UNIT–1 : Introduction, Processes & Processor Management (PYQ Solution)

PART–A Answers (≈70 Words Each)

Q1. What is an operating system? Give two examples. / Define operating system. List the objectives of operating system. (2021, 2023, 2025)

An operating system (OS) is system software that acts as an interface between the user and the computer hardware and manages all resources such as CPU, memory, files and I/O devices.

Objectives: convenience for the user, efficient use of hardware, and ability to evolve without breaking existing programs.

Examples: Windows and Linux (also UNIX, Android, macOS).

Q2. Explain the features of operating system. (2022, Feb 2024)

The main features of an operating system are:

- **Process management: creation, scheduling and termination of processes.**
- **Memory management: allocation, paging and virtual memory.**
- **File management: creating, storing, protecting and retrieving files.**
- **Device management: controlling I/O devices through drivers.**
- **Protection and security: user authentication and access control.**
- **Error detection, accounting and a user interface (CLI or GUI).**

Q3. What is the kernel? / Differentiate between kernel and shell. / What is Kernel? (2019, 2021, 2022, 2024)

The kernel is the core part of the OS. It stays in main memory all the time and directly controls the hardware, memory, processes and I/O.

Kernel	Shell
Core of the OS, talks to hardware	Command interpreter, talks to the user
Always in memory	Runs as a user program
Manages process, memory, files	Passes commands to the kernel through system calls

---

Q4. What is the difference between user mode and kernel mode? (2025)

User mode	Kernel mode
Mode bit = 1, restricted instructions	Mode bit = 0, privileged instructions allowed
User programs run here	OS code runs here
A crash affects only that program	A crash can bring down the whole system

A trap or system call switches the CPU from user mode to kernel mode, and the return instruction switches it back.

Q5. What do you mean by system call? (2024)

A system call is the programmatic interface through which a user program requests a service from the kernel. The call executes a trap, the CPU moves from user mode to kernel mode, the kernel performs the service and control returns to the program.

Types: process control (`fork`, `exit`), file management (`open`, `read`, `write`), device management, information maintenance (`getpid`) and communication (`pipe`).

Q6. What are batch system? / What is multi user operating system? (2021)

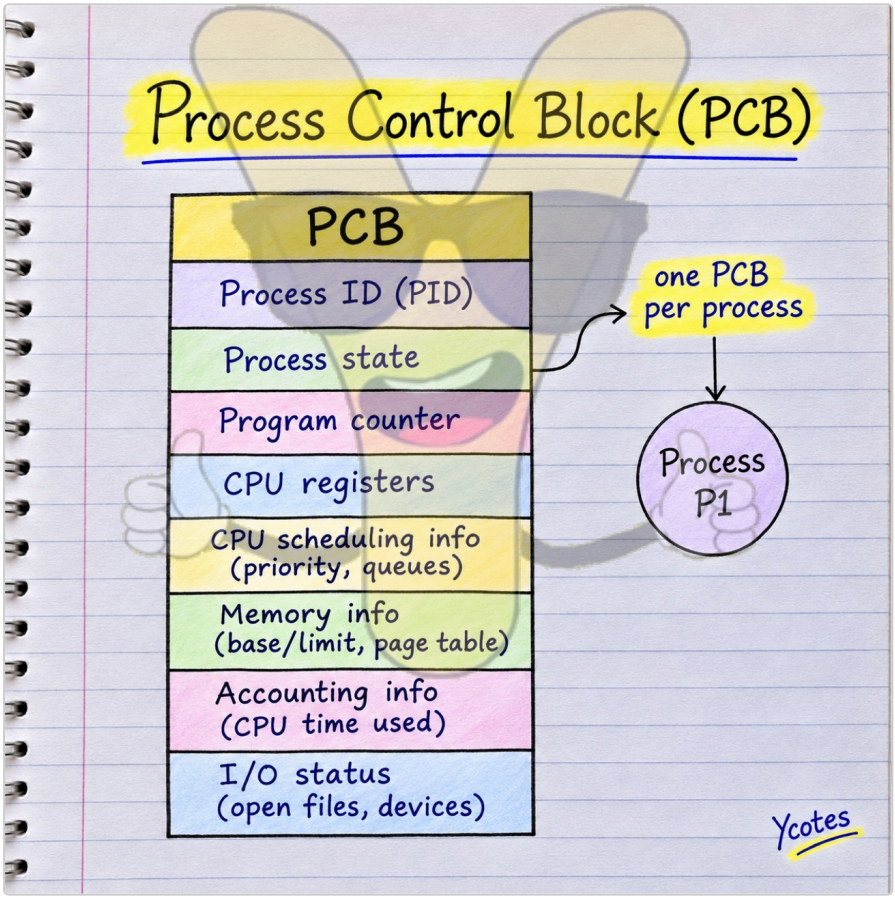
Batch system: similar jobs are grouped into a batch and executed one after another by the operating system without any user interaction, for example payroll processing. It gives high throughput but poor response time.

Multi-user operating system: it allows many users to use the same computer at the same time through different terminals. CPU time is shared among users, for example UNIX and Linux servers.

Q7. What is a process? / What is process control block? (2021, 2024)

A process is a program in execution. It consists of program code, data, stack, heap and the program counter.

The Process Control Block (PCB) is the kernel data structure that stores all information about one process: PID, state, program counter, CPU registers, scheduling information, memory-management information, accounting and I/O status. It is created with the process and deleted when the process terminates.



Q8. Differentiate between process and thread. / What is thread? (2022, 2023, 2025)

A thread is the smallest unit of CPU execution, also called a lightweight process.

