**MODULE 2**

**5 Mark Questions**

**1. What are the activities performed by the shell in its interpretive cycle.**

* The shell issues the prompt and waits for you to enter a command.
* 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.
* It then passes on the command line to the kernel for execution.
* The shell waits for the command to complete and normally can't do any work while the command is running.
* After command execution is complete, the prompt reappears and the shell returns to its wait in role to start the next cycle. You are now free to enter another command.

**2. Explain wild cards with example.**

****

1. **The \* :** The metacharacters, some filenames beginning with \*, is one of the characters of the shell's wild-card set. It matches any number of characters (including none). It thus matches all filenames specified in the previous command line which can now be shortened in this way:

$ ls chap\*

chap chapz chap01 chap02 chap03 chap04 chap15 chap16 chap17 chapx chapy

**2. The ?:** The next wild-card is the ?, which matches a single character. When used with same string chap (chap?), the shell matches all five-character filenames beginning with chap. Appending another creates the pattern chap??, which matches six-character filenames.

$ ls chap?

chapx chapy chapz

$1s chap??

chap01 chap02 chap03 chap04 chapl5 chap16 chap17

**3. Matching the dot:** The \* doesn't match all files beginning with a . (dot) or the / of a pathname. If you want to list all hidden filenames in your directory having at least three characters after the dot, the dot must be matched explicitly:

$ 1s.???\*

.bash\_profile .exrc .netscape .profile

However, if the filename contains a dot anywhere but at the beginning, it need not be matched explicitly, For example, the expression emp\*lst matches a dot embedded in the filename:

$ls emp\*lst

emp. Lst emp1.1st emp221st emp2.1st empn.lst

**4. The Character Class:** The character class comprises a set of characters enclosed by the rectangular brackets, [and] it matches a single character in the class. The pattern [abcd] is a character class, and it matches, but single character - an a. b, c. or d. This can be combined with any string or another expression, so selecting chap01, chap02 and chap04 now becomes a simple matter.

$ ls chap0[124]

chap0l chap02 chap04

Range specification is also possible inside the class with a - (hyphen); the two characters on either side of it from the range of characters to be matched. Here are two examples:

ls chap0[1-4] Lists chap01, chap02, chap03 and chap04

ls chap[x-z] Lists chapx, chapy and chapz

**Negating the Character Class (!):** You can use the **!** as the first character in the class to negate the class.

\*.[!co] Matches all filenames with a single-character extension but not the .c or .o files.

[!a-z A-Z]\* Matches all filenames that don't begin with an alphabetic character.

**3. Explain the three standard files.**

The files which perform all terminal-related activity with the three files that the shell makes available to every command. These special file are actually stream of characters. A stream is simply a sequence of bytes. When a user logs in, the shell makes available three files representing three streams. Each stream is associated with a default device, and this device is a terminal.

* Standard Input: The file (or stream) representing input, which is connected to the keyboard.
* Standard Output: The file (or stream) representing output, which is connected to the display
* Standard Error: The file (or stream) representing error messages that emanate from the command or shell. This is also connected to the display.

**Standard Input**

We have used the **cat** and **wc** commands to read disk files. These commands have an additional

method of taking input. When they are used without arguments, they read the file representing the standard input. This file is indeed special; it can represent three input sources:

* The keyboard, the default source.
* A file using redirection with the < symbol (a metacharacters).
* Another program using a pipeline.

The shell's manipulative nature finds place here. It can reassign the standard input file to a disk

File. This means it can redirect the standard input to originate from a file on disk. This reassignment or redirection requires the < symbol:

$ wc < sample.txt

3 14 71 //File containing the above 3 lines

**Command:** wc < sample. Txt

1. On seeing the <, the shell opens the disk file, sample.txt, for reading.

2. It unplugs the standard input file from its default source and assigns it to sample.txt.

3. wc reads from standard input which has earlier been reassigned by the shell to sample.txt.

**Standard Output**

All commands displaying output on the terminal actually write to the standard output file as a stream of characters, and not directly to the terminal as such. There are three possible destinations of this stream

* The terminal, the default destination.
* A file using the redirection symbols > and >>,
* As input to another program using a pipeline.

