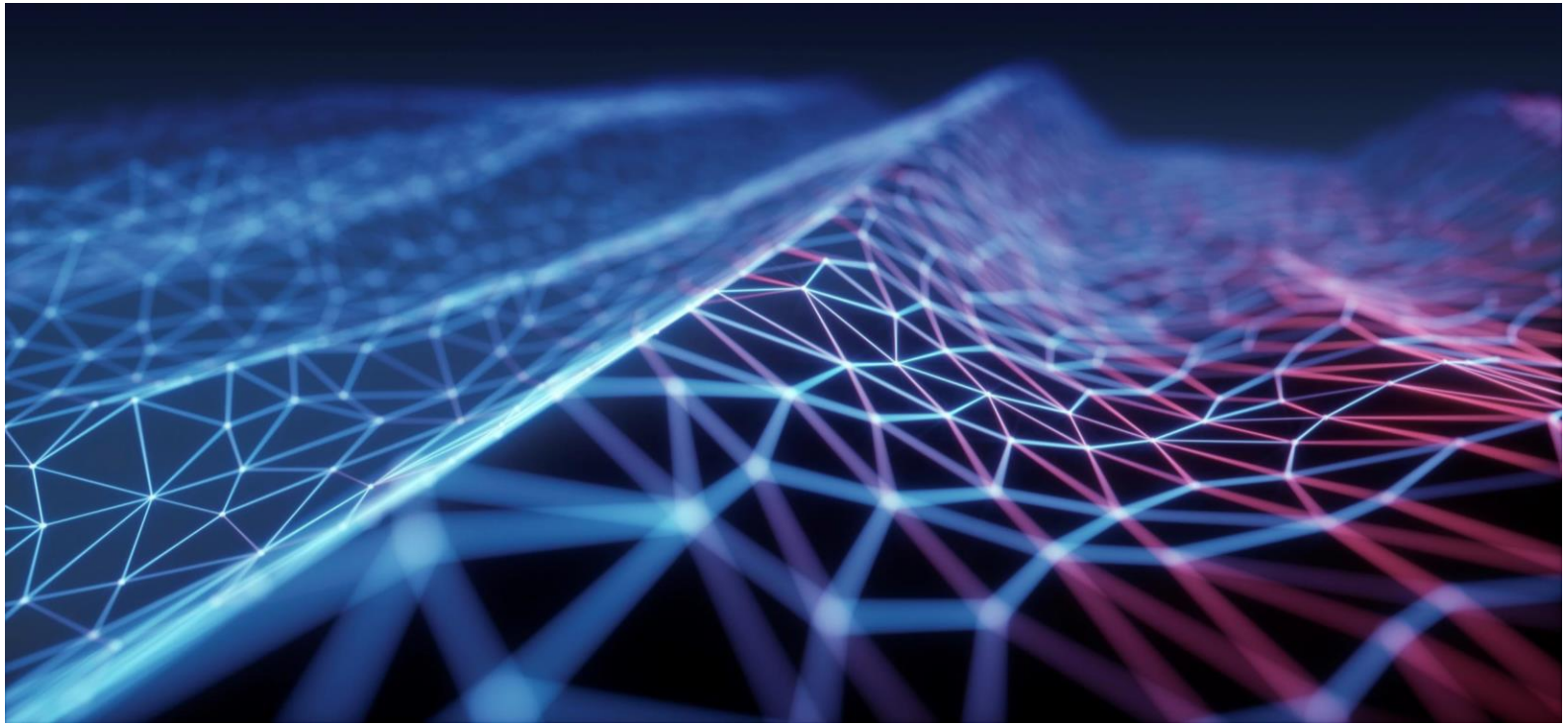

Vidyayāmruthamashnuthe

UNIX PROGRAMMING – MODULE 2

AKASH HEGDE

ASSISTANT PROFESSOR

DEPARTMENT OF ISE

MODULE 2

File Attributes and Permissions	Listing files – "ls" command
	File permissions
	Directory Permissions
The Shell	Wild cards
	Removing the special meaning of wild cards
	Three standard files
Connecting Commands	Pipes
	Regular Expressions
Programming the Shell	Variables
	Command line arguments
	Control statements
	Positional parameters

LISTING FILE ATTRIBUTES – LS COMMAND

- **ls** – listing files in a directory.
- **ls -l** used for long listing of files – displays several attributes of a file.
  - Example: **ls -l chap1 chap2 chap3**
- inode – data structure that stores the details of a file (number).
- **ls** fetches details of files from their respective inodes.
- List is preceded by total count of number of blocks occupied by the files on disk. Each block consists of 512 bytes (1024 bytes in Linux).

