

Operating System (315319)

Model Question Paper — Set C · Model Answers

Q.1

a) Four functions of operating system.

2 M

1. **Process management** — creation, deletion, scheduling and synchronisation of processes.
2. **Memory management** — allocation and de-allocation of main memory, keeping track of used memory.
3. **File management** — creating, deleting and organising files and directories.
4. **Device (I/O) management** — controlling I/O devices through drivers.
5. **Security and protection**, error detection, accounting, user interface.

(Any four, ½ M each)

b) System call with examples.

2 M

System call: A system call is the programming interface through which a user program requests a **service from the operating system kernel** (such as I/O, file access or process creation). It switches the CPU from user mode to kernel mode.

Examples: fork(), exit(), wait(), open(), read(), write(), close(), getpid().

c) Function of long-term scheduler.

2 M

The **long-term scheduler (job scheduler)** selects processes from the **job pool** (on the disk) and loads them into main memory, i.e. into the **ready queue** for execution.

- It controls the **degree of multiprogramming** (number of processes in memory).
- It selects a good mix of I/O-bound and CPU-bound processes.

d) Virtual memory and advantage.

2 M

Virtual memory is a technique that allows the execution of a process that is not completely in main memory; it gives the user an illusion of a very large memory by using part of the disk as an extension of RAM (pages are brought in on demand).

Advantage: programs larger than physical memory can be executed. (Also: more processes in memory, higher CPU utilisation.)

e) Program vs process.

2 M

Program	Process
A program is a passive entity — a set of instructions stored in a file on disk.	A process is an active entity — a program in execution.
Has no program counter or state.	Has a program counter, registers, stack, state and PCB.

Exists until it is deleted (long lifetime).	Exists only from creation to termination (limited lifetime).
Needs only storage space on disk.	Needs CPU, memory, I/O and other resources.

(Any two, 1 M each)

f) File types with extensions.

2 M

File type	Usual extension	Function
Executable	exe, com, bin, or none	ready-to-run machine-language program
Object	obj, o	compiled machine language, not linked
Source code	c, cpp, java, py	source code in various languages
Text	txt, doc	textual data, documents
Archive	zip, tar, rar	related files grouped into one, compressed
Multimedia	mp3, mp4, avi	audio / video information

(Any four, ½ M each)

g) Logical and physical address.

2 M

i) Logical address: An address generated by the **CPU** while a program is running; it is also called a virtual address. The set of all logical addresses is the logical address space.

ii) Physical address: The actual address of a location in **main memory**, as seen by the memory unit (loaded into the memory-address register). The memory-management unit (MMU) maps logical addresses to physical addresses at run time.

Q.2

a) Time-sharing OS; difference from multiprogrammed OS.

4 M

A **time-sharing (multitasking) OS** is a logical extension of multiprogramming in which the CPU switches among several jobs **so frequently** that users can interact with each program while it is running. Each user is given a small **time slice** (quantum) of CPU time in turn (Round Robin), so every user feels that he has the whole computer. It needs interactive I/O (keyboard, display), CPU scheduling, memory management, virtual memory and a file system. Examples: UNIX, Linux, Multics.

Multiprogrammed OS	Time-sharing OS
Several jobs are kept in memory; CPU switches to another job only when the current job waits (e.g. for I/O).	CPU switches among jobs after every time slice, even if the job is not waiting.
Main aim: maximise CPU utilisation.	Main aim: minimise response time for interactive users.
No (or little) user interaction while the job runs.	Direct interaction of many users with the system.

Example: early batch multiprogramming systems.	Example: UNIX, Linux multi-user terminals.
--	--

(Explanation 2 M, differences 2 M)

b) OS tools.

4 M

i) User management: used to create, modify and delete user accounts and groups, set passwords, and assign permissions/rights to users (e.g. Windows *Computer Management* → *Local Users and Groups*, *netplwiz*; Linux *useradd*, *passwd*, *usermod*). It controls who can log in and what each user can access.

ii) Task scheduler: used to schedule programs or scripts to run **automatically** at a given time or on an event (at log-on, at start-up, daily/weekly), e.g. automatic backup or disk cleanup. Windows: *Task Scheduler* (*taskschd.msc*); Linux: *cron* / *at*.

iii) Performance monitor: used to watch the real-time and logged performance of the system — CPU usage, memory usage, disk and network activity — using counters and graphs, to find bottlenecks. Windows: *Performance Monitor* (*perfmon*), Task Manager; Linux: *top*, *vmstat*.

iv) Device management: used to view and control the hardware devices attached to the computer — install, update, roll back, enable/disable or uninstall **device drivers** and check device status/conflicts. Windows: *Device Manager* (*devmgmt.msc*); Linux: *lsusb*, *lspci*.

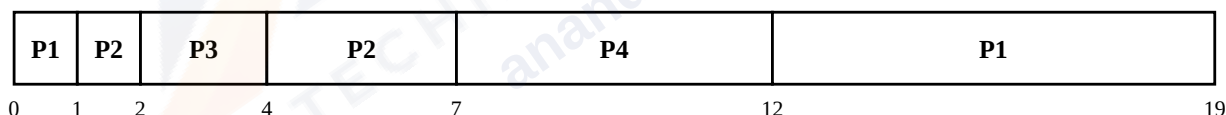
(Each tool 1 M)

c) SRTN scheduling with example.