The shell can effect redirection of this stream when it sees the > or >> symbols in the command line. We can replace the default destination (the terminal) with any file by using the > operator, followed by the filename:

**$ wc sample.txt > newfile**

**$ cat newfile**

3 14 71 sample.txt

The first command sends the word count of sample.txt to newfile; nothing appears on the terminal screen. If the output file doesn’t exist, the shell creates it before executing the command. If it exists, the shell overwrites it. The shell also provides the >> symbol to append to a file.

wc sample.txt >> newfile // doesn’t disturb existing contents

**Command**: wc sample.txt > newfile

* On seeing the >, the shell opens the disk file, newfile, for writing.
* It unplugs the standard output file from its default destination and assigns it to newfile.
* wc (and not the shell) opens the file sample.txt for reading.
* wc writes to standard output which has earlier been reassigned by the shell to newfile.

**Standard Error**

The three standard files is represented by a number, called a **file descriptor**. A file is opened by referring to its pathname, but subsequent read and write operations identify the file by this file descriptor. The kernel maintains a table of file descriptors tor every process running in the system. The first three slots are generally allocated to the three standard streams in this manner:

0-Standard input

1-Standard output

2-Standard error

These descriptors are implicitly prefixed to the redirection symbols. For instance, > and >1 mean the same thing to the shell, while < and 0< also are identical. If your program opens a file, in all probability, the file will be allocated the descriptor 3.

We need to explicitly use one of these descriptors when handling the standard error stream. When you enter an incorrect command or try to open a nonexistent file, certain diagnostic message show up on the screen. This is the standard error stream whose default destination is the terminal. Trying to "cat" a nonexistent file produces the error stream:

$ cat foo

cat: cannot open foo

cat fails to open the file and writes to standard error. If you are not using the C shell, you can redirect this stream to a file.

**4. Explain mechanism of process creation in UNIX.**

There are three distinct phases in the creation of a process and uses three important system calls or functions-fork, exec and wait.

The three phases are discussed below:

* **Fork** A process in UNIX is created with the fork system call, which creates a copy of the

process that invokes it. The process image is practically identical to that of the calling process, except for a few parameters like the PID. When a process is forked in this way, the child gets a new PID. The forking mechanism is responsible for the multiplication of processes in the system.

* **Exec** Forking creates a process but it is not enough to run a new program. To do that, the

forked child needs to overwrite its own image with the code and data of the new program.

This mechanism is called exec, and the child process is said to exec a new program. No new process is created here, the PID and PPID of the executed process remain unchanged.

* **Wait** The parent then executes the wait system call to wait for the child process to complete. It picks up the exit status of the child (explained shortly) and then continues with its other functions. A parent may not decide to wait for the death of its child.

All this means that when you run a command (say, cat) from the shell, the shell first forks another shell process. The newly forked shell then overlays itself with the executable image of cat, which then starts to run. The parent (the shell) waits for cat to terminate and then picks up the exit status of the child. This is a number returned by the child to the kernel, and has great significance in both shell programming and systems programming.

When a process is forked, the child has a different PID and PPID from its parent. However, it inherits most of the environment of its parent.

**5. Explain options that can be used with ps command.**

The `ps` command in Unix/Linux is used to display information about processes running on a system. It provides various options that allow you to customize the information displayed. Here are some commonly used options with the `ps` command:

**1. `-e` or `-A`:** Display information about all processes running on the system, including those of other users.

**2. `-u <username>`:** Show processes owned by a specific user. Replace `<username>` with the actual username.

**3. `-p <pid>`:** Display information for a specific process with a given Process ID (PID). Replace `<pid>` with the PID of the process.

**4. `-f`:** Display a full listing that includes more details about each process, such as the parent process (PPID), the terminal associated with the process, the CPU and memory usage, and more.

**5. `-l`:** Provide a long-format listing with additional details, including process state codes, process group ID (PGID), and session ID (SID).

**6. `-t <terminal>`:** Show processes associated with a specific terminal. Replace `<terminal>` with the terminal name or number.

**7. `-g <group>`:** Display processes belonging to a specific process group. Replace `<group>` with the group name or ID.

**8. `-s <session>`:** Show processes associated with a specific session. Replace `<session>` with the session ID.

**9. `-C <command>`:** Display processes that match a specific command name. Replace `<command>` with the name of the command or process you want to filter by.

**10. `-o <format>`:** Customize the output format by specifying which process information fields to display. For example, you can use `-o pid, ppid, cmd` to display only the PID, PPID, and command name.

**11. `-r`:** Show only running processes.

**12. `-H`:** Display a process hierarchy/tree, showing the relationship between parent and child processes.

**13. `-N`:** Show processes that do not belong to the terminal session (useful for system daemons and background processes).