LISTING FILE ATTRIBUTES – LS COMMAND

Attribute	Description
File Type and Permissions	Type and permissions associated with the file. **d** (directory), **b** or **c** (device), **r** (read), **w** (write), **x** (execute)
Links	Number of links associated with the file. Greater than one – same file has more than one name.
Ownership	Owner of the file – full authority to change content and permissions.
Group Ownership	Group that owns the file – work on the same file.
File Size	Size of the file in bytes – only a character count and not the occupied disk space.
Last Modification Time	Last modification time of the file – content must change, not permissions/ownership.
Filename	Name of the file in ASCII collating sequence.

LISTING FILE ATTRIBUTES – LS COMMAND

- **ls -ld** – displays only directories in long listing.
- Must use **–ld** for directory long listing. No way to display only directories using **–d**
- **ls -li** – displays inode numbers of files in long listing.

FILE OWNERSHIP

- Create a new file – you are the owner of the file.
- Group can access your file – group owner of the file.
- Copy another file – you are the owner of the copy.
- Cannot create files in another user's directories - you do not own those directories and do not have write access.
- Group privileges set by owner of the shared file and not by group members.

FILE OWNERSHIP

- User ID (UID) –
  - name and numeric representation of the user.
  - maintained in /etc/passwd
- Group ID (GID) –
  - name and numeric representation of the group.
  - maintained in /etc/group (and /etc/passwd – only number)
- View UID and GID – **id** command

FILE PERMISSIONS

- Categorical distribution of file access rights.
- 3 categories – owner (*user*), group owner, others (*the world*).
- Permissions – **r** (read), **w** (write), **x** (execute).
- Owner's permissions override the group's permissions when one has ownership of the associated file.

CHANGING FILE PERMISSIONS – CHMOD COMMAND

- File or directory created with a set of default permissions.
- **touch** command – used to just create a file.
  - Example: **touch newfile**
- **chmod** – change mode.
- Set the permissions of one or more files for all three categories of users (user, group and others).

  (Note: user and owner are used interchangeably. Both are same.)

CHANGING FILE PERMISSIONS – CHMOD COMMAND

- Can only be run by the user (owner) of the file and the superuser.
- **chmod** 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.

RELATIVE PERMISSIONS

- **chmod** only changes the permissions specified in the command line.
- Other permissions are left unchanged.
- Syntax: *chmod category operation permission*
- Argument – expression comprising letters and symbols that describe user category and type of permission being assigned or removed.

RELATIVE PERMISSIONS

- Components of the specified expression in **chmod** –
  - *category* – the user category (user, group, others)
  - *operation* – the operation to be performed (assign / remove permission)
  - *permission* – the type of permission (read, write, execute)
- Suitable abbreviations for the components are used, compact expressions are framed and then used as argument to **chmod** command.

RELATIVE PERMISSIONS

- Always better to check the permissions using **ls –l** after using **chmod**.
- Assigning relative permissions (**+** operator) –
  - **chmod u+x newfile** – assigning execute permission to the user of *newfile*
  - **chmod ugo+x newfile** – assigning execute permission to user, group and others
  - **chmod a+x newfile** – assigning execute permission to all (identical to 2nd)
  - **chmod u+w file1 file2 file3** – assigning permissions to multiple files at once

RELATIVE PERMISSIONS

- Removing relative permissions (**–** operator) –
  - **chmod u–x newfile** – removing execute permission from the user of *newfile*
  - **chmod ugo–x newfile** – removing execute permission from user, group and others
  - **chmod a–x newfile** – removing execute permission from all (identical to 2nd)
  - **chmod u–w file1 file2 file3** – removing permissions from multiple files at once

RELATIVE PERMISSIONS

- Multiple permissions can also be set at once.
  - **chmod o+wx newfile** – assigning write and execute permission to others
- Multiple expressions can also be used at once, delimited by commas.
  - **chmod ug+x,g-w,o+x newfile** – assigning execute permission to all three categories and removing write permission from group.

RELATIVE PERMISSIONS

Abbreviation	Description
Category	u – user g – group o – others a – all (*ugo*)
Operation	+ – assigns permission - – removes permission = – assigns absolute permission
Permission	r – read permission w – write permission x – execute permission

ABSOLUTE PERMISSIONS

- **chmod** can be used to change absolute permissions without knowing the current permissions of the file.
- The expression – string of three octal numbers (base 8).
- Octal digits range from 0 to 7 – set of three bits can represent one octal digit.

ABSOLUTE PERMISSIONS

- Absolute permissions of each category –
  - Read permission – octal 4, binary 100
  - Write permission – octal 2, binary 010
  - Execute permission – octal 1, binary 001
- Add up the numbers for each category and use in **chmod** command.
  - Example: read and write permission is 4+2 = 6, all permissions 4+2+1 = 7, and so on.

