

Q. Explain all Process Scheduling Policies.

Process Scheduling Policies

The act of determining which process in the ready state should be moved to the running state is known as Process Scheduling.

The prime aim of the process scheduling system is to keep the CPU busy all the time and to deliver minimum response time for all programs. For achieving this, the scheduler must apply appropriate rules for swapping processes IN and OUT of CPU.

Schedulers fall into one of the two general categories

- **Non-pre-emptive scheduling.** When the currently executing process gives up the CPU voluntarily.
- **Pre-emptive scheduling.** When the operating system decides to favor another process, pre-empting the currently executing process.

Non-pre-emptive scheduling

(1) First Come First Served (FCFS)

- First-Come-First-Served algorithm is the simplest scheduling algorithm is the simplest scheduling algorithm.
- Processes are dispatched according to their arrival time on the ready queue.
- Being a Non preemptive discipline, once a process has a CPU, it runs to completion.
- The FCFS scheduling is fair in the formal sense or human sense of fairness but it is unfair in the sense that long jobs make short jobs wait and unimportant jobs make important jobs wait.
- One of the major drawbacks of this scheme is that the average time is often quite long.

Process	P1	P2	P3	P4	P5	Total
Process Time	5	6	7	2	3	23
Waiting Time	0	5	11	18	20	54
Turn Around Time	5	11	18	20	23	77

Average waiting time = $54/5 = 10.8\text{ms}$

Average Turnaround time = $77/5 = 15.4\text{ms}$

(2) Shortest Job First (SJF) / Shortest Process Next (SPN)

- Shortest Job First other name of this algorithm is **Shortest Process Next**.
- Shortest-Job-First is a non-preemptive discipline in which waiting job (or process) with the smallest estimated time runs next.
- The SJF scheduling is especially appropriate for batch jobs for which the run times are known in advance.
- As SJF scheduling algorithm gives the minimum average time for a given set of processes, it is probably optimal.
- The SJF algorithm favors short jobs (or processors) at the expense of longer ones.
- The obvious problem with SJF scheme is that it requires precise knowledge of how long a job or process will run, and this information is not usually available.
- The best SJF algorithm can do is to rely on user estimates of run times.
- Like FCFS, SJF is non preemptive therefore, it is not useful in timesharing environment in which reasonable response time must be guaranteed.

Process	P1	P2	P3	P4	P5	Total
Process Time	5	6	7	2	3	23
Waiting Time	5	10	16	0	2	35
Turn Around Time	10	16	23	2	5	56
Process Sequence	P4	P5	P1	P2	P3	

Average waiting time = $35/5 = 7\text{ms}$

Average Turnaround time = $56/5 = 11.2\text{ms}$

Pre-emptive scheduling

(1) Shortest Remaining Time Next

- This is a preemptive version of the previous one, in which the scheduler always dispatches that ready process which has the shortest expected remaining time to completion.
- The dispatching decision is always made when a new process is submitted: the currently running process has obviously a smaller remaining time than all other ready ones; otherwise, it wouldn't have been dispatched.
- But a newly submitted job may have an even shorter one, in which case the currently running one would be blocked and put in ready state. This decreases the average turnaround time with respect to shortest process next (SPN).
- Initially at 0ms arrival time, we have only one process p1. We execute this process for 1ms, now remaining time for p1=8ms. At 1ms we have two processes p1 and p2. The burst time for p1=8ms and p2=4. Minimum is 4ms. So p2 will be executed for 1ms as the next process arrives after 1ms
- Now remaining time for p2=3ms. At 2ms we have 3 processes p1=8ms, p2=3ms and p3=5ms. Minimum=3ms, p2 will be executed for 1ms.
- At 3ms, there will be four processes p1=8ms, p2=3ms and p3=5ms. Minimum is 3ms, so p2 will be executed for 1ms

Process	P1	P2	P3	P4	P5	Total
Process Time (ms)	9	4	5	7	3	28
Arrival Time (ms)	0	1	2	3	4	

ms	p1	p2	p3	p4	p5
0	9	x	x	x	x
1	8	4	x	x	x
2	8	3	5	x	x
3	8	2	5	7	x
4	8	1	5	7	3
5	8	0	5	7	3
8	8	x	5	7	0
13	8	x	x	7	x
20	8	x	x	0	x
28	0	x	x	x	x

- At 4ms, p2=1 which is minimum, it will be executed and it will be finished. At 5ms there are 4 process p1=8ms, p3=5ms, p4=7ms and p5=3ms.