Process	Thread
Heavyweight, has its own address space	Lightweight, shares the address space of its process
Slow to create and switch	Fast to create and switch
Needs IPC to communicate	Shares data directly

Threads of one process share code, data and files but have their own program counter, registers and stack.

Q9. What is multithreading? Mention one advantage. (2025)

Multithreading means running several threads inside a single process so that they execute concurrently and share the same code, data and open files.

Advantage: better responsiveness. If one thread is blocked, for example waiting for I/O, the other threads of the process continue to run, so the application stays active. Other benefits are resource sharing, economy and use of multiple cores.

Q10. What is semaphore? (2019, Feb 2024)

A semaphore is an integer variable used for process synchronisation. It is accessed only through two atomic operations.

```
wait(S):   while (S <= 0) ;  S = S - 1;    // P operation
signal(S): S = S + 1;                      // V operation
```

A binary semaphore (0 or 1) gives mutual exclusion and a counting semaphore controls access to a number of identical resources.

Q11. What are the disadvantages of semaphore? (2023)

The disadvantages of semaphores are:

- **Busy waiting: the waiting process keeps testing the value and wastes CPU time (spinlock).**
- **Deadlock: a wrong order of `wait` and `signal` can block processes forever.**
- **Priority inversion: a high-priority process waits for a low-priority one.**

- **Starvation if the waiting queue is served unfairly.**
- **Hard to use correctly in large programs, because one mistake breaks mutual exclusion.**

Q12. Briefly explain the two methods of inter process communication. (2019)

Two methods of inter-process communication (IPC) are:

1. **Shared memory: processes share a memory region and read and write it directly. It is fast but needs explicit synchronisation.**
2. **Message passing: processes exchange messages using `send()` and `receive()` through the kernel. It is slower but easier to use and suits distributed systems.**

Q13. What is starvation? How can we overcome it? (2019, Feb 2024)

Starvation is a situation in which a process waits indefinitely because other processes always get the resource or CPU before it, for example a low-priority process in priority scheduling or a long job in SJF.

Remedy: aging. The priority of a process is increased gradually the longer it waits, so that it eventually gets the CPU. Fair queueing such as FCFS also avoids starvation.

Q14. What are schedulers? / What is the role of dispatcher? / What are context switches? (2021, 2022, 2023)

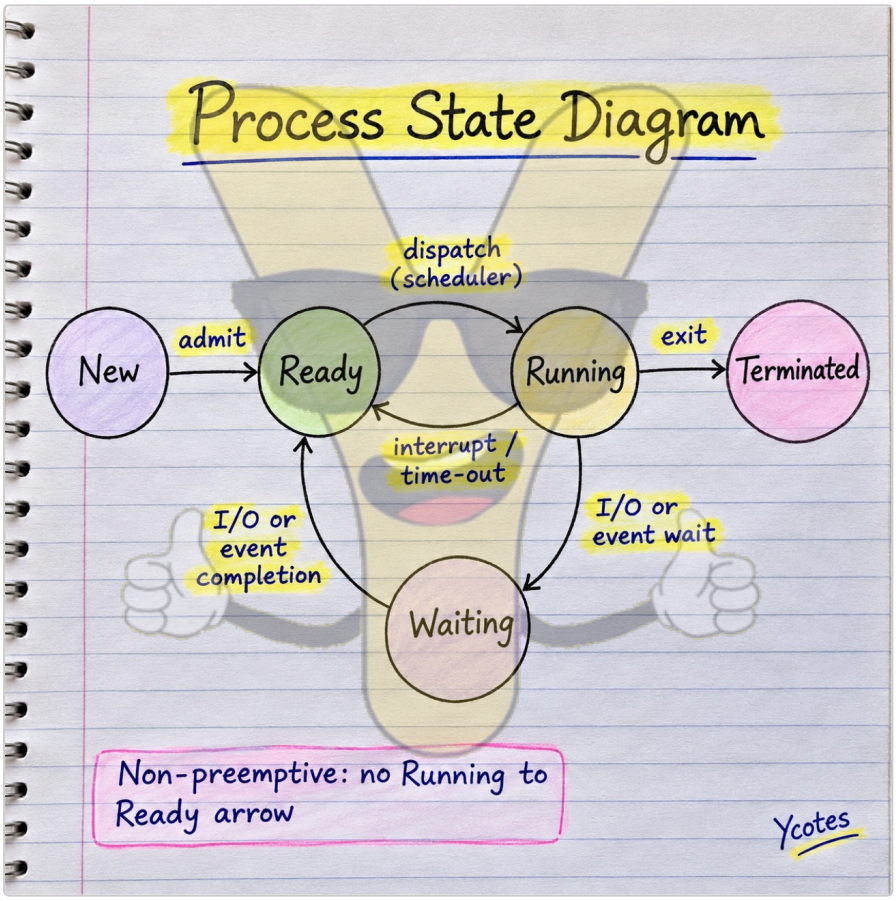
- **Scheduler: selects which process gets the CPU or enters memory. There are long-term, short-term and medium-term schedulers.**
- **Dispatcher: gives the CPU to the process chosen by the short-term scheduler. It does the context switch, switches to user mode and jumps to the right instruction. The time it takes is the dispatch latency.**
- **Context switch: saving the state of the running process in its PCB and loading the state of the next process.**

The scheduler decides and the dispatcher acts.

Q15. Draw process state diagram. (2019, Feb 2024)

A process moves through five states: new, ready, running, waiting and terminated.