ABSOLUTE PERMISSIONS

Binary	Octal	Permissions	Significance
000	0	---	No permissions
001	1	--x	Executable only
010	2	-w-	Writable only
011	3	-wx	Writable and executable
100	4	r--	Readable only
101	5	r-x	Readable and executable
110	6	rw-	Readable and writable
111	7	rwx	Readable, writable and executable (all permissions)

ABSOLUTE PERMISSIONS

- 3 categories and 3 permissions for each category – 3 octal digits can describe a file's permissions completely.
- Most significant digit – user, least significant digit – others.
- **chmod 666 newfile** – read and write permissions to all 3 categories.
- **chmod 644 newfile** – read and write permissions to user, only read permission to group and others (default file permissions in UNIX)
- **chmod 777 newfile** – all permissions to all 3 categories.
- **chmod 000 newfile** – no permissions to all 3 categories (file is useless).

RECURSIVE USAGE OF CHMOD COMMAND

- **chmod** can descend a directory hierarchy and apply the expression to every file and subdirectory within it – recursive option (**-R**)
- **chmod –R a+x shell_scripts** – all files and subdirectories have execute permission.
- Either relative or absolute permission can be used.
- Multiple directories and filenames can also be provided at once.
  - **chmod -R 755 .** (works on hidden files)
  - **chmod -R a+x *** (leaves out hidden files)

DIRECTORY PERMISSIONS

- Directories also have their own permissions.
- Permissions of files are directly influenced by the permissions of the directory that is housing them.
- Possible cases – file cannot be accessed even if it has read permission, file can be changed even if it is write-protected.

DIRECTORY PERMISSIONS

- Default directory permissions – **755 (rwxr-xr-x)**
- Directory must never be writable by group and others.
- If files are being changed even though they may appear to be protected, check the directory permissions.

THE SHELL

- Main interface between user and kernel.
- Looks for special symbols in the command line, performs the tasks associated with them and finally executes the command.
- Unique and multi-faceted –
  - A command interpreter
  - A programming language
  - A process that creates an environment for the user to work in

THE SHELL'S INTERPRETIVE CYCLE

- Issues the prompt and waits for user to enter a command.
- Scans the command line for metacharacters after a command is entered and expands abbreviations to recreate a simplified command line.
- Passes on the command line to the kernel for execution.
- Waits for the command to complete and does not do any work in the meantime.
- Prompt reappears after command is executed and the shell begins its next cycle.

SHELL OFFERINGS

- Bourne family –
  - Bourne shell (**/bin/sh**)
  - Korn shell (**/bin/ksh**)
  - Bash shell (**/bin/bash**)
- C family –
  - C shell (**/bin/csh**)
  - TC shell (**/bin/tcsh**)

WILD CARDS

- Example – listing all files starting with chap
  - **ls chap01 chap02 chap03 chap1 chap2 chap3 chap**
- File names are similar – all occurrences of that similar pattern
  - **ls chap***
- Pattern is framed with ordinary characters (like **chap**) and a metacharacter (like *) using well-defined rules.
- Pattern is then used as an argument to the command, and shell will expand it suitably before the command is executed.
- *Wild cards* – metacharacters that are used to construct the generalized pattern for matching filenames.

WILD CARDS

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 the characters x and z
[!ijk]	A single character that is not an I, j or k (not available in C shell)
[!x-z]	A single character that is not within the ASCII range of the characters x and z (not available in C shell)
{pat1, pat2...}	pat1, pat2, and so on (not available in Bourne shell)

WILD CARDS

- * – matches any number of characters, including none.
  - **ls chap***
  - **echo ***
  - **rm ***
  - **ls *chap***

WILD CARDS

- **?** – matches a single character.
  - **ls chap?**
  - **ls chap??**
  - **rm ?**
  - **ls ?chap?**

WILD CARDS

- Matching the dot – * does not match all files beginning with a **.** or **/**
- **.** (dot) must be matched explicitly if you want to list all hidden filenames in your directory.
  - **ls .???***
- No need to match explicitly if the dot is not at the beginning of the filename.
  - **ls student*txt**

WILD CARDS

- The character class – comprises a set of characters enclosed by **[** and **]** but matches only a single character within the class.
- Example - **[abcd]** is a character class matching a single character – an **a**, **b**, **c** or **d**.
- Can be combined with any string or another wildcard expression.
  - **ls chap[124]**
- Range specification in character class (valid range specification – left side character of the hyphen must have lower ASCII value than right side character of the hyphen).
  - **ls chap[1-4]**

WILD CARDS

- Negating the character class – reverses the matching of character class.
- Example - **[!abcd]** is a character class matching none of **a**, **b**, **c** or **d**.
- Can be used for file extensions –
  - **ls *.[!sh]**
- Unavailable in C shell.