Minimum is p5=3ms. It will be executed for next 3ms. At 8ms p3=5ms is minimum it will be executed for next 5ms. At 13ms p4=7ms is minimum so it will be executed for next 7ms. At 20ms, p1 process will be executed for next 8ms. Waiting time = Total waiting time – Number of milliseconds the process has already executed – arrival time

	Total waiting time	Already executed	Arrival time	Waiting time
P1	20	1	0	19ms
P2	4	3	1	0ms
P3	8	0	2	6ms
P4	13	0	3	10ms
P5	5	0	4	1ms
Total waiting time = 36ms				
Average waiting time = $36/5 = 7.2\text{ms}$				
Turnaround time = Total Turnaround time – arrival time				

	Total turnaround time	Arrival time	Waiting time	Total turnaround time = 64ms Average turnaround time = $64/5 = 12.8\text{ms}$
P1	28	0	28ms	
P2	5	1	4ms	
P3	13	2	11ms	
P4	20	3	17ms	
P5	8	4	4ms	

(2) Round Robin

- Round-robin (RR) is one of the simplest scheduling algorithms for processes in an operating system.
- As the term is generally used, time slices are assigned to each process in equal portions and in circular order, handling all processes without priority (also known as cyclic executive).
- Round-robin scheduling is simple, easy to implement, and starvation-free. Round-robin scheduling can also be applied to other scheduling problems, such as data packet scheduling in computer networks.
- The name of the algorithm comes from the round-robin principle known from other fields, where each person takes an equal share of something in turn.

Process	P1	P2	P3	P4	P5	Total
Process Time (ms)	8	6	2	7	9	32

- Suppose Time quantum (Time interval) is of 4ms. So each will be executed for 4ms in FCFS order.

ms	p1	p2	p3	p4	p5
0	8	6	2	7	9
4	4	6	2	7	9
8	4	2	2	7	9
10	4	2	0	7	9
14	4	2	x	3	9
18	4	2	x	3	5
22	0	2	x	3	5
24	x	0	x	3	5
27	x	x	x	0	5
31	x	x	x	x	1
32	x	x	x	x	0

Priority Policy (Preemptive and Non-Preemptive)

- Priority scheduling is a more general case of SJF, in which each job is assigned a priority and the job with the highest priority gets scheduled first. (SJF uses the inverse of the next expected burst time as its priority - The smaller the expected burst, the higher the priority.)
- Note that in practice, priorities are implemented using integers within a fixed range, but there is no agreed-upon convention as to whether "high" priorities use large numbers or small numbers. This book uses low number for high priorities, with 0 being the highest possible priority.
- For example, the following Gantt chart is based upon this process burst times and priorities, and yields an average waiting time of 8.2 ms:

Process	P1	P2	P3	P4	P5	Total
Process Time	10	1	2	1	5	19
Waiting Time	6	0	16	18	1	41
Turn Around Time	16	1	18	19	6	60
Priority	3	1	4	5	2	
Process Sequence	P3	P1	P4	P5	P2	



- Priorities can be assigned either internally or externally. Internal priorities are assigned by the OS using criteria such as average burst time, ratio of CPU to I/O activity, system resource use, and other factors available to the kernel. External priorities are assigned by users, based on the importance of the job, fees paid, politics, etc.
- Priority scheduling can be either preemptive or non-preemptive.
- Priority scheduling can suffer from a major problem known as indefinite blocking, or starvation, in which a low-priority task can wait forever because there are always some other jobs around that have higher priority.

- If this problem is allowed to occur, then processes will either run eventually when the system load lightens (at say 2:00 a.m.), or will eventually get lost when the system is shut down or crashes. (There are rumors of jobs that have been stuck for years.)
- One common solution to this problem is aging, in which priorities of jobs increase the longer they wait. Under this scheme a low-priority job will eventually get its priority raised high enough that it gets run.