- **New to ready: admit**
- **Ready to running: dispatch**
- **Running to ready: time-out or interrupt**
- **Running to waiting: I/O or event wait**
- **Waiting to ready: I/O or event completion**
- **Running to terminated: exit**



PART–B Answers (≈250 Words Each)

Q1. Explain the need and various services provided by operating system. (2019)

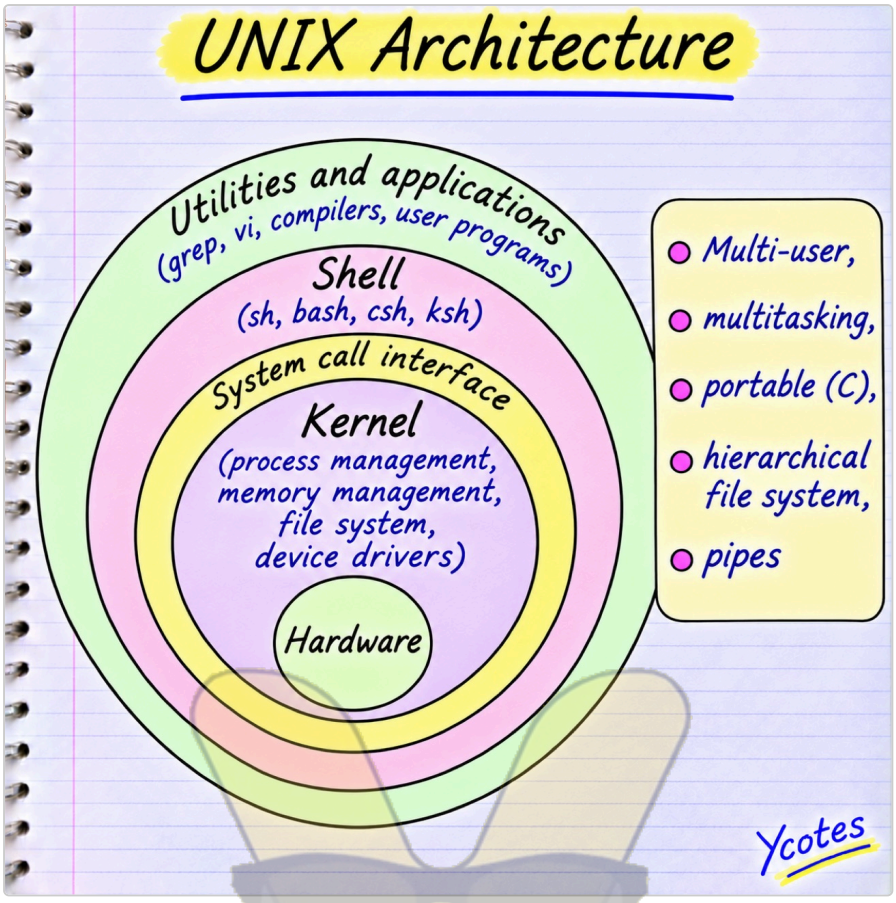
Need: bare hardware is difficult to program and has to be shared among many users and programs. The OS hides the hardware complexity, provides a convenient interface and allocates resources fairly and efficiently.

Services provided to the user:

1. **User interface:** command line, graphical or batch interface.
2. **Program execution:** load a program into memory, run it and end it normally or abnormally.
3. **I/O operations:** programs cannot access devices directly, so the OS performs I/O for them.
4. **File-system manipulation:** create, delete, read, write and search files and directories.
5. **Communication:** between processes on the same or different computers by shared memory or message passing.
6. **Error detection:** handling CPU, memory, I/O and program errors so that computing continues correctly.

Services that ensure efficient operation of the system:

- **Resource allocation:** CPU cycles, memory, files and devices among multiple jobs.
- **Accounting:** keeping track of which user uses how much of which resource.
- **Protection and security:** controlling access to resources and authenticating users.



Together these services make the computer easy to use and keep it efficient and secure.

Q2. Explain different operations performed by the operating system. (2023)

The operating system performs the following operations.

1. **Process management:** creates and terminates processes, schedules the CPU, suspends and resumes processes, and provides synchronisation and communication between them.
2. **Memory management:** keeps track of which memory is used by whom, allocates and deallocates memory, and swaps processes or pages in and out.
3. **File management:** creates, deletes, reads, writes, copies and protects files and directories, and manages free disk space.
4. **Device (I/O) management:** buffering, caching, spooling, device drivers and interrupt handling.
5. **Secondary-storage management:** disk space allocation and disk scheduling.

6. **Protection and security:** the mode bit separates user and kernel mode, access rights and passwords protect resources.
7. **Dual-mode and interrupt operation:** the OS handles interrupts, traps and system calls, and uses a timer so that no program can keep the CPU forever.
8. **Networking and user interface:** communication across systems and a CLI or GUI for the user.

These operations let the OS act as a resource manager and a control program for all user programs.

Q3. Describe multilevel feedback queue scheduling with the help of an example. (2019)

Multilevel Feedback Queue (MLFQ) scheduling uses several ready queues with different priorities and lets a process **move between queues** according to its CPU behaviour. This avoids both starvation and the need to know burst times in advance.

Example configuration:

Queue	Algorithm	Quantum
Q0 (highest priority)	Round Robin	8 ms
Q1	Round Robin	16 ms
Q2 (lowest priority)	FCFS	until completion

Working:

1. A new process enters Q0 and gets 8 ms.
2. If it does not finish in 8 ms, it is preempted and moved to the tail of Q1.
3. In Q1 it gets 16 ms; if still unfinished it is demoted to Q2 and runs there in FCFS order.
4. Q1 is served only when Q0 is empty, and Q2 only when Q0 and Q1 are empty. A new arrival in Q0 preempts a running process of a lower queue.
5. A process that waits too long in a low queue is promoted to a higher queue (aging).