WILD CARDS

- Matching dissimilar patterns – uses **{ }** for housing the patterns and **,** as the delimiter
- Can be used for files –
  - **cp /home/user2/Downloads/*.{c,py,sh} .**
- Can be used for directories –
  - **cp /home/user2/Downloads/{dir1,dir2,dir3}* .**
- Unavailable in Bourne shell.

REMOVING THE MEANING OF METACHARACTERS

- Situation causing nuisance – filenames consisting of metacharacters.
- Dangerous to execute normal pattern matching commands.
- All metacharacters must be protected so that the shell does not interpret them.
- Two solutions –
  - Escaping – providing a \ (backslash) before the wild card.
  - Quoting – enclosing the wild card or entire pattern within quotes.

REMOVING THE MEANING OF METACHARACTERS

- **Escaping** – using \ immediately before a metacharacter to remove its special meaning.
- Example – * means that the asterisk must be matched literally instead of being interpreted as a metacharacter.
- Removing a file named **chap*** – **rm chap***
- Listing a file named **chap[1-3]** – **ls chap\[1-3\]**

REMOVING THE MEANING OF METACHARACTERS

- *Escaping the space* – Shell uses space to delimit command line arguments. Use \ before space.
  - Example: **rm My\ Document.doc**
- *Escaping the* \ itself – Adding another \ before the \
  - Example: **echo **
- *Escaping the newline character* – split the arguments into two lines using \
  - Gives secondary prompt **>** or **?**

REMOVING THE MEANING OF METACHARACTERS

- **Quoting** – enclosing command line arguments in quotes to remove the special meaning.
- Examples: **echo '\'**
  rm "My Document.doc"
- Quoting helpful for large number of command line arguments.
- Single quotes – protect all special characters, except **' '** (single quote)
- Double quotes – protect all, except **" "** (double quote), **$** and **`** (backquote)

REDIRECTION – THE THREE STANDARD FILES

- Shell associates three files with the terminal – two for display and one for keyboard.
- **Standard input** – file (or stream) representing input, which is connected to the keyboard.
- **Standard output** – file (or stream) representing output, which is connected to the display.
- **Standard error** – file (or stream) representing error messages that emanate from the command or shell. Also connected to the display.

REDIRECTION – THE THREE STANDARD FILES

- Three standard files represent three different streams of characters.
- Stream – sequence of bytes.
- Every command that uses streams will always find these files open and available.
- These files are closed when command completes execution.
- Shell can associate and disassociate each of these files with physical devices or disks.

REDIRECTION – THE THREE STANDARD FILES

- **Standard Input** – read from standard input file. Represents three input sources.
  - The keyboard, the default source.
  - A file using redirection with the **<** symbol (metacharacter).
  - Another program using a pipeline.
- **cat** and **wc** without arguments read the file representing standard input.
  End of input – **Ctrl+D**

REDIRECTION – THE THREE STANDARD FILES

- Redirect standard input to originate from a file on disk – **wc < sample.txt**
  - Shell opens disk file *sample.txt* for reading
  - Shell unplugs standard input from its default source and assigns it to *sample.txt*
  - **wc** reads from standard input which has been reassigned by the shell to *sample.txt*
- Taking input both from file and standard input – **cat file1 – file2**
  - (hyphen) symbol is used to indicate sequence of taking input when command is taking input from multiple inputs at once.

REDIRECTION – THE THREE STANDARD FILES

- **Standard Output** – write to standard output file. Three possible destinations –
  - The terminal, default destination.
  - A file using the redirection symbols **>** and **>>**
  - As input to another program using a pipeline.
- Write to a file using **>** symbol – **wc sample.txt > newfile**
- Append to a file using **>>** symbol – **wc sample.txt >> newfile**

REDIRECTION – THE THREE STANDARD FILES

- Redirect standard output to any file on disk – **wc sample.txt > newfile**
  - Shell opens disk file *newfile* for writing
  - Shell unplugs standard output file from its default destination and assigns it to *newfile*
  - **wc** opens the file *sample.txt* for reading.
  - **wc** writes to standard output which has been reassigned by the shell to *newfile*
- Removing the contents of a file without opening/deleting it – **cat newfile > newfile**

REDIRECTION – THE THREE STANDARD FILES

- Each of the three standard files is represented by a number called a *file descriptor*.
- File is opened by referring to its pathname.
  Subsequent read and write operations identify the file by this file descriptor.
- Kernel maintains a table of file descriptors for every process running in the system.
- First three slots – **0** (*standard input*), **1** (*standard output*), **2** (*standard error*)
- Descriptors implicitly prefixed to the redirection symbols – **<** and **<0** are identical, whereas **>** and **>1** are also identical.