4 M

SRTN (Shortest Remaining Time Next) is the **preemptive version of SJF**. Whenever a new process arrives in the ready queue, its burst time is compared with the **remaining time** of the running process. If the new process needs less time, the running process is preempted and the CPU is given to the new process. It gives the minimum average waiting time, but long processes may starve and remaining times must be known.

SRTN (Shortest Remaining Time Next, preemptive SJF)



Gantt chart

Process	AT	BT	Completion time (CT)	TAT = CT - AT	WT = TAT - BT
P1	0	8	19	19 - 0 = 19	19 - 8 = 11
P2	1	4	7	7 - 1 = 6	6 - 4 = 2
P3	2	2	4	4 - 2 = 2	2 - 2 = 0
P4	3	5	12	12 - 3 = 9	9 - 5 = 4

Average waiting time = $17 / 4 = 4.25$ ms

At $t = 1$, $P2(4) < \text{remaining of } P1(7) \rightarrow P1$ preempted; at $t = 2$, $P3(2) < \text{remaining of } P2(3) \rightarrow P2$ preempted.

(Explanation 2 M, Gantt chart and average WT 2 M)

d) Tree-structured directory.

4 M

A **tree-structured directory** is a generalisation of the two-level directory to a tree of **arbitrary height**. There is one **root directory**; every directory can contain files and **sub-directories**. Every file has a unique path name. Each process has a **current (working) directory**. Path names can be **absolute** (from the root, e.g. /home/ravi/prog/a.c) or **relative** (from the current directory, e.g. prog/a.c). One bit in each directory entry tells whether the entry is a file (0) or a sub-directory (1).

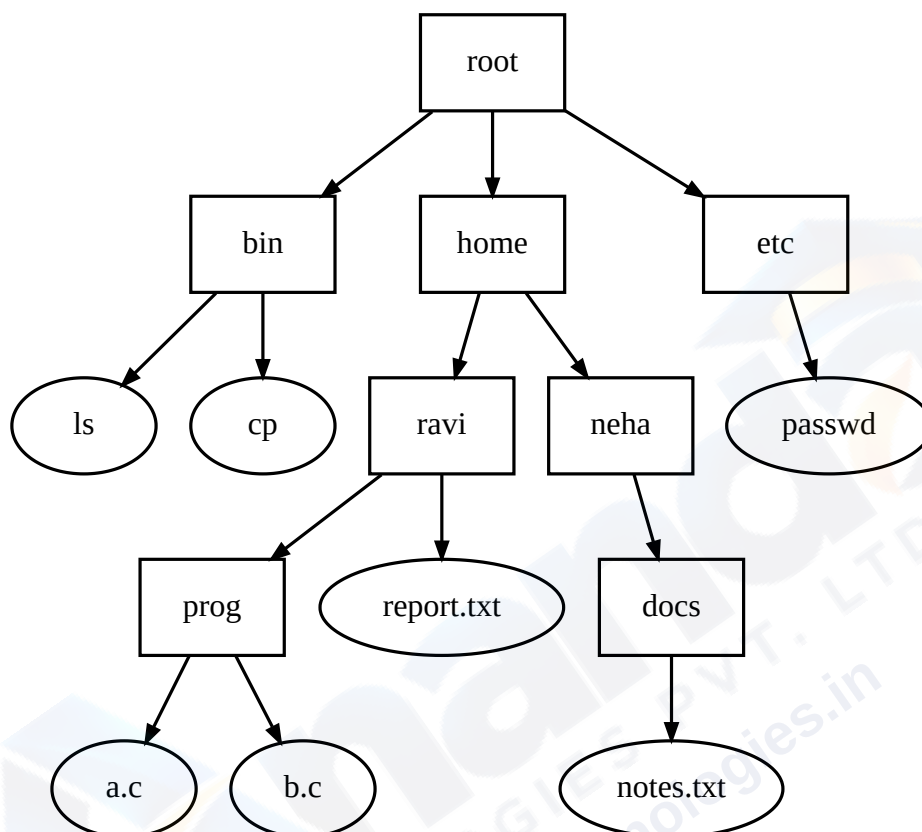


Fig.: Tree-structured directory (box = directory, ellipse = file)

Advantages: 1) Users can **group** related files into sub-directories. 2) Searching is efficient (path name). 3) Same file names are allowed in different directories. 4) Scalable and easy to manage; protection can be set per directory. (Disadvantage: files cannot be shared directly — acyclic graph is needed for that.)

(Explanation 2 M, diagram 1 M, advantages 1 M)

Q.3

a) Process management and secondary-storage management.

4 M

i) **Process management** — the OS is responsible for:

- Creating and deleting both user and system processes.
- Suspending and resuming processes.
- Providing mechanisms for **process synchronisation**.
- Providing mechanisms for **process communication** (IPC).
- Providing mechanisms for **deadlock handling**; scheduling processes on the CPU.

ii) **Secondary-storage management** — main memory is volatile and small, so disks are used as the main on-line storage. The OS is responsible for:

- **Free-space management** (bitmap, linked list).
- **Storage allocation** — allocating disk blocks to files.
- **Disk scheduling** — ordering disk I/O requests efficiently.

