
echo "Your Name From Command line is : $1 " ;
fi
```

Execution & Output:

$ chmod 777 strcmp.sh

- $ sh strcmp.sh Henry
- Your Name From Command line is : Henry
- $ sh strcmp.sh
- Enter Your Name: Smith [Enter]
- Your Name From Input line is : Smith
- $ sh strcmp.sh
- Enter Your Name: [Enter]
- You have not entered your name!

THE case CONDITIONAL

- The case statement is similar to switch statement in C.
- The statement matches an expression for more than one alternative, and permit multi-way branching.
- The general syntax of the case statement is as follows:

```
case expression in
pattern1) command1 ;;
pattern2) command2 ;;
pattern3) command3 ;;
……..
esac
```

Example:

```
#!/bin/sh
#Shell script to illustrate CASE conditional – menu.sh
echo "\t MENU\n 1. List of files\n 2. Today's Date\n 3. Users of System\n 4. Quit\n";
echo "Enter your option: \c";
read choice
case "$choice" in
1) ls -l ;;
2) date ;;
3) who ;;
4) exit ;;
*) echo "Invalid Option!"
esac
```

Execution & Output:

- $ chmod 777 menu.sh
- $ sh menu.sh
- MENU
- 1. List of files
- 2. Today's Date
- 3. Users of System
- 4. Quit
- Enter your option: 2
- Sun Jan 13 15:40:13 IST 2013

while: LOOPING

- The while statement should be quite familiar to many programmers.
- It repeatedly performs a set of instructions until the control commands returns a true exit status.

- **The general syntax of this command is as follows:**

while condition is true
do
commands
done

- The commands enclosed by do and done are executed repeatedly as long as condition remains true.

Example:

```
#!/bin/sh
#Shell script to illustrate while loop – while.sh
answer=y;
while [ "$answer" = "y" ]
do
echo "Enter Branch Code and Name: \c"
read code name #Read both together
echo "$code|$name" >> newlist #Append a line to newlist
echo "Enter anymore (y/n)? \c"
read anymore
case $anymore in
y*|Y*) answer=y ;; #Also accepts yes, YES etc.
n*|N*) answer=n ;; #Also accepts no, NO etc.
*) answer=n ;; #Any other reply means no
esac
done
```

Execution & Output:

- $ chmod 777 while.sh
- $ sh while.sh
- Enter Branch Code and Name: CS COMPUTER [Enter]
- Enter anymore (y/n)? y [Enter]
- Enter Branch Code and Name: EC ELECTRONICS [Enter]
- Enter anymore (y/n)? n [Enter]
- $ cat newlist
- CS|COMPUTER
- EC|ELECTRONICS

for: LOOPING WITH A LIST

- The shell's for loop differs in structure from the ones used in other programming language.
- Unlike while and until, for doesn't test a condition, but uses a list instead.
- **The general syntax of for loop is as follows**

for
variable in list
do
commands
done

- The loop body also uses the keywords do and done, but the additional parameters here are variable and list.
- Each whitespace-separated word in list is assigned to variable in turn, and commands are executed until list is exhausted.

Example:

```
#!/bin/sh
#Shell script to illustrate use of for loop – forloop.sh
for file in chap1 chap2 chap3 chap4
do
cp $file ${file}.bak
echo "$file copied to $file.bak"
done
```

Execution & Output:

- $ chmod 777 forloop.sh
- $ sh forloop.sh
- chap1 copied to chap1.bak
- chap2 copied to chap2.bak
- chap3 copied to chap3.bak
- chap4 copied to chap4.bak

Possible sources of the list:

1. list from variables - $ for var in $x $y $z
2. list from command substitution- $ for var in `cat foo`
3. list from wild-cards- $ for file in *.htm *.html
4. list from positional parameters- $ for var in “$@”

set AND shift: MANIPULATING THE POSITIONAL PARAMETERS

- set: Set the positional parameters
- set assigns its arguments to the positional parameters $1, $2 and so on.
- This feature is especially useful for picking up individual fields from the output of a program.

- Example:

$ set `date` #Output of date command assigned to positional parameters $1, $2 & so on.
$ echo $*
Sun Jan 13 15:40:13 IST 2013
$ echo “The date today is $2 $3 $6”
The date today is Jan 13 2013

shift: Shifting Arguments Left

- shift transfers the contents of a positional parameters to its immediate lower numbered one.
- This is done as many times as the statement is called.
- Example:
- $ set `date`
- $ echo $*
- Sun Jan 13 15:40:13 IST 2013
- $ echo $1 $2 $3
- Sun Jan 13
- $ shift #Shifts 1 place
- Jan 13 15:40:13
- $ echo $1 $2 $3
- $ shift 2 #Shifts 2 places
- $ echo $1 $2 $3
- 15:40:13 IST 2013

The Here Document (<<)

- The shell uses the << symbol to read data from the same file containing the script. This is referred to as a here document, signifying that the data is here rather than in aspirate file. Any command using standard input can slo take input from a here document.
- Example:

mailx kumar << MARK
Your program for printing the invoices has been executed on `date`.Check the print queue
The updated file is

$flname MARK

- The string (MARK) is delimiter. The shell treats every line following the command and delimited by MARK as input to the command. Kumar at the other end will see three lines of message text with the date inserted by command. The word MARK itself doesn't show up.
- Using Here Document with Interactive Programs:
- A shell script can be made to work non-interactively by supplying inputs through here
- document.
- Example:
- $ search.sh <<
- END > director
- >emp.lst
- >END
- **Output:**
- Enter the pattern to be searched: Enter the file to be used: Searching for director from file
- emp.lst
- 9876 Jai Sharma Director Productions
- 2356 Rohit Director Sales
- Selected records shown above.
- The script search.sh will run non-interactively and display the lines containing "director" in the file emp.lst.

trap: interrupting a Program

- Normally, the shell scripts terminate whenever the interrupt key is pressed.
- It is not a good programming practice because a lot of temporary files will be stored on disk.
- The trap statement lets you do the things you want to do when a script receives a signal.
- The trap statement is normally placed at the beginning of the shell script and uses two lists:

- trap ‘command_list’ signal_list
- When a script is sent any of the signals in signal_list, trap executes the commands in command_list. The signal list can contain the integer values or names (without SIG prefix) of one or more signals – the ones used with the kill command.
- Example: To remove all temporary files named after the PID number of the shell:
- trap ‘rm $$* ; echo “Program Interrupted” ; exit’ HUP INT TERM trap is a signal
- handler. It first removes all files expanded from $$*, echoes a message and finally
- terminates the script when signals SIGHUP (1), SIGINT (2) or SIGTERM(15) are sent to the shell process running the script.
- A script can also be made to ignore the signals by using a null command list.
- Example:
- trap ‘’ 1 2 15