REDIRECTION – THE THREE STANDARD FILES

- **Standard Error** – diagnostic error messages on the screen due to incorrect command or opening a non-existent file.
- Default destination – the terminal. Uses **2>** symbol for redirection.
- Redirect standard error – **cat zfile 2> error_output**
- Append standard error – **cat zfile 2>> error_output**

CONNECTING COMMANDS – PIPES AND FILTERS

- Two individual streams of data that can be manipulated by shell – standard input and standard output.
- Connection of these streams of data – pipes and filters.
- Closely connected to redirection.
- One command takes input from another.
- Main approach used to solve text manipulation programs.

CONNECTING COMMANDS – PIPES AND FILTERS

- Example (without pipe):
  ls –l > file_details.txt
  wc < file_details.txt
- Output of **ls –l** redirected to intermediate file **file_details.txt**
- **wc** takes input from **file_details.txt** and displays it.
- Two main disadvantages – process can be slow for long-running commands, necessity of an intermediate file.

CONNECTING COMMANDS – PIPES AND FILTERS

- Example (using pipe): **ls –l | wc**
- Output of **ls -l** passed directly to the input of **wc**
- **ls** is said to be *piped* to **wc**
- | - ***pipe*** operator
- Multiple commands connected this way - *pipeline* is said to be formed.
- Shell sets up this connection and commands have no knowledge of it.

CONNECTING COMMANDS – PIPES AND FILTERS

- Redirection (using pipe): **ls –l | wc > store_file_data.txt**
- Output of **ls -l** passed directly to the input of **wc**
- This is redirected to **store_file_data.txt**
- Multiple pipes can be used on the same line – **ls | wc | wc**

CONNECTING COMMANDS – PIPES AND FILTERS

- All programs run ***simultaneously*** in a pipeline.
- Built-in mechanism to control the flow of the stream.
- Both read and write drivers must work in unison.
- One operates faster than another – appropriate driver must readjust the flow.
  Example: **ls | more**
- Command on the left of | - must use standard output.
  Command on the right of | - must use standard input.

CONNECTING COMMANDS – PIPES AND FILTERS

- **Filters** – set of commands that take input from standard input stream, perform operations on them and write output to standard output stream.
- Examples: **head, tail, more, less, cut, paste, sort, pr, tr, uniq, grep, sed, awk**
- Mostly work on entire lines or fields in files.
- Limited functionality when used in standalone mode – often combined with pipes to perform powerful text manipulation.

REGULAR EXPRESSIONS (REGEX)

- String that can be used to describe several sequences of characters.
- Used by different UNIX commands such as **sed, awk, grep**.
- Shell's wild cards used to match similar filenames with a single expression.
- Regex – match a group of similar patterns.
- The metacharacter set for regex overshadows the wild cards used by the shell.
- Two categories – basic regular expressions (BRE) and extended regular expressions (ERE)

REGULAR EXPRESSIONS (REGEX)

- **grep** – global regular expression print
- Scans its input for a pattern and displays lines containing the pattern, the line numbers or filenames where the pattern occurs.
- Syntax : **grep** *options pattern filenames*
- Searches for *pattern* in one or more *filenames*, or the standard input if no filename is specified.
- Example: display lines containing the string "*sales*" in *emp.lst* – **grep "sales" emp.lst**

REGULAR EXPRESSIONS (REGEX)

- **grep** – can search standard input for pattern and save standard output in a file.
  - Example: **cat emp.lst | grep sales > file_emp.txt**
- Quoting not necessary for single search string but recommended.
- Quoting *essential* for search string with multiple words or containing metacharacters.
  - Example: **grep "jai sharma" emp.lst**

REGULAR EXPRESSIONS (REGEX)

- Multiple filenames with grep – displays the filenames with output.
  - Example: **grep "director" emp1.lst emp2.lst**
- Command failure – returns the prompt silently.
  - Example: **grep president emp.lst**

REGULAR EXPRESSIONS (REGEX)

Option	Significance	Example
-i	Ignores case for matching	**grep -i "agarwal" emp.lst**
-v	Doesn't display lines matching expression	**grep -v "Agarwal" emp.lst**
-n	Displays line numbers along with lines	**grep -n "sales" emp.lst**
-c	Displays count of number of occurrences	**grep -c "director" emp.lst**
-l	Display list of filenames only	**grep -l "manager" *.lst**
-e *exp*	Specifies expression *exp*. Can use multiple times. Also used for matching expression beginning with a hyphen.	**grep -e "Agarwal" -e "aggarwal" -e "agrawal" emp.lst**
-x	Matches pattern with entire line	
-f *file*	Takes patterns from *file*, one per line	
-E	Treats patterns as extended regular expression (ERE)	
-F	Matches multiple fixed strings	

BASIC REGULAR EXPRESSIONS (BRE)

- If an expression uses any of the UNIX metacharacters – regular expression.
- Common query and substitution requirements.
- **grep** supports basic regular expressions (BRE) by ***default*** and extended regular expressions (ERE) with **–E** option.
- **sed** supports only the BRE set.