Q . Explain virtual memory location with paging.

Virtual Memory

A computer can address more memory than the amount physically installed on the system. This extra memory is actually called **virtual memory** and it is a section of a hard disk that's set up to emulate the computer's RAM.

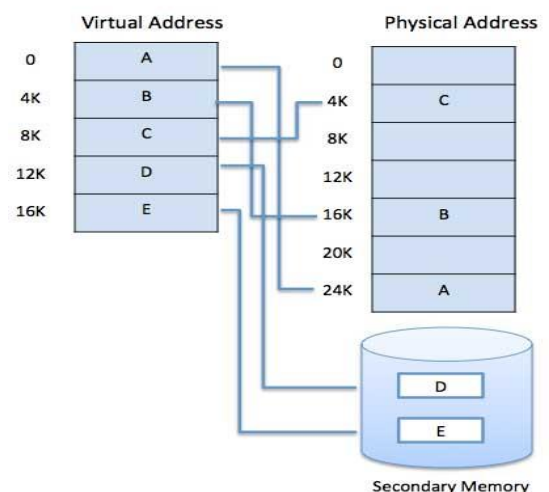
The main visible advantage of this scheme is that programs can be larger than physical memory. Virtual memory serves two purposes. First, it allows us to extend the use of physical memory by using disk. Second, it allows us to have memory protection, because each virtual address is translated to a physical address.

Following are the situations, when entire program is not required to be loaded fully in main memory.

- User written error handling routines are used only when an error occurred in the data or computation.
- Certain options and features of a program may be used rarely.
- Many tables are assigned a fixed amount of address space even though only a small amount of the table is actually used.
- The ability to execute a program that is only partially in memory would counter many benefits.
- Less number of I/O would be needed to load or swap each user program into memory.
- A program would no longer be constrained by the amount of physical memory that is available.
- Each user program could take less physical memory, more programs could be run the same time, with a corresponding increase in CPU utilization and throughput.

Modern microprocessors intended for general-purpose use, a memory management unit, or MMU, is built into the hardware. The MMU's job is to translate virtual addresses into physical addresses.

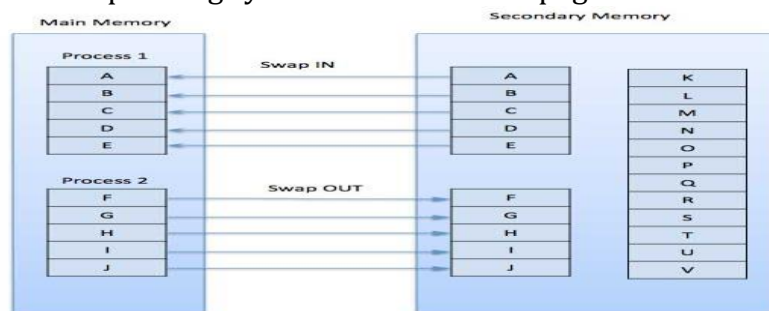
Virtual memory is commonly implemented by demand paging. It can also be implemented in a segmentation system. Demand segmentation can also be used to provide virtual memory.



Virtual Memory with Paging

A paging system is same as swapping where processes reside in secondary memory and pages are loaded only on demand, not in advance. When a context switch occurs, the operating system does not copy any of the old program's pages out to the disk or any of the new program's pages into the main memory. Instead, it just begins executing the new program after loading the first page and fetches that program's pages as they are referenced.

While executing a program, if the program references a page which is not available in the main memory because it was swapped out a little ago, the processor treats this invalid memory reference as a page fault and transfers control from the program to the operating system to demand the page back into the memory.



Advantages

- Large virtual memory.
- More efficient use of memory.
- There is no limit on degree of multiprogramming.

Disadvantages

- Number of tables and the amount of processor overhead for handling page interrupts are greater than in the case of the simple paged management techniques.