**6. Explain job execution and their priorities**.

In Unix-like operating systems, job execution and priorities are key concepts for managing and scheduling processes. The operating system uses various mechanisms to manage and prioritize the execution of processes and jobs. Here's an overview of how job execution and priorities work in Unix:

**1. \*\*Processes and Jobs\*\*:**

- A process is an instance of a running program. Each process has a unique Process ID (PID).

- A job is a collection of one or more related processes. Often, a job consists of a single process, but it can involve multiple processes working together.

**2. \*\*Process Priority\*\*:**

- Unix systems assign a priority to each process, known as a "nice value." The nice value determines how much CPU time a process gets. A lower nice value indicates higher priority.

- A positive nice value increases the process's priority, allowing it to use less CPU time.

- A negative nice value decreases the process's priority, making it use more CPU time.

- The default nice value is usually 0. Users can use the `nice` command to modify the nice value when launching a process.

**3. \*\*Process Scheduling\*\*:**

- The Unix scheduler is responsible for determining which process or job gets access to the CPU and for how long.

- Processes with higher priority (lower nice values) are scheduled more frequently and get more CPU time.

- The scheduler uses a time-sharing mechanism to allocate CPU time to various processes, ensuring fair access.

**4. \*\*Foreground and Background Jobs\*\*:**

- Jobs in Unix can run in the foreground or background.

- Foreground jobs are executed in the current terminal session and typically interact with the user directly.

- Background jobs do not require user interaction and run independently of the terminal session.

**5. \*\*Job Control\*\*:**

- Unix provides mechanisms for job control, including the `bg`, `fg`, and `jobs` commands.

- `bg`: Puts a suspended or stopped job in the background.

- `fg`: Brings a background job to the foreground.

- `jobs`: Lists the currently running and background jobs in the current shell session.

**6. \*\*Priority Adjustment\*\*:**

- Users and administrators can adjust process priorities using tools like the `nice` command and the `renice` command.

- The `renice` command allows you to change the nice value of an existing process, increasing or decreasing its priority.

**7. \*\*Process States\*\*:**

- Processes can be in various states, including "Running," "Sleeping," "Stopped," and "Zombie." The scheduler and the priority system influence how processes transition between these states.

**8. \*\*Real-time Priority\*\*:**

- Some Unix systems support real-time processes, which have higher priority than regular processes and are designed for time-critical tasks.

**7. What are the main reasons to terminate a process in shell.**

The kill command sends a signal, usually with the intention of killing one or more processes. kil1

is an internal command in most shells; the external /bin/kill is executed only when the shell

lacks the kill capability. The command uses one or more PIDs as its arguments, and by default uses the SIGTERM (15) signal. Thus,

kill 105 It's like using kill -s TERM 105

terminates the job having PID 105. To facilitate premature termination, the & operator displays the PID of the process that's run in the background. If you don't remember the PID, use ps to know that and then use ki11.

If you run more than one job-either in the background or in different windows in the X Window

system, you can kill them all with a single kill statement. Just specify all their PIDs with kil1:

kill 121 122 125 132 138 144

If all these processes have the same parent, you may simply kill the parent in order to kill all

children. However, when you use **nohup** with a set of commands and log out, you can't kill the

parent as **init** acquires their parentage. You then have to kill the processes individually becaust

you can't kill **init.**

***Killing the Last Background Job***

For most shells, the system variable **$!** Stores the PID of the last background job --the same number seen when the **&** is affixed to a command. So you can kill the last background process without using the **ps** command to find out its PID:

For most shells, the system variable $! stores the PID of the last

**$ sort o emp.1st emp.1st &**

345

**$kil1 $!** Kills the sort command

If your shell supports job control (as most shells do), you can use **kill** with a slightly different

syntax to terminate a job.

***Using kill with Other Signals***

By default, **Kill** uses the SIGTERM signal (15) to terminate the process. You would have noticed that some programs simply ignore it and continue execution normally. In that case, the process can be killed with the SIGKILL signal (9). This signal can't be generated at the press of a key, so you must use kill with the signal name (without the SIG) preceded by the -s option:

By default, ki11 uses the SIGTERM signal (15) to terminate the

kill -s KILL 1211 Recommended way of using kill

kill -9 121 Same as above but not recommended

A simple kill command (with TERM) won't kill the login shell. You can kill your login shell by

using any of these commands:

kill -9 $$ $$ stores PID of current shell

kill -s KILL 0 Kills all processes including the login shell

**8. Explain job control.**

A Job is the name given to a group of processes. The easiest way of creating a job is to run a

pipeline of two or more commands. Now consider that you expect a job to complete in 10 minutes and it goes on for half an hour. If you kill the job now, you'll lose a lot of work. If you are using the, C shell, Korn shell or Bash, you can use their job control facilities to manipulate jobs. Job control in these shells means that you can

* Relegate a job to the, background (bg).
* Bring it back to the foreground (fg).
* List the active jobs (jobs).
* Suspend a foreground job (/Ctrl-s/).
* Kill a job (ki11).