(Each component 2 M)

b) Many-to-one model.

4 M

In the **many-to-one model**, many user-level threads are mapped to a single kernel thread.

Thread management is done by the thread library in user space, so it is efficient (fast creation and switching).

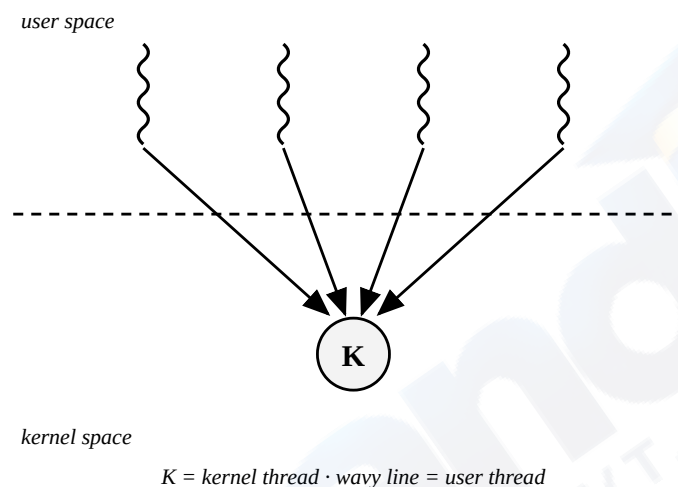


Fig.: Many-to-one model

Drawbacks:

- If one thread makes a **blocking system call**, the **entire process is blocked**.
- Only one thread can access the kernel at a time, so multiple threads **cannot run in parallel** on multicore / multiprocessor systems.
- Very few systems use it now.

Example: Solaris Green Threads, GNU Portable Threads.

(Explanation 1 M, diagram 2 M, drawbacks 1 M)

c) Segmentation with hardware diagram.

4 M

Segmentation is a memory-management scheme that supports the **user's view of memory**. A program is a collection of variable-size logical units called **segments** — main program, functions, stack, symbol table, arrays. A logical address is a pair **<segment number s, offset d>**. Each process has a **segment table**; each entry has a **base** (starting physical address of the segment) and a **limit** (length of the segment).

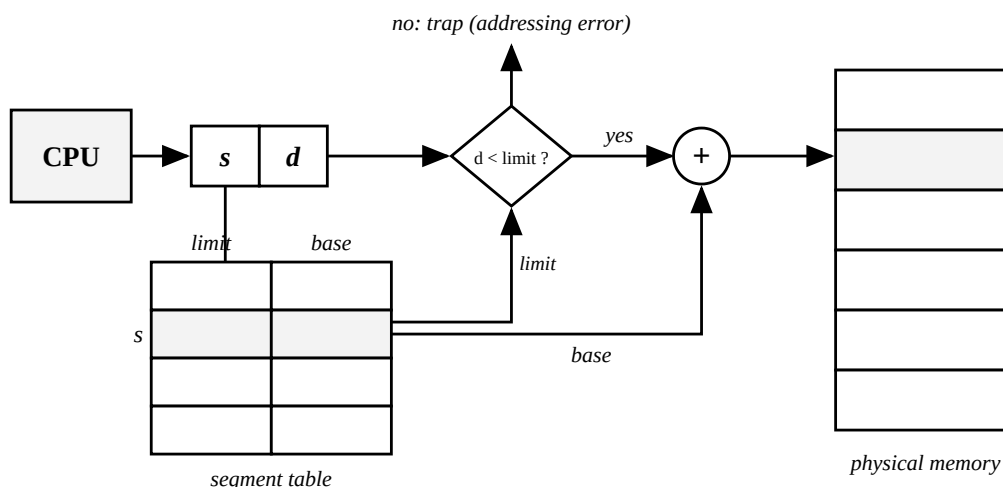


Fig.: Segmentation hardware

1. The segment number s is used as an index into the segment table.
2. The offset d is compared with the limit of that segment. If $d \geq \text{limit}$, a **trap** (addressing error) is sent to the OS.
3. If $d < \text{limit}$, physical address = base + d .

Example: segment 2 has base 5200 and limit 600; logical address (2, 120) $\rightarrow 5200 + 120 = 5320$;
 logical address (2, 650) $\rightarrow$ trap, since $650 \geq 600$.

(Explanation 2 M, diagram 2 M)

d) File operations: create, read, write, delete.

4 M

1. **Create:** two steps — find space for the file in the file system, and make an entry for the new file in the directory (name, location, attributes).
2. **Write:** a system call specifies the file name and the data to be written. The directory is searched to find the file, and the system keeps a **write pointer** to the location where the next write is to take place; it is updated after every write.
3. **Read:** a system call specifies the file name and where (in memory) the next block should be put. The directory is searched, and a **read pointer** gives the location of the next read; it is updated after the read. (Read and write usually share one current-file-position pointer.)
4. **Delete:** the directory is searched for the named file; then all its file space is released (so it can be reused) and the directory entry is erased.

(Each operation 1 M)

Q.4

a) Multilevel queue scheduling.