Q. Explain Unix File Type in Details?

By default UNIX have only 3 types of files. They are

1. Regular files
2. Directory files
3. Special Files (This category is having 5 sub types in it.)

So in practical we have total 7 types (1+1+5) of files in Linux/Unix. And in Solaris we have 8 types. And you can see the file type indication at leftmost part of "ls -l" command. Here are those files type.

1. Regular File (-)
2. Directory Files(d)
3. Special or Device Files
 - a) Block file (b)
 - b) Character device file (c)
 - c) Named pipe file or just a pipe file (p)
 - d) Symbolic link file (l)
 - e) Socket file (s)

For your information there is one more file type called door File (D) which is present in Sun Solaris as mention earlier. A door is a special file for inter-process

communication between a client and server (so total 8 types in UNIX machines). We will learn about different types of files as below sequence for every file type.

Ordinary or Regular Files

A large majority of the files found on UNIX and Linux systems are ordinary files. Ordinary files contain ASCII (human-readable) text, executable program, binaries program data, and image file, Compress File and more.

Directories

A directory is a binary file used to track and locate other files and directories. The binary format is used so that directories containing large numbers of filenames can be search quickly.

Device (Special)/ Block Files

Device or special files are used for device I/O on UNIX and Linux systems. They appear in a file system just like an ordinary file or a directory.

On UNIX systems there are two flavors of special files for each device, character special files and block special files. Linux systems only provide one special file for each device.

When a character special file is used for device I/O, data is transferred one character at a time. This type of access is called raw device access.

When a block special file is used for device I/O, data is transferred in large fixed-size blocks. This type of access is called block device access.

Links

A link is a tool used for having multiple filenames that reference a single file on a physical disk. They appear in a file system just like an ordinary file or a directory.

Like special files, links also come in two different flavors. There are hard links and symbolic links.

Hard links do not actually link to the original file. Instead they maintain their own copy of the original file's attributes (i.e. location on disk, file access permissions, etc.). If the original file is deleted, its data can still be accessed using the hard link.

On the other hand, symbolic links contain a pointer, or pathname, to the original file. If the original file is deleted, its data can no longer be accessed using the symbolic link, and the link is then considered to be a stale link.

Named Pipes

Named pipes are tools that allow two or more system processes to communicate with each other using a file that acts as a pipe between them. This type of communication is known as inter process communication or IPC for short.

Sockets

Sockets are also tools used for inter process communication. The difference between sockets and pipes is that sockets will facilitate communication between processes running on different systems, or over the network.

With so many different types of files, it's often wise to identify a file's type before performing any operation with it. The **ls -l** command and the **file** command are useful for determining file types.

Consider the long listing of the **livefirelabs1** file:

-rw-rw-r-- 1 student1 student1 0 Jun 27 18:55 livefirelabs1

The first character of the first field indicates the file type. In this example, the first character is a - (hyphen) indicating that **livefirelabs1** is an ordinary or regular file. Consider the long listing of the **live1** file:

rw-rw-rw- 1 student1 student1 13 Jun 27 17:57 live1 -> livefirelabs1

The first character of the first field is the letter **l** indicating **live1** is a symbolic link. The following is a table listing what characters represent what types of files:

Ordinary or Regular File

d - Directory

c - Character special file

b - Block special file

l - Symbolic link

p - Named pipe

s - Socket

The **file** command is also helpful for determining file types. The syntax for this command is: **\$ file filename.**

File Type	First Character in File Listing	Description
Regular file	-	Normal files such as text, data, or executable files
Directory	d	Files that are lists of other files
Link	l	A shortcut that points to the location of the actual file
Special file	c	Mechanism used for input and output, such as files in /dev
Socket	s	A special file that provides inter-process networking protected by the file system's access control
Pipe	P	A special file that allows processes to communicate with each other without using network socket semantics

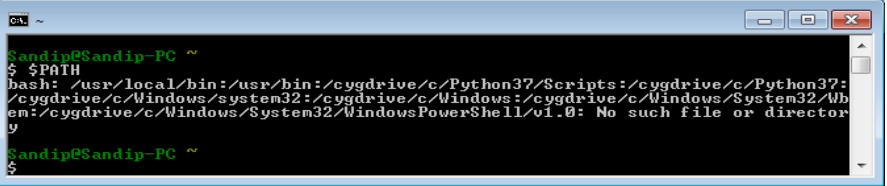
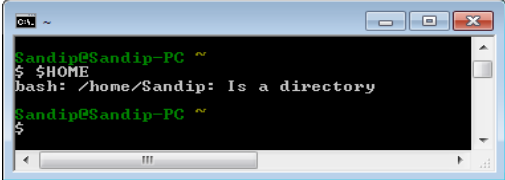
Q Explain System Variables and User Variables.