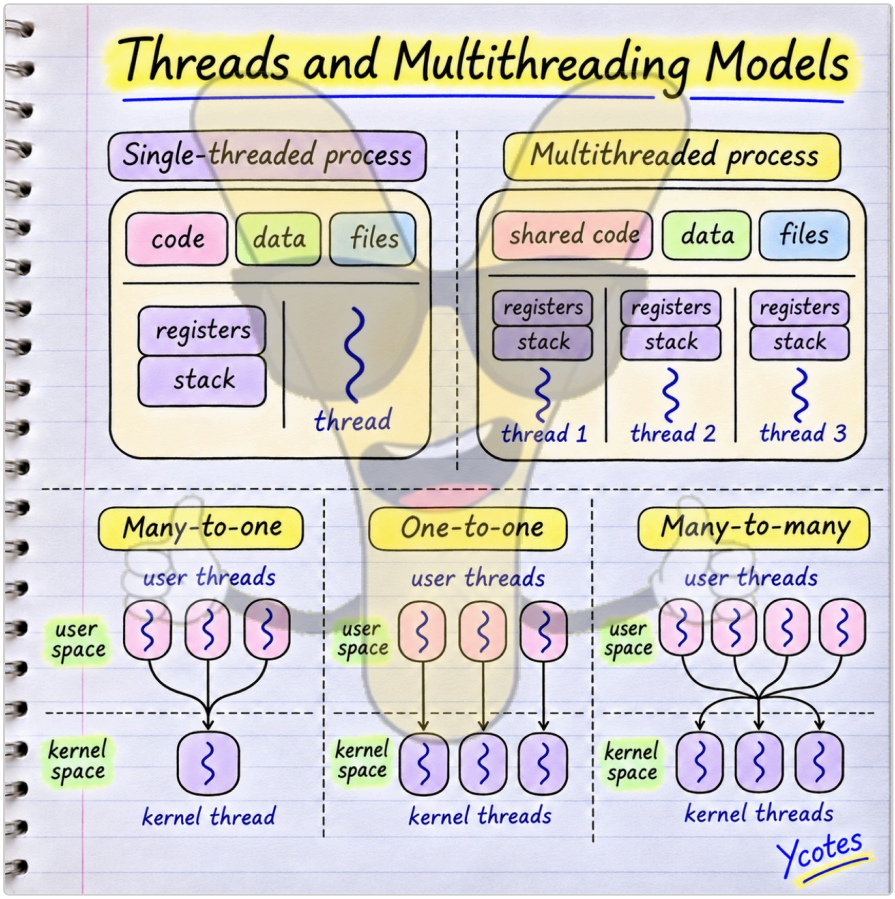
Numerical illustration: a job of 30 ms uses 8 ms in Q0, 16 ms in Q1 and its last 6 ms in Q2. A short job of 5 ms finishes in Q0 itself.

Parameters: number of queues, algorithm of each queue, rule to promote, rule to demote, and the queue in which a new process enters.

Advantage: short and interactive jobs get quick service; long jobs do not starve.
Disadvantage: it is the most complex scheduler to design and tune.

Q4. What is thread management in operating system? Explain the applications of thread. (2021)

Thread management is the creation, scheduling, synchronisation and termination of threads of a process. It is done by a thread library (POSIX `pthreads`, Java threads, Windows threads) or by the kernel. A thread has its own thread ID, program counter, register set and stack, but shares code, data and open files with the other threads of its process.



Thread operations: create, run, join (wait for finish), yield, cancel and thread-local storage.

Multithreading models: many-to-one, one-to-one and many-to-many mapping of user threads to kernel threads.

Benefits:

- **Responsiveness:** the program continues when one thread blocks.

- **Resource sharing:** threads share memory and files.
- **Economy:** creating and switching threads is cheaper than for processes.
- **Scalability:** threads run in parallel on multiple cores.

Applications of threads:

1. **Web server:** one thread for each client request.
2. **Word processor:** separate threads for typing, spell check and auto-save.
3. **Web browser:** a thread for each tab, download and rendering.
4. **Games and multimedia:** graphics, sound and input handled in parallel.
5. **Parallel computing:** matrix multiplication and sorting on many cores.
6. **Database servers:** many queries processed concurrently.

Q5. Explain in detail the following CPU scheduling algorithms - (a) Priority Scheduling (b) Round Robin. (2021, Feb 2024)

Priority scheduling: the CPU is allocated to the process with the highest priority. In RTU problems a smaller number means a higher priority unless stated otherwise. Equal priorities are served FCFS. It can be preemptive (a newly arrived higher-priority process takes the CPU) or non-preemptive. Its problem is **starvation** of low-priority processes, and the solution is **aging**.

Worked example (all arrive at 0, smaller number = higher priority): P1 (BT 5, pri 3), P2 (BT 2, pri 1), P3 (BT 4, pri 2).

Priority scheduling

Time	0-2	2-6	6-11
Process	P2	P3	P1

Process	AT	BT	Priority	CT	TAT = CT - AT	WT = TAT - BT
P1	0	5	3	11	11	6
P2	0	2	1	2	2	0
P3	0	4	2	6	6	2

Average WT = (6 + 0 + 2) / 3 = 8/3 = **2.67 ms**; Average TAT = (11 + 2 + 6) / 3 = 19/3 = **6.33 ms**

Round Robin (RR): designed for time-sharing systems. Every process gets a small time quantum q. A process not finished in its quantum is preempted and put at the tail of the circular ready queue.

- If there are n processes, each waits at most $(n-1)q$ before getting the CPU again.
- Large q makes RR behave like FCFS; very small q causes too many context switches.
- It gives good response time but a higher average waiting time than SJF.