BASIC REGULAR EXPRESSIONS (BRE)

Expression	Matches
*	Zero or more occurrences of the previous character
g*	Nothing or g, gg, ggg, etc.
.	A single character
.*	Nothing or any number of characters
[pqr]	A single character p, q or r
[c1-c2]	A single character within the ASCII range represented by c1 and c2
[1-3]	A digit between 1 and 3
[^pqr]	A single character which is not p, q or r
[^a-zA-Z]	A non-alphabetic character

BASIC REGULAR EXPRESSIONS (BRE)

Expression	Matches
^pat	Pattern *pat* at the beginning of the line
pat$	Pattern *pat* at the end of the line
bash$	*bash* at end of line
^bash$	*bash* is the only word in line
^$	Lines containing nothing

BASIC REGULAR EXPRESSIONS (BRE)

- The **character class** – specify a group of characters enclosed within a pair of rectangular brackets [ ]
- Match is performed for a single character in the group.
- Example: **grep "[aA]g[ar][ar]wal" emp.lst**

BASIC REGULAR EXPRESSIONS (BRE)

- The * – refers to the immediately preceding character.
- Previous character can occur many times, or not at all.
- Not similar to * used in shell!
- Example: **grep "[aA]gg*[ar][ar]wal" emp.lst**

BASIC REGULAR EXPRESSIONS (BRE)

- The **. (dot)** – matches a single character.
- Similar to **?** used in the shell.
- Powerful when used with * - combined **.*** together signifies any number of characters, including none.
- Example: **grep "j.*saxena" emp.lst**

BASIC REGULAR EXPRESSIONS (BRE)

- The **^ (caret)** – matches at the beginning of the line.
- Example: **grep "^2" emp.lst**
- Reverse search example: **grep "^[^2]" emp.lst**
- The **$** – matches at the end of the line.
- Example: **grep "7...$" emp.lst**
- Necessary when a pattern occurs in more than one place in a line.

BASIC REGULAR EXPRESSIONS (BRE)

- Triple role of the **^ (caret)** –
  - Beginning of a character class – negates every character of the class.
    Example: **[^a-z]**
  - Outside character class, but beginning of the expression – pattern is matched at the beginning of the line.
    Example: **^2...**
  - Any other location – matches itself literally.
    Example: **a^b**

BASIC REGULAR EXPRESSIONS (BRE)

- Metacharacters lose their meaning –
  - **-** (hyphen/minus) – inside character class if it is not enclosed on either side by a suitable character, or when placed outside the class.
  - **.** and * – inside character class.
  - Escaping necessary for matching metacharacters, such as **\[** or **\.***

EXTENDED REGULAR EXPRESSIONS (ERE)

- Possible to match dissimilar patterns with a single expression.
- *Usage*: **grep** with the **-E** option, or **egrep** without any options.
- Two special characters in extended set: **+** and **?**

EXTENDED REGULAR EXPRESSIONS (ERE)

- The **+** (plus) - matches one or more occurrences of the previous character.
- Example: **grep –E "[aA]gg+arwal" emp.lst**
- The **?** (question mark) - matches zero or one occurrence of the previous character.
- Example: **grep –E "[aA]gg?arwal" emp.lst**

EXTENDED REGULAR EXPRESSIONS (ERE)

- Matching multiple patterns using | (and)
  - Example1: **grep –E "sengupta|dasgupta" emp.lst**
  - Example2: **grep –E "(sen|das)gupta" emp.lst**

EXTENDED REGULAR EXPRESSIONS (ERE)

Expression	Matches
ch+	One or more occurrences of character *ch*
ch?	Zero or one occurrence of character *ch*
exp1|exp2	Either exp1 or exp2
(x1|x2)x3	Either x1x3 or x2x3

VARIABLES

- **Variable** – character string to which a value is assigned.
- Value assigned could be a number, text, filename, device.
- *Syntax*: **variable_name=variable_value**
- *Examples*: **a=2, student1_name="abcde"**
- Accessed using the **$** operator.
  Example: **echo $student1_name**

VARIABLES

- **Local (ordinary) variable** – present within the current instance of the shell.
- Not available to programs that are started by the shell.
- Set at the command line and lost when the terminal is shut down.
- Default variable available in the command line.
- *Examples*: **a=2, student1_name="abcde", DOWNLOAD_DIR=/home/abcde/Downloads**

VARIABLES

- **Environment variable** – available in the user's total environment, i.e., the sub-shells that run shell scripts, mail commands, editors.
- Available to any child process of the shell.
- *Examples*: **HOME, PATH, SHELL**
- **set** command – displays all variables available in the current shell.
  env/printenv command - displays only environment variables.