4 M

In **multilevel queue scheduling**, the ready queue is partitioned into **several separate queues**. Processes are **permanently assigned** to one queue based on a property such as memory size, process priority or process type (foreground/interactive vs background/batch). Each queue has its **own scheduling algorithm** — e.g. the foreground queue uses Round Robin and the background queue uses FCFS.

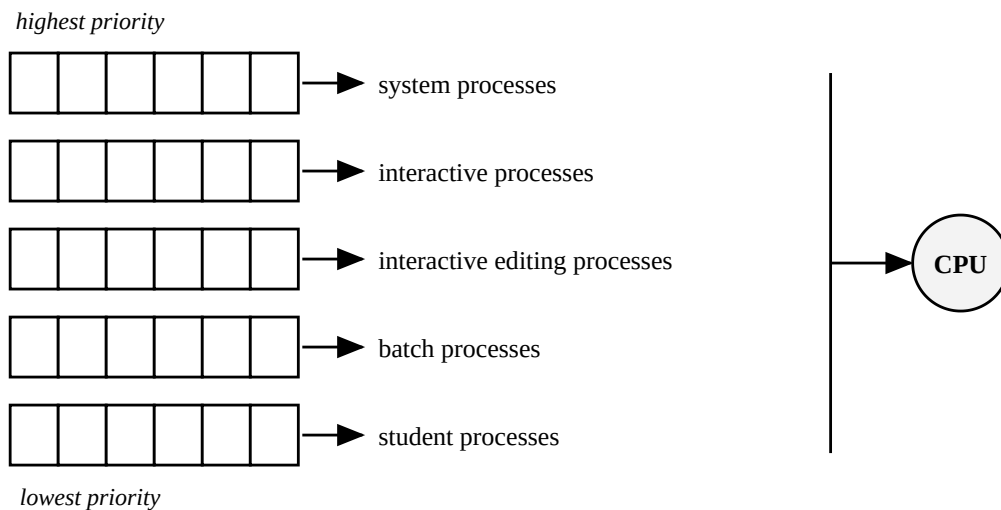


Fig.: Multilevel queue scheduling

- Scheduling among the queues is usually **fixed-priority preemptive**: no process in a lower queue runs unless all higher queues are empty; a higher-priority arrival preempts a lower one.
- Alternatively, **time slicing** between queues: e.g. 80% CPU time to foreground (RR) and 20% to background (FCFS).
- Disadvantage: low-priority processes may **starve**; processes cannot move between queues (that is allowed in multilevel feedback queue).

(Explanation 2 M, diagram 2 M)

b) Safe state; safe, unsafe and deadlock.

4 M

Safe state: A state is **safe** if the system can allocate resources to each process (up to its maximum) in some order and still avoid a deadlock. Formally, the system is in a safe state only if there exists a **safe sequence** $\langle P_1, P_2, \dots, P_n \rangle$ such that, for each P_i , the resources P_i can still request can be satisfied by the currently available resources plus the resources held by all P_j with $j < i$.

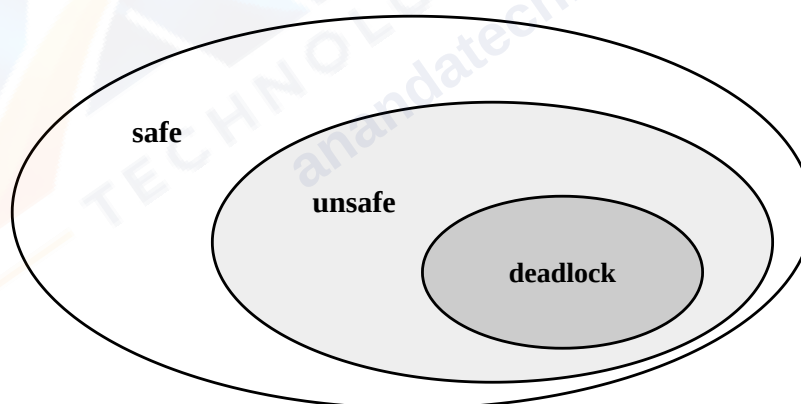


Fig.: Safe, unsafe and deadlocked state spaces

- A **safe state is not a deadlocked state**; in a safe state the OS can always avoid deadlock.
- A **deadlocked state is an unsafe state**.
- Not all unsafe states are deadlocks — an unsafe state **may lead** to a deadlock (it depends on future requests).

- Deadlock avoidance (e.g. Banker's algorithm) grants a request only if the system stays in a safe state.

(Safe state 1 M, diagram 1 M, relation 2 M)

c) Benefits of threads.

4 M

1. **Responsiveness:** multithreading an interactive application allows a program to continue running even if part of it is blocked or doing a long operation (e.g. a browser displays a page while one thread loads an image).
2. **Resource sharing:** threads share the memory and resources of the process they belong to by default, so several threads of activity can work within the same address space.
3. **Economy:** allocating memory and resources for process creation is costly; since threads share resources, it is much cheaper to create threads and to context-switch between them.
4. **Scalability (utilisation of multiprocessor architectures):** threads of one process can run in parallel on different processors/cores, increasing concurrency; a single-threaded process can use only one CPU.

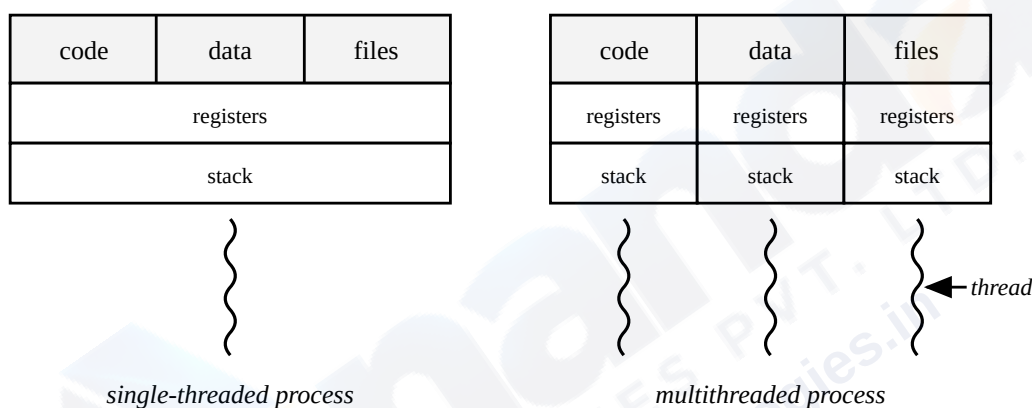


Fig.: Single-threaded and multithreaded process

(Each benefit 1 M)

d) Bitmap vs linked-list free-space management.

4 M

Bitmap (bit vector)	Linked list
Each disk block is represented by one bit (1 = free, 0 = allocated).	All free blocks are linked together; each free block stores a pointer to the next free block.
Needs extra space for the bit vector (one bit per block), kept in memory.	No extra space needed — pointers are stored in the free blocks themselves.
Easy and fast to find the first free block or n consecutive free blocks.	Finding contiguous free blocks is difficult; traversal requires reading each block (slow I/O).
Inefficient for very large disks unless the whole vector is in memory.	Getting the first free block is easy (head of the list).
Example: 001111001111110001100000...	Example: head → 2 → 3 → 4 → 5 → 8 → ... → NULL

(Any four points, 1 M each)

e) Contiguous allocation disadvantages; linked allocation.

4 M

Disadvantages of contiguous allocation: 1) **External fragmentation** of free space. 2) The **size of the file must be declared** at creation time. 3) A file **cannot grow** easily if the following blocks are in use. 4) Needs **compaction**, which is time consuming.

Linked allocation: each file is a **linked list of disk blocks**, which may be scattered anywhere on the disk. The directory contains a pointer to the **first and last block** of the file. Each block contains a pointer to the next block (the last block has nil/-1). No external fragmentation; a file can grow as long as free blocks exist. But it supports only sequential access efficiently, and pointers use space in every block.

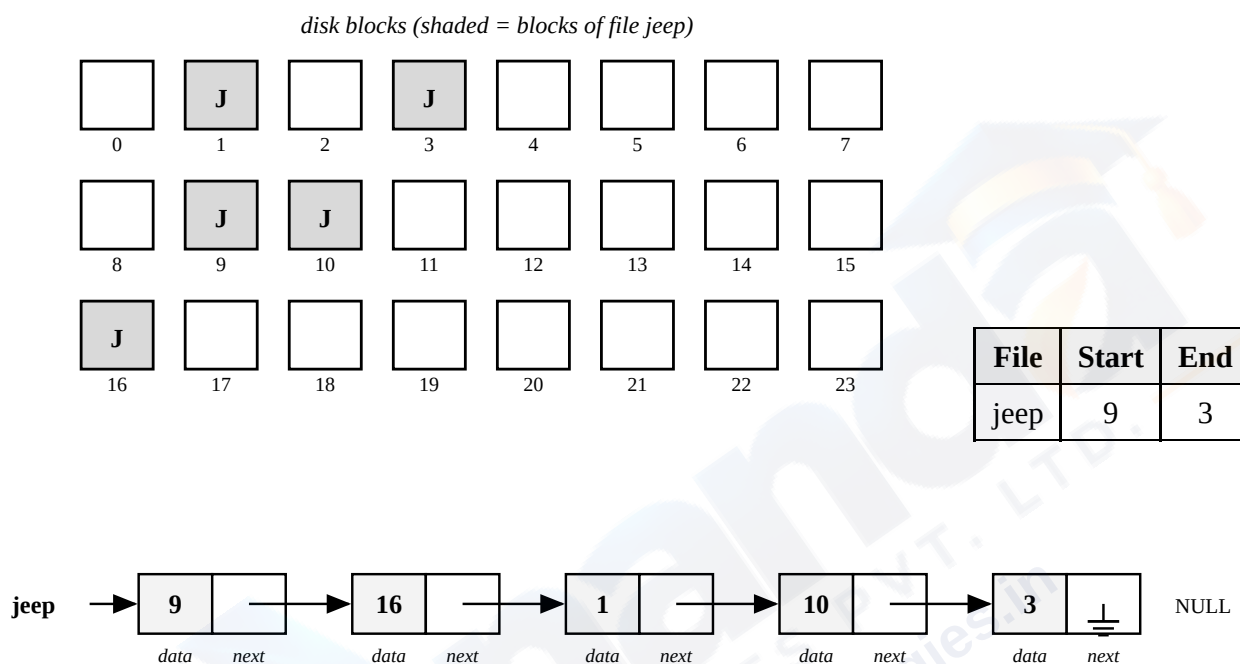


Fig.: Linked allocation — blocks of file 'jeep' (data | pointer to next block); block 3 is the last block

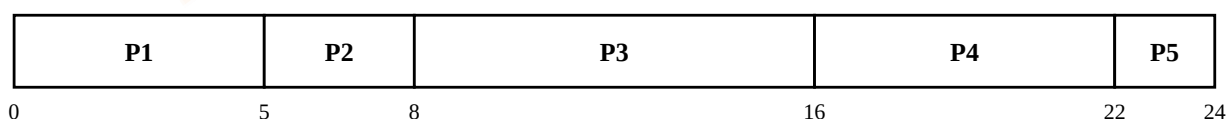
(Disadvantages 1 M, explanation 2 M, diagram 1 M)

Q.5

a) FCFS, SJF and preemptive Priority.

6 M

FCFS (First Come First Served)



Gantt chart

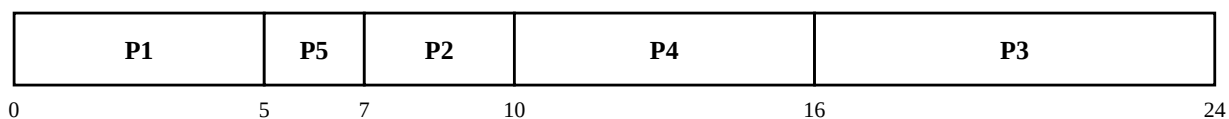
Process	AT	BT	Completion time (CT)	TAT = CT - AT	WT = TAT - BT
P1	0	5	5	5 - 0 = 5	5 - 5 = 0
P2	1	3	8	8 - 1 = 7	7 - 3 = 4
P3	2	8	16	16 - 2 = 14	14 - 8 = 6

P4	3	6	22	$22 - 3 = 19$	$19 - 6 = 13$
P5	4	2	24	$24 - 4 = 20$	$20 - 2 = 18$

Average turnaround time = $65 / 5 = 13$ ms

Average waiting time = $41 / 5 = 8.2$ ms

SJF (Shortest Job First, non-preemptive)



Gantt chart

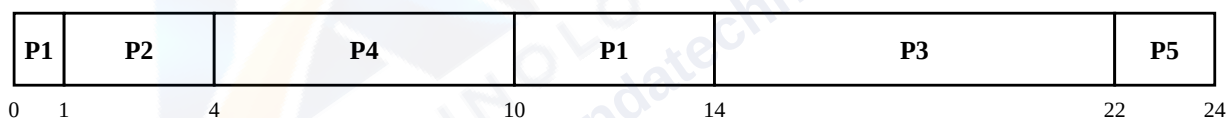
Process	AT	BT	Completion time (CT)	TAT = CT - AT	WT = TAT - BT
P1	0	5	5	$5 - 0 = 5$	$5 - 5 = 0$
P2	1	3	10	$10 - 1 = 9$	$9 - 3 = 6$
P3	2	8	24	$24 - 2 = 22$	$22 - 8 = 14$
P4	3	6	16	$16 - 3 = 13$	$13 - 6 = 7$
P5	4	2	7	$7 - 4 = 3$	$3 - 2 = 1$

Average turnaround time = $52 / 5 = 10.4$ ms

Average waiting time = $28 / 5 = 5.6$ ms

Priority (preemptive)

At each arrival the highest-priority (lowest number) ready process runs; P1 is preempted at $t = 1$ by P2 (priority 1).



Gantt chart

Process	AT	BT	Completion time (CT)	TAT = CT - AT	WT = TAT - BT
P1	0	5	14	$14 - 0 = 14$	$14 - 5 = 9$
P2	1	3	4	$4 - 1 = 3$	$3 - 3 = 0$
P3	2	8	22	$22 - 2 = 20$	$20 - 8 = 12$
P4	3	6	10	$10 - 3 = 7$	$7 - 6 = 1$
P5	4	2	24	$24 - 4 = 20$	$20 - 2 = 18$

Average turnaround time = $64 / 5 = 12.8$ ms

Average waiting time = $40 / 5 = 8$ ms

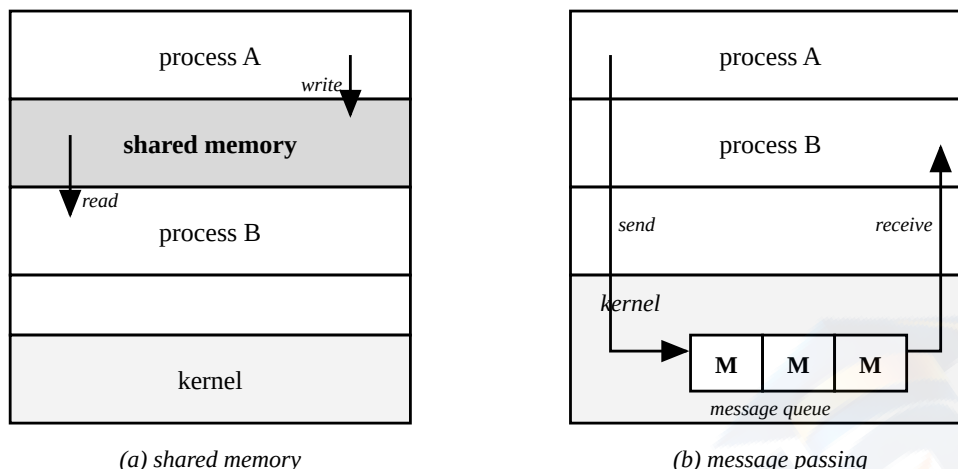
Algorithm	Average WT (ms)	Average TAT (ms)
FCFS	8.2	13

SJF (non-preemptive)	5.6	10.4
Priority (preemptive)	8	12.8

(Each algorithm: Gantt chart 1 M, averages 1 M)

b) Shared memory vs message passing.

6 M



(a) shared memory

(b) message passing

Fig.: Models of IPC — (a) shared memory, (b) message passing

Shared-memory model	Message-passing model
A region of memory is shared by the cooperating processes; they exchange data by reading and writing in it.	Processes exchange data by sending and receiving messages through the kernel (send / receive).
Kernel is needed only to establish the shared region (system calls); after that no kernel help is needed.	Every message passes through the kernel (system call for each send/receive).
Faster — data is accessed at memory speed.	Slower — system calls and copying for every message.
Processes must handle synchronisation themselves (to avoid writing at the same location simultaneously).	Synchronisation is provided by the send/receive primitives (blocking / non-blocking).
Suitable for large amounts of data on the same computer.	Suitable for small amounts of data and for distributed systems (different computers).
Example: shmget(), shmat() in UNIX; producer–consumer with a shared buffer.	Example: pipes, message queues, sockets; chat applications.

(Diagrams 2 M, any four points of comparison 4 M)

c) Banker's algorithm with four resource types.

6 M

Step 1: Need matrix (Need = Max – Allocation)

Process	Allocation (A B C D)	Max (A B C D)	Need (A B C D)
P0	1 0 2 0	3 2 2 2	2 2 0 2
P1	2 1 1 1	4 1 3 3	2 0 2 2
P2	0 1 2 2	1 3 4 4	1 2 2 2

P3	3 0 0 1	3 2 1 1	0 2 1 0
P4	1 2 1 0	2 4 3 2	1 2 2 2

Step 2: Safety algorithm. Work = Available = (1, 2, 1, 1); Finish[i] = false for all i. Find a process with Finish = false and Need ≤ Work; then Work = Work + Allocation, Finish = true. Repeat.

Process	Need ≤ Work	Result	New Work = Work + Allocation
P0	(2, 2, 0, 2) ≤ (1, 2, 1, 1) ?	No — wait	—
P1	(2, 0, 2, 2) ≤ (1, 2, 1, 1) ?	No — wait	—
P2	(1, 2, 2, 2) ≤ (1, 2, 1, 1) ?	No — wait	—
P3	(0, 2, 1, 0) ≤ (1, 2, 1, 1) ?	Yes	(1, 2, 1, 1) + (3, 0, 0, 1) = (4, 2, 1, 2)
P4	(1, 2, 2, 2) ≤ (4, 2, 1, 2) ?	No — wait	—
P0	(2, 2, 0, 2) ≤ (4, 2, 1, 2) ?	Yes	(4, 2, 1, 2) + (1, 0, 2, 0) = (5, 2, 3, 2)
P1	(2, 0, 2, 2) ≤ (5, 2, 3, 2) ?	Yes	(5, 2, 3, 2) + (2, 1, 1, 1) = (7, 3, 4, 3)
P2	(1, 2, 2, 2) ≤ (7, 3, 4, 3) ?	Yes	(7, 3, 4, 3) + (0, 1, 2, 2) = (7, 4, 6, 5)
P4	(1, 2, 2, 2) ≤ (7, 4, 6, 5) ?	Yes	(7, 4, 6, 5) + (1, 2, 1, 0) = (8, 6, 7, 5)

All processes can finish. **The system is in a SAFE state. Safe sequence: <P3, P0, P1, P2, P4>**

(Need matrix 2 M, safety steps 3 M, safe sequence 1 M)

Q.6

a) System call; execution; types.

6 M

System call: System calls provide an **interface to the services made available by the operating system**. When a user program needs an OS service (file I/O, process creation, device access), it makes a system call; these are usually available as routines written in C/C++ and accessed through an API (e.g. POSIX API, Win32 API).

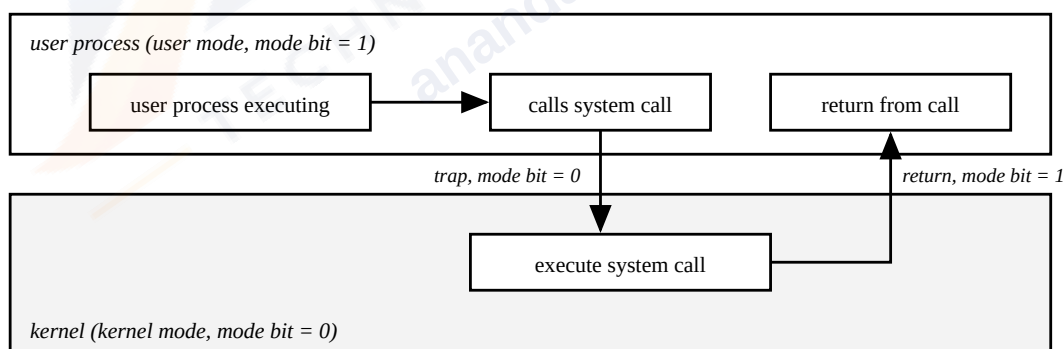


Fig.: Transition from user mode to kernel mode during a system call

Execution of a system call:

1. The user process executes in **user mode** (mode bit = 1) and calls a library function, e.g. `read()`.
2. Parameters are passed in registers, in a block/table in memory, or on the stack; the system-call number is placed in a register.

3. A **trap** (software interrupt) instruction switches the CPU to **kernel mode** (mode bit = 0) and transfers control to the system-call handler.
4. The kernel looks up the system-call table using the number, executes the required service routine.
5. Control returns to the user program with the result, and the mode bit is set back to user mode (1).

Types (any two):

- **Process control:** create / terminate process, load, execute, wait for time / event, allocate and free memory — `fork()`, `exec()`, `exit()`, `wait()`.
- **File management:** create / delete file, open, close, read, write, reposition — `open()`, `read()`, `write()`, `close()`.
- **Device management:** request / release device, read, write — `ioctl()`.
- **Information maintenance:** get/set time or date, system data — `getpid()`, `alarm()`, `sleep()`.
- **Communication:** create/delete connection, send/receive messages, shared memory — `pipe()`, `shmget()`, `mmap()`.

(Definition 1 M, diagram and execution 3 M, two types 2 M)

b) FIFO and LRU page replacement (4 frames).

6 M

Reference string (20 references): 3, 5, 2, 6, 5, 3, 1, 7, 5, 3, 5, 2, 8, 7, 2, 5, 3, 5, 2, 7; frames = 4.

FIFO (First In First Out)

Rule: replace the page that was brought into memory first (oldest page).

Ref.	3	5	2	6	5	3	1	7	5	3	5	2	8	7	2	5	3	5	2	7
F1	3	3	3	3	3	3	1	1	1	1	1	2	2	2	2	2	3	3	3	3
F2		5	5	5	5	5	5	7	7	7	7	7	8	8	8	8	8	8	2	2
F3			2	2	2	2	2	2	5	5	5	5	5	7	7	7	7	7	7	7
F4				6	6	6	6	6	6	3	3	3	3	3	3	5	5	5	5	5
Fault	F	F	F	F	H	H	F	F	F	F	H	F	F	F	H	F	F	H	F	H

F = page fault, H = hit. **Number of page faults (FIFO) = 14** (hits = 6); page fault ratio = $14/20 = 0.70$

LRU (Least Recently Used)

Rule: replace the page that has not been used for the longest time in the past.

Ref.	3	5	2	6	5	3	1	7	5	3	5	2	8	7	2	5	3	5	2	7
F1	3	3	3	3	3	3	3	3	3	3	3	3	3	7	7	7	7	7	7	7
F2		5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5
F3			2	2	2	2	1	1	1	1	1	2	2	2	2	2	2	2	2	2
F4				6	6	6	6	7	7	7	7	7	8	8	8	8	3	3	3	3
Fault	F	F	F	F	H	H	F	F	H	H	H	F	F	F	H	H	F	H	H	H

F = page fault, H = hit. **Number of page faults (LRU) = 10** (hits = 10); page fault ratio = $10/20 = 0.50$

Result: FIFO = 14 page faults; LRU = 10 page faults. LRU performs better for this string because it keeps recently used pages.

(Each algorithm 3 M)

c) Linux commands **ps -u**, **wait**, **exit**.

6 M

i) ps -u: ps displays the status of processes; the -u option selects the processes that belong to a given **user** (user name or UID).

Syntax: **ps -u username**

```
$ ps -u student
  PID TTY          TIME CMD
 1834 ?            00:00:00 systemd
 2314 pts/0        00:00:00 bash
 2398 pts/0        00:00:00 ps
```

ii) wait: a shell built-in that **waits for a background process (job) to finish** and returns its exit status. Without an argument it waits for all background jobs of the current shell. Useful in scripts so that the next command runs only after the background job completes.

Syntax: **wait [PID or %job]**

```
$ sleep 10 &
[1] 2405
$ wait 2405          # prompt returns only after sleep finishes
$ echo $?
0
```

iii) exit: **terminates the current shell** (or a shell script) and returns an exit status (0 = success, non-zero = error) to the parent process. In a terminal, it logs out / closes the terminal session.

Syntax: **exit [n]**

```
$ bash              # start a child shell
$ exit 3            # child shell ends with status 3
exit
$ echo $?
3
```

(PID numbers differ on every system.)

(Each command: use 1 M, syntax with example 1 M)

Note: This is a **model question paper prepared by Ananda Technologies** only for practice and exam preparation, based on the pattern of previous MSBTE question papers and the K-scheme syllabus. It is **not an official MSBTE paper**, and it does **not** mean that these questions will come in the exam. Answers follow standard textbook points; marks shown are indicative. Free for students — anandatechnologies.in