Worked example (q = 2, all arrive at 0): P1 (BT 5), P2 (BT 3), P3 (BT 2).

Round Robin (q = 2)

Time	**0-2**	**2-4**	**4-6**	**6-8**	**8-9**	**9-10**
Process	P1	P2	P3	P1	P2	P1

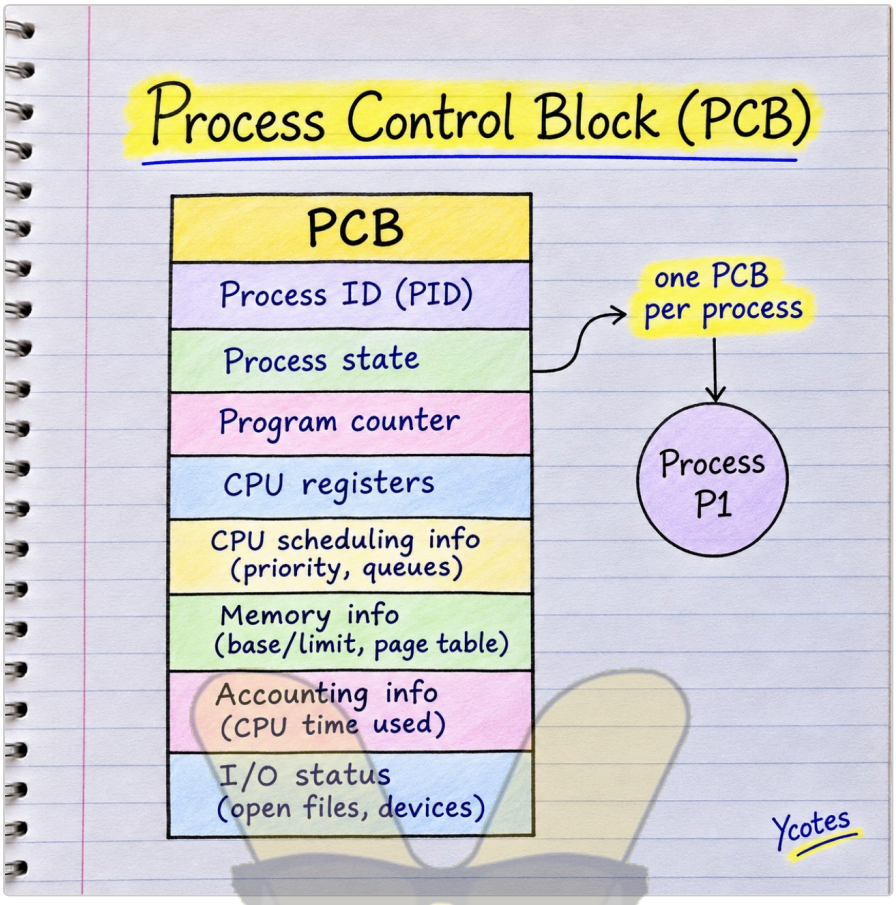
Process	**AT**	**BT**	**CT**	**TAT = CT - AT**	**WT = TAT - BT**
P1	0	5	10	10	5
P2	0	3	9	9	6
P3	0	2	6	6	4

Average WT = (5 + 6 + 4) / 3 = 15/3 = **5 ms**; Average TAT = (10 + 9 + 6) / 3 = 25/3 = **8.33 ms**

Comparison: priority scheduling favours important jobs and may starve others; RR is fair and gives good response time but does not consider importance.

Q6. What is process control block? Explain with diagram and its contents. (2021)

The **Process Control Block (PCB)** is a data structure maintained by the kernel for every process. It is also called the task control block and is the only representation of the process to the OS. It is created when the process is created and removed when it terminates.



Contents of the PCB:

1. **Process ID (PID)** and parent PID.
2. **Process state:** new, ready, running, waiting or terminated.
3. **Program counter:** address of the next instruction.
4. **CPU registers:** accumulator, index registers, stack pointer, saved at an interrupt.
5. **CPU-scheduling information:** priority, scheduling queue pointers.
6. **Memory-management information:** base and limit registers, page or segment tables.
7. **Accounting information:** CPU time used, time limits, job number.
8. **I/O status information:** list of open files and I/O devices allocated.

Use: during a context switch the OS saves the state of the running process into its PCB and loads the state of the next process from its PCB. This lets the process resume exactly where it stopped. The PCBs of all ready processes are linked in the ready queue.