Environment variables are dynamic values which affect the processes or programs on a computer. They exist in every operating system, but types may vary. Environment variables can be created, edited, saved, and deleted and give information about the system behavior.

Environment variables can change the way a software/programs behave E.g. \$LANG environment variable stores the value of the language that the user understands. This value is read by an application such that a Chinese user is shown a Mandarin interface while an American user is shown an English interface.

Let's study some common environment variables

Variable	Description
PS2	<p>PS2 (Prompt String 2) is one of the prompts available in Linux/Unix. The other prompts are PS1, PS3 and PS4. This is very much useful for entering a large command in multiple lines and when you execute incomplete command, this prompt will come into picture.</p> <p>Check what your default PS2 prompt by executing below command: echo \$PS2 Output: ></p> <p>If you see the prompt changed from '>' to '-->'. This will be very handy and more informative when dealing with a command which spreads on multiple lines.</p>

	PS2="--->" Set the above PS2 prompt and check it yourself with following data echo \$PS2 Output: -->
PATH	 <p>This variable contains a colon (:)-separated list of directories in which your system looks for executable files.</p> <p>When you enter a command on terminal, the shell looks for the command in different directories mentioned in the \$PATH variable. If the command is found, it executes. Otherwise, it returns with an error 'command not found'.</p>
HOME	Indicates the home directory of the current user: the default argument for the cd built-in command. 
LOGNAME/ USER	Contains your username. It's set automatically when you log in.
TERM	Default terminal emulator
SHELL	Shell being used by the user
IFS	The Internal Field Separator to separate input on the command line. By default, this is a space.
MAIL	The path to the current user's mailbox.
MAILCHECK	The MAILCHECK environment variable contains the minimum number of seconds that must elapse before dbx checks for incoming mail. The check is performed just before printing the prompt, if \$MAILCHECK seconds have elapsed since the last check. If \$MAILCHECK is null or is zero, the check is performed before each prompt. If \$MAILCHECK is unset, or is set to a non-numeric value, checking for mail is disabled. The default is 600 seconds

Accessing Variable values

In order to determine value of a variable, use the command Variables are Case Sensitive. Make sure that you type the variable name in the right letter case otherwise you may not get the desired results.

The 'env' command displays all the environment variables.

```
home@VirtualBox:~$ echo $USER
home
home@VirtualBox:~$ echo $HOME
/home/home
home@VirtualBox:~$ echo $UID
1000
home@VirtualBox:~$ echo $TERM
xterm
home@VirtualBox:~$ echo $PATH
/usr/lib/lightdm/lightdm:/usr/local/sbin:/usr/local/bin:/bin:/usr/games
home@VirtualBox:~$
```

```
guru99@VirtualBox:~$ env
SSH_AGENT_PID=8193
DBUS_STARTER_ADDRESS=unix:abstract=/tmp/dbus-3cJBjUA4b1018600004deb
GPG_AGENT_INFO=/tmp/keyring-lp9SZ1/gpg:0:1
TERM=xterm
SHELL=/bin/bash
XDG_SESSION_COOKIE=a5fb982bea8a8cb0299c2c9f00000007-134
WINDOWID=58720261
```

USER VARIABLE

1. SET

The set command assigns a value to a variable (**or multiple values to multiple variables**). Without any options, all set variables are shown.

If a value has spaces in it, it should be contained in quotes. If a list of values is desired, parentheses () should be used for the assignment, while individual access is done via square brackets [].

A single **set** command can be used to set many variables, but such a use is not recommended.