VARIABLES

Env. Variable	Significance
HOME	Home directory – the directory a user is placed on logging in
PATH	List of directories searched by shell to locate a command
LOGNAME or USER	Login name of user
MAIL	Absolute pathname of user's mailbox file
MAILCHECK	Mail checking interval for incoming file
TERM	Type of terminal
PWD	Absolute pathname of current directory (Korn and Bash)
CDPATH	List of directories searched by *cd* when used with a non-absolute pathname
PS1 and PS2	Primary and secondary prompt strings
SHELL	User's login shell and one invoked by programs having shell escapes

.PROFILE

- **.profile** – one of the login scripts which is executed when the user logs in.
- Can have one of three names in Bash - **.profile**, **.bash_profile** or **.bash_login**.
- **ls –a** will display one of these scripts in the *home* directory.
- Contains commands that are meant to be executed only once in a session.
- Allows customization of operating environment to suit user's requirements.
- Changes must be saved and either log out and log in again or execute the script.

READING USER INPUT

- **read** – used to take input from the user and make the script interactive.
- Used with one or more variables.
- Input supplied through the standard input is read into these variables.
- *Sample script* - take two inputs (*pattern* and *filename*) using **read** and use **grep** to search for *pattern* in the given *file*.
- Single read statement can be used with one or more variables – multiple arguments.
- Number of arguments < number of variables – leftover variables unassigned.
  Number of arguments > number of variables – leftovers assigned to last variable.

COMMAND LINE ARGUMENTS

- Shell scripts accept command line arguments and can run non-interactively.
- Arguments specified within a shell script – ***positional parameters***.
- Command itself **$0**, first argument **$1**, second argument **$2**, and so on.
- *Sample script* – take two command line arguments (*pattern* and *filename*) as inputs and search for the pattern using **grep**.

COMMAND LINE ARGUMENTS

Parameter	Significance
$1, $2, ...	Positional parameters representing command line arguments
$#	Number of arguments specified in command line
$0	Name of executed command
$*	Complete set of positional parameters as a single string
"$@"	Each quoted string treated as a separate argument
$?	Exit status of last command
$$	PID of current shell
$!	PID of the last background job

EXIT AND EXIT STATUS

- C programs and shell scripts use same function to terminate a program - **exit**.
- Run with numeric argument – **0** for success, **1** for failure. (**exit 0** and **exit 1**)
- Command failure - **exit** returned a non-zero value.
- **$?** can be used to check exit status of last command.
- *Example1:* **cat emp.lst ; echo $?**
  Example2: **grep "CEO" emp.lst; echo $?**

LOGICAL OPERATORS FOR CONDITIONAL EXECUTION

- Two logical operators used for conditional execution - **&&** and ||
- *Typical syntax:* **cmd1 && cmd2**, **cmd1 || cmd2**
- **&&** - command2 executed only when command1 succeeds.
  - Example: **grep "director" emp.lst && echo "Pattern successfully found in file."**
- || - command2 executed only when command1 fails.
  - Example: **grep "CEO" emp.lst || echo "Pattern not found!"**
- || usually used with **exit** command to terminate scripts when some command fails.

IF CONDITIONAL

- Two-way decisions depending on the fulfillment of a certain condition.
- Form 1

```
if command is successful
then
        execute commands
else
        execute commands
fi
```

Form 2

```
if command is successful
then
        execute commands
fi
```

Form 3

```
if command is succesful
then
        execute commands
elif command is successful
then...
else...
fi
```

- *Sample script* – searching existence of two users in */etc/passwd* using **if**

TEST COMMAND AND ITS SHORTHAND

- **if** cannot be used to directly handle true/false values returned by expressions.
- **test** uses operators to evaluate the condition and returns true/false exit status, which is then used by **if** to make decisions.
- Works in three ways – compares numbers, compares strings, checks file's attributes.
- **test** does not display output directly – necessary to use the parameter **$?**
  - Example: a=3; b=5; c=5.5

```
test $a -eq $b ; echo $?
test $a -lt $b ; echo $?
test $c -gt $a ; echo $?
test -f emp.lst ; echo $?
test ! -w emp.lst ; echo "False that file is not writable!"
```

TEST COMMAND AND ITS SHORTHAND

Operator for Numerical Comparison	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