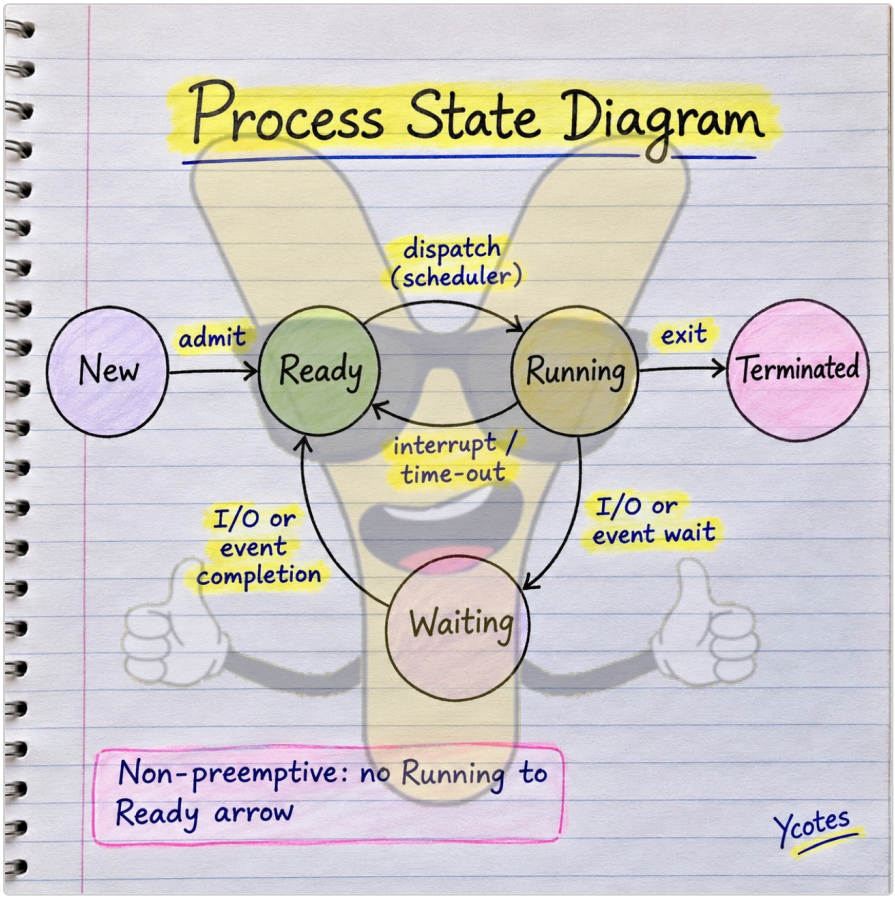
Q7. What are preemptive and non-preemptive scheduling process?

Explain the process state diagram in detail. (2022)

Preemptive scheduling: the CPU can be taken away from a running process before it finishes, for example when its time slice ends or a higher-priority process arrives. Examples: SRTF, Round Robin, preemptive priority. It gives better response time but has overhead and needs synchronisation care.

Non-preemptive scheduling: once the CPU is given to a process it keeps it until it finishes or blocks for I/O. Examples: FCFS, SJF. It is simple and has low overhead but a long job can delay all others (convoy effect).

Process state diagram:



State	Meaning
New	Process is being created
Ready	Waiting in the ready queue for the CPU
Running	Instructions are being executed
Waiting	Waiting for I/O or an event
Terminated	Execution finished

Transitions:

- **New to ready (admit):** the long-term scheduler accepts the process.
- **Ready to running (dispatch):** the short-term scheduler selects it.
- **Running to ready (time-out / interrupt):** its quantum ends or a higher-priority process arrives. This arrow exists only in preemptive systems.
- **Running to waiting:** the process requests I/O or waits for an event.
- **Waiting to ready:** the I/O or event completes.
- **Running to terminated (exit):** the process finishes.

Q8. Explain the followings - (a) Inter-process Communication (b) Mutual Exclusion and Race Condition (c) Critical Section. (2022)

(a) Inter-process Communication (IPC): it allows cooperating processes to exchange data and synchronise their actions. The two models are **shared memory** (a common region read and written directly, fast, needs synchronisation) and **message passing** (`send` and `receive` through the kernel, easier, good for distributed systems). Other mechanisms are pipes, FIFOs, sockets and signals.

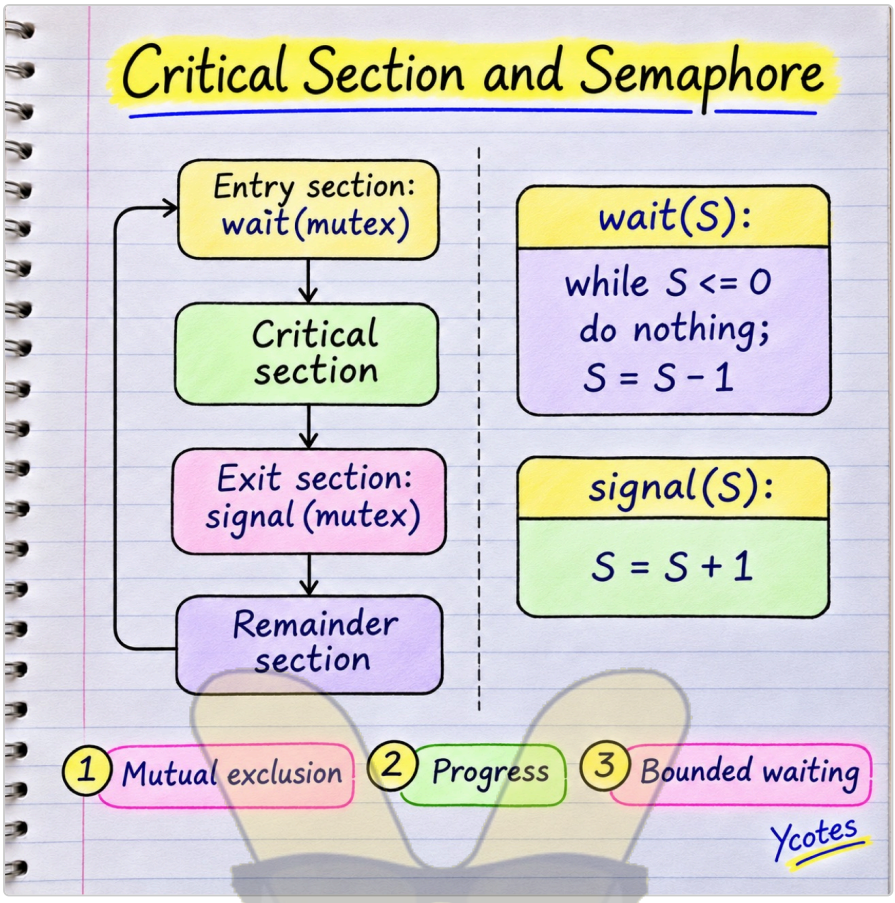
(b) Race condition and mutual exclusion:

A **race condition** occurs when several processes access and change shared data at the same time and the result depends on the order of execution. For example, with `counter = 5`, `counter++` by P1 and `counter--` by P2 may give 4 or 6 instead of 5 depending on how their load, change and store steps interleave.