Examples:

```
set value = 7
echo $value
set new_val = "some number, like seven"
echo $new_val
```

2. UNSET

The unset command to remove a variable from your shell environment.

Examples:

```
set value = 7
echo $value
unset value
echo $value
```

3. ECHO

The echo command is used to displays the given text/message on the screen/terminal window. This is often used to show the user what is happening or to prompt for a response.

Examples:

```
echo "BCA SEM 4 Semester"
```

Q. Explain GRUB Configuration

The configuration file (/boot/grub/grub.conf), which is used to create the list of operating systems to boot in GRUB's menu interface, essentially allows the user to select a pre-set group of commands to execute. The commands given in GRUB Commands can be used, as well as some special commands that are only available in the configuration file.

Configuration File Structure

The GRUB menu interface configuration file is /boot/grub/grub.conf. The commands to set the global preferences for the menu interface are placed at the top of the file, followed by stanzas for each operating kernel or operating system listed in the menu.

The following is a very basic GRUB menu configuration file designed to boot either Red Hat Enterprise Linux:

```

default=0
timeout=10
splashimage=(hd0,0)/grub/splash.xpm.gz

# section to load Linux
title Red Hat Enterprise Linux (2.4.21-1.ent)
    root (hd0,0)
    kernel /vmlinuz-2.4.21-1 ro root=/dev/sda2
    initrd /initrd-2.4.21-1.img

# section to load Windows
title Windows
    rootnoverify (hd0,0)
    chainloader +1

```

This file configures GRUB to build a menu with Red Hat Enterprise Linux as the default operating system and sets it to auto-boot after 10 seconds. Two sections are given, one for each operating system entry, with commands specific to the system disk partition table.

Configuring a GRUB menu configuration file to boot multiple operating systems is beyond the scope of this chapter. Consult Additional Resources for a list of additional resources.

Configuration File Directives

The following are directives commonly used in the GRUB menu configuration file:

- **chainloader** *</path/to/file>* — Loads the specified file as a chain loader. Replace *</path/to/file>* with the absolute path to the chain loader. If the file is located on the first sector of the specified partition, use the blocklist notation, +1.
- **color** *<normal-color>* *<selected-color>* — Allows specific colors to be used in the menu, where two colors are configured as the foreground and background. Use simple color names such as red/black.

For example: **color red/black green/blue**

- **default**=*<integer>* — Replace *<integer>* with the default entry title number to be loaded if the menu interface times out.
- **fallback**=*<integer>* — Replace *<integer>* with the entry title number to try if the first attempt fails.
- **hiddenmenu** — Prevents the GRUB menu interface from being displayed, loading the default entry when the timeout period expires. The user can see the standard GRUB menu by pressing the [Esc] key.
- **initrd** *</path/to/initrd>* — Enables users to specify an initial RAM disk to use when booting. Replace *</path/to/initrd>* with the absolute path to the initial RAM disk.
- **kernel** *</path/to/kernel>* *<option-1>* *<option-N>* — Specifies the kernel file to load when booting the operating system. Replace *</path/to/kernel>* with an absolute path from the partition specified by the root directive. Multiple options can be passed to the kernel when it is loaded.
- **password**=*<password>* — Prevents a user who does not know the password from editing the entries for this menu option.

Optionally, it is possible to specify an alternate menu configuration file after the `password=<password>` directive. In this case, GRUB restarts the second stage boot loader and uses the specified alternate configuration file to build the menu. If an alternate menu configuration file is left out of the command, a user who knows the password is allowed to edit the current configuration file.

- **root (<device-type><device-number>,<partition>)** — Configures the root partition for GRUB, such as (hd0,0), and mounts the partition.
- **rootnoverify (<device-type><device-number>,<partition>)** — Configures the root partition for GRUB, just like the root command, but does not mount the partition.
- **timeout=<integer>** — Specifies the interval, in seconds, that GRUB waits before loading the entry designated in the default command.
- **splashimage=<path-to-image>** — Specifies the location of the splash screen image to be used when GRUB boots.
- **title group-title** — Specifies a title to be used with a particular group of commands used to load a kernel or operating system.