TEST COMMAND AND ITS SHORTHAND

Operator for String Comparison	Meaning
s1* = *s2	String *s1* is equal to string *s2*
s1* != *s2	String *s1* is not equal to string *s2*
-n *str*	String *str* is not a null string
-z *str*	String *str* is a null string
str	String *str* is assigned and not null
s1* == *s2	String *s1* is equal to string *s2* (Korn and Bash only)

TEST COMMAND AND ITS SHORTHAND

Operator for File Tests	Meaning
-f *file*	*file* exists and is a regular file
-r *file*	*file* exists and is readable
-w *file*	*file* exists and is writable
-x *file*	*file* exists and is executable
-d *file*	*file* exists and is a directory
-s *file*	*file* exists and has a size greater than zero
-e *file*	*file* exists (Korn and Bash only)
-u *file*	*file* exists and has SUID bit set
-k *file*	*file* exists and has sticky bit set
-L *file*	*file* exists and is a symbolic link (Korn and Bash only)

TEST COMMAND AND ITS SHORTHAND

Operator for File Tests	Meaning
f1* -nt *f2	*f1* is newer than *f2* (Korn and Bash only)
f1* -ot *f2	*f1* is older than *f2* (Korn and Bash only)
f1* -ef *f2	*f1* is linked to *f2* (Korn and Bash only)

TEST COMMAND AND ITS SHORTHAND

- Shorthand for **test** – pair of rectangular brackets enclosing the expression.
  test $x -eq $y is the same as **[$x -eq $y]**
- *Sample scripts* -
  Numeric comparison: evaluating shell parameter $#
  String comparison: checking for null string – both interactive and non-interactive
  File tests: various file tests – exists, readable, writable

CASE CONDITIONAL

- Similar to **switch** statement in C.
- Matches an expression for more than one alternative and uses a compact construct to permit multi-way branching.
- *General syntax*:

```
case expression in
        pattern1) commands1 ;;
        pattern2) commands2 ;;
        pattern3) commands3 ;;
        .....
esac
```

- *Sample script* – menu to input choice and display corresponding output.
- Can match multiple patterns, use wild-cards.

WHILE LOOP

- Loops – perform a set of instructions repeatedly.
- Shell offers three types – **while**, **until** and **for** loops.
- Repeat the instruction set until the control command returns a true exit status.
- *General syntax*:

```
while condition is true
do
        commands
done
```

- *Sample script* – accepting ID and description of products in a loop.
- Can be used to wait for a file, set up infinite loops using **sleep**.

FOR LOOP

- Does not test any condition, but uses a list instead.
- *General syntax*:

```
for variable in list
do
        commands
done
```

- *Sample script* – Scan a file repeatedly for each argument passed.
- Multiple sources of the list – variables, command substitution, wild cards, positional parameters.

SET AND SHIFT: MANIPULATING THE POSITIONAL PARAMETERS

- **set –** used to assign arguments to positional parameters.
- Useful when picking up individual fields from the output of a program.
- *Example1 (assigning values)*:

```
set 9876 2345 6213
echo "\$1 is $1, \$2 is $2, \$3 is $3"
echo "The $# arguments are: $*"
```

- *Example2 (command substitution)*:

```
set `date`
echo $*
echo "Today's date: $3 $2 $6"
```

SET AND SHIFT: MANIPULATING THE POSITIONAL PARAMETERS

- **shift –** used to transfer the contents of a positional parameter to its immediate lower numbered one.
- It is done as many times as the statement is called - **$2** becomes **$1**, **$3** becomes **$2**, and so on.
- *Example*: **set `date`**
  echo "$@" ; echo $1 $2 $3
  shift ; echo $1 $2 $3
  shift 2 ; echo $1 $2 $3
- *Sample script* - Example with shift

HERE DOCUMENT (<<)

- Used when data that must be read by the program is fixed and limited.
- Reading data from same file containing the script – **here document** (**<<**) is used.
- Signifies that the data is here itself, rather than in a separate file.
- Any command that uses standard input can also take input from a here document.
- Useful when used with commands that don't accept a filename.
- Contents are interpreted and processed by shell before they are fed as input to a particular command.

HERE DOCUMENT (<<)

- Command substitutions and variables can be used in input via here document, but not possible in normal standard input.
- An interactive script can be run non-interactively using here document.
- *Example*: **bash sample_script_grep.sh << END**
  director
  emp.lst
  END

TRAP STATEMENT – INTERRUPTING A PROGRAM

- Default way of terminating shell scripts – pressing the interrupt key (Ctrl+C).
- Not recommended due to lot of temporary files on disk.
- **trap** - set of things to do if the script receives a signal.
- Normally placed at the beginning of a shell script and uses two lists – command list and signal list.

```
trap 'command_list' signal_list
```

- Signal list – integer values/names of one or more signals.
- Can also be used to ignore signals and continue processing - null command list.


THANK YOU