Mutual exclusion means only one process may use the shared resource at a time. It prevents race conditions. It is achieved by locks, semaphores, monitors and Peterson's algorithm.

(c) Critical section: the part of a process code in which it accesses shared resources. Structure:

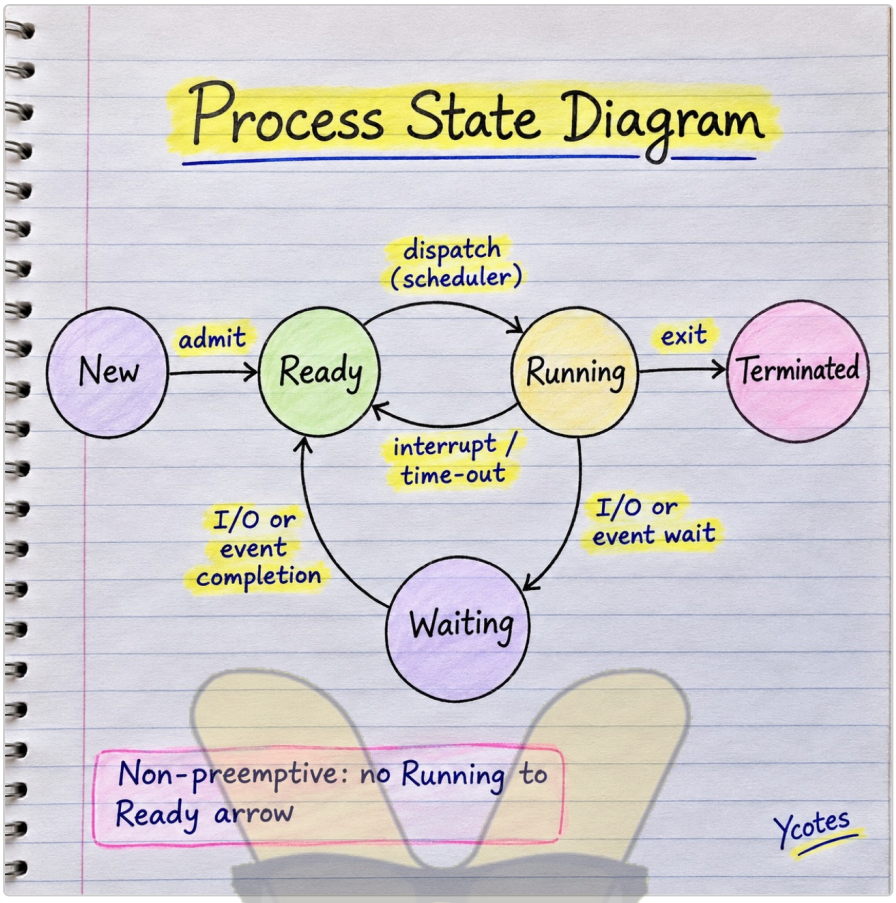
```
do {
    entry section;
    critical section;
    exit section;
    remainder section;
} while (true);
```



A solution must satisfy three requirements: **mutual exclusion**, **progress** (only processes wanting to enter decide who enters next, without indefinite postponement) and **bounded waiting** (a limit on how long a process waits).

Q9. Define process. How many different states a process has? Explain when a process changes the state using proper diagram. (2023)

A **process** is a program in execution, made of code, data, stack, heap and program counter. A process has **five states**: new, ready, running, waiting and terminated.



State changes:

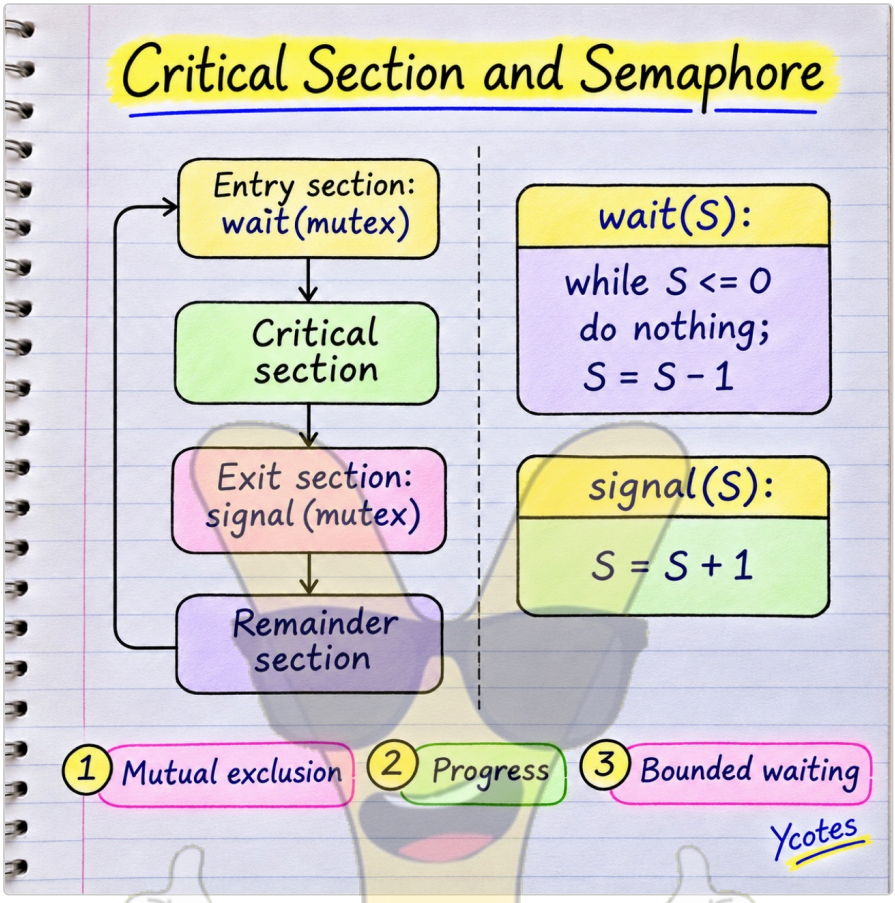
Transition	When it happens
New to Ready	Admitted by the long-term scheduler after creation
Ready to Running	Dispatcher gives the CPU (scheduler dispatch)
Running to Ready	Time quantum expires or a higher-priority process arrives (interrupt)
Running to Waiting	Process requests I/O or waits for an event
Waiting to Ready	The I/O or event it waited for is completed
Running to Terminated	Process executes its last instruction or calls `exit`

Only one process can be in the running state on one CPU at any instant, but many can be ready or waiting. The OS keeps the PCB of each process and moves it between the ready queue and the device queues as the state changes.

Q10. Explain Reader-writer problem and discuss the solution with proper

diagram. (2023)

The **Readers-Writers problem:** a shared database is used by many **readers** (only read) and **writers** (read and write). Many readers may read together, but a writer needs **exclusive access**: no reader and no other writer at the same time.



Shared data: `mutex = 1` (protects `readcount`), `wrt = 1` (exclusive access), `readcount = 0`.

```
Writer:                          Reader:
wait(wrt);                       wait(mutex);
  write;                           readcount++;
signal(wrt);                       if (readcount == 1) wait(wrt);
                                 signal(mutex);
                                   read;
                                 wait(mutex);
                                   readcount--;
                                   if (readcount == 0) signal(wrt);
                                 signal(mutex);
```

Explanation:

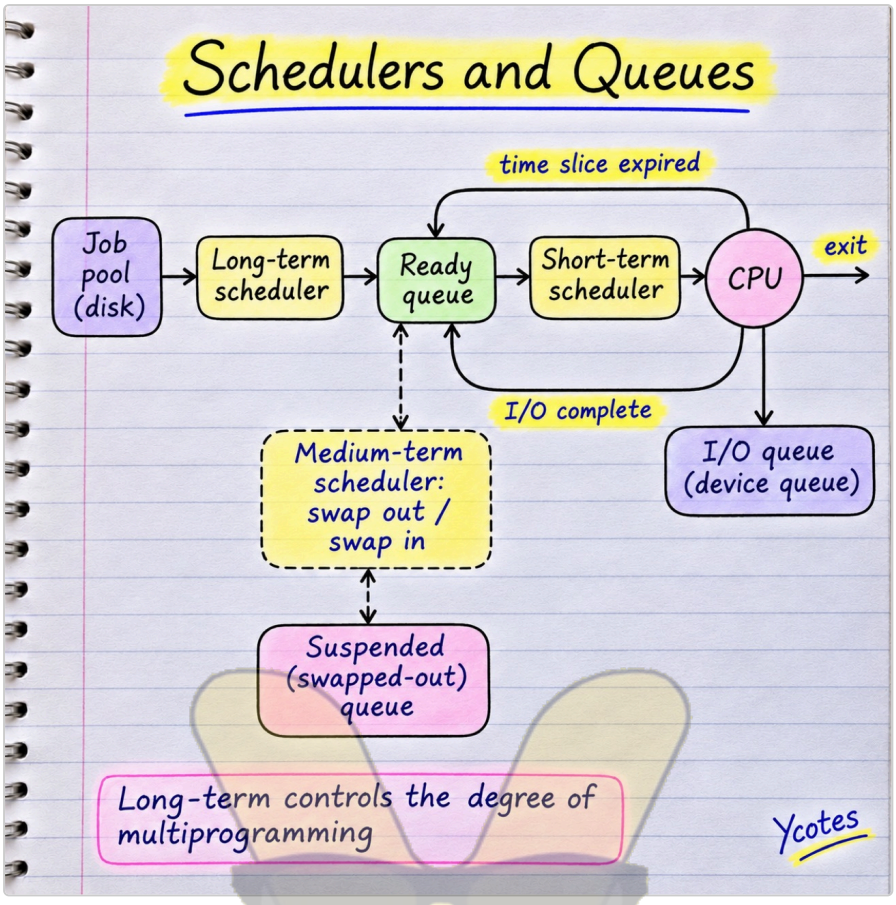
- The **first reader locks out writers** by executing `wait(wrt)`.
- Other readers enter without waiting because `wrt` is already held for readers.

- The **last reader** releases `wrt` so a waiting writer can enter.
- A writer simply takes `wrt`, so it excludes everybody.

Drawback: this first solution gives readers priority, so writers may **starve** while readers keep arriving. A second solution gives writers priority, with the opposite risk for readers.

Q11. Write the differences among short term, medium term and long term scheduling. (2023)

	Long-term (job) scheduler	**Short-term (CPU) scheduler**	**Medium-term scheduler**
Function	Selects jobs from the job pool and loads them into memory	Selects a ready process and allocates the CPU	Swaps processes out of and back into memory
Frequency	Least frequent (seconds to minutes)	Most frequent (milliseconds)	Intermediate
Controls	**Degree of multiprogramming**	Which process runs next	Memory pressure and process mix
Speed	Slow