The commands needed to perform these activities are shown in parentheses. To begin our discussion if you have invoked a command and the prompt hasn't yet returned, you can suspend the job by pressing [Ctrl-z]. You'll then see the following message:

[1] + Stopped spell uxtip02> uxtip02.spell

Observe that the job has not been terminated yet; it's only suspended ("stopped"). You can now

use the **bg** command to push the current foreground job to the background:

**$ bg**

**[1] spell uxti p02> uxtip02. spell&** A single-process job

The & at the end of the line indicates that the job is now running in the background. So a foreground job goes to the background, first with [Ctrl-z], and then with the **bg** command. You can start more jobs in the background any time:

**$ sort permuted. index > sorted. index &**

[2] 530 [2] indicates second job

**$ wc -1 uxtip?? > word \_count &**

[3] 540

Each of these jobs comprises a single process. Now that you have three jobs running, you can have a listing of their status with the jobs command:

**$ jobs**

[3] +Running wc -l uxtip?? > word\_count &

[1] – Running spell uxtip02 > uxtip02.spell&

[2] Running sort permuted.index > sorted.index &

You can now bring any of the background jobs to the foreground with the **fg** command. To bring

the current (most recent) job to the foreground, use

**fg**

This will bring the **wc** command in the foreground. The **fg** and **bg** commands can also be used

with the job number, job name or a string as arguments, prefixed by the % symbol:

fg %1 Brings first job to foreground

fg %sort Brings sort job to foreground

bg %2 Sends second job to background

bg %?prem Sends to background job containing string perm

A any time, however, you can terminate a job with the **kill** command using the same identifiers as above. Thus, **kill %1** kills the first background job with SIGTERM, while **kill -s KILL %spel1**

At this point, it must be mentioned that the reason why we used [ctrl-z] for suspending a job is that, by default, this is the character set by the **stty** command for this purpose. When you use this command in those shells that support job control, you’ll probably see a line of output similar to this:

start = ^q; Stop = ^s; susp = ^z; dsusp ^y;

The third assignment shows the suspend character as Z, which is stty's way of representing

[ctrl-z]. You can change it if you want but the normally won’t be necessary.

**9. With example explain one time and batch execution commands.**

UNIX provides sophisticated facilities to schedule a job to run at a specified time of day. If the

system load varies greatly throughout the day, it makes sense to schedule less urgent jobs at a time when the system overheads are low. The **at and batch** commands make such scheduling possible.

**at: ONE-TIME EXECUTION**

at takes as its argument the time the job is to be executed and displays the at> prompt. Input has to be supplied from the standard input:

**$ at 14:08**

**at> empawk2.sh**

[Ctrl-d]

commands will be executed using /usr/bin/bash

job 1041188880.a at Sun Dec 29 14:08:00 2002

The job goes to the queue, and at 2:08 p.m. today, the script file empawk2.sh will be executed. at

shows the job number, the date and time of scheduled execution. This job number is derived from the number of seconds that have elapsed.

at doesn't indicate the name of the script to be executed; that is something the user has to remember. The standard output and standard error of the program are mailed to the user, who can use any mail reading program to view it. Alternatively, a user may prefer to redirect the output of the command itself:

**at 15:08**

**empawk2.sh > rep.lst**

You can also use the -f option to take commands from a file. However, any error messages that

may be generated when executing a program will, in the absence of redirection, continue to be mail to the user. To mail job completion to the user, use the -m option.

at also offers the key words now, noon, midnight, today and tomorrow. Moreover, it accepts the + symbols to act as an operator. The words that can be used with this operator hours, days, weeks, months and years. The following forms show the use of some of the key words and operation.

at 5pm 24-hour format assumed

at 3:08pm

at noon At 12:00 hours today

at now+ 1 year At current time after one year

at 3:08pm + 1 day At 3:08 p.m. tomorrow

at 15:08 December 18, 2001

at 9am tomorrow

The month name and day of the week, when used at all, must be either fully spelled out or abbreviated to three letters. Jobs can be listed with the **at -1** command and removed with **at -r**.

**batch: Execute in Batch Queue**

The batch command also schedules jobs for later execution, but unlike at, jobs are executed as

soon as the system load permits. The command doesn't take any arguments but uses an internal

algorithm to determine the execution time. This prevents too many CPU-hungry jobs from running at the same time. The response of **batch** is similar to **at** otherwise:

**$ batch <empawk2.sh**

Commands wil1 be executed using /usr/bin/bash

job 1041185673. b at Sun Dec 29 13:14:33 2002

Any job scheduled with batch goes to a special at queue from where it can be removed with

**at –r.**

**10. Write a shell script to print first 10 numbers (1-10)?**

#!/bin/bash

# Loop through numbers 1 to 10 and print each number

for ((i=1; i<=10; i++)); do

echo $i

done

**11. Discuss head and tail command along with its options.**

**head Command:**

**Purpose:** The head command is used to display the beginning of a file or input data.

**Common Usage:**

Bash

head [OPTIONS] [FILE(s)]

**Common Options:**

-n <NUM> or --lines=<NUM>: Display the first <NUM> lines of the file.

-c <NUM> or --bytes=<NUM>: Display the first <NUM> bytes of the file.

-q or --quiet, --silent: Suppresses the printing of headers when multiple files are specified.

**tail Command:**

**Purpose:** The tail command is used to display the end of a file or input data.

**Common Usage:**

bash

tail [OPTIONS] [FILE(s)]

**Common Options:**

-n <NUM> or --lines=<NUM>: Display the last <NUM> lines of the file.

-c <NUM> or --bytes=<NUM>: Display the last <NUM> bytes of the file.

**12. Explain sort commands with options**

The sort command in Unix is used to sort lines of text files or streams in a specified order. It is a versatile command that can be customized using various options to perform sorting tasks efficiently. Here's an explanation of the sort command along with some common options:

Certainly! Here are some common options for the `sort` command:

1. `-f` or `--ignore-case`: Perform a case-insensitive sort.

2. `-r` or `--reverse`: Sort in reverse (descending) order.

3. `-n` or `--numeric-sort`: Sort numerically.

4. `-u` or `--unique`: Remove duplicate lines from the output.

5. `-t <CHAR>` or `--field-separator=<CHAR>`: Specify a custom field separator.

6. `-k <KEYDEF>` or `--key=<KEYDEF>`: Define a custom sorting key.

7. `-b` or `--ignore-leading-blanks`: Ignore leading whitespace characters.

8. `-c` or `--check`: Check if the input is sorted.

9. `-o <OUTPUT>` or `--output=<OUTPUT>`: Specify the output file.

**13. Explain cut commands with examples.**

The cut command in Unix is used for extracting sections (columns) of lines from a text file or input stream.

It is particularly useful when working with structured data, such as CSV files or tabular data, where you want to extract specific fields or columns. The cut command allows you to specify the delimiter that separates fields and which field(s) you want to extract.

Certainly! Here are the common options for the `cut` command:

1. `-b, --bytes=<list>`: Select bytes from each line.

2. `-c, --characters=<list>`: Select characters from each line.

3. `-d, --delimiter=<delimiter>`: Use `<delimiter>` as the field delimiter character.

4. `-f, --fields=<list>`: Select fields (columns) from each line.

5. `--complement`: Invert the selection to choose non-specified bytes, characters, or fields.

6. `--output-delimiter=<delimiter>`: Use `<delimiter>` to separate output fields.

7. `-s, --only-delimited`: Suppress lines with no field delimiter character.

8. `-z, --zero-terminated`: Treat input as a set of lines separated by null characters.

**14. What are tr commands explain the options and syntax**

The tr command in Unix is used for translating or deleting characters from a text stream. It's a versatile tool for character-level manipulation of text data.

**Syntax:**

tr [OPTIONS] SET1 [SET2]

Certainly! Here are some common options for the `tr` command in Unix:

- `-c`: Complement SET1. Replace characters not in SET1.

- `-d`: Delete characters in SET1 from the input.

- `-s`: Squeeze repeated characters. Replace sequences of identical characters with a single instance.

- `--help`: Display help information about the `tr` command.

- `--version`: Display the version information for the `tr` command.

These options allow you to customize the behavior of the `tr` command when translating or deleting characters in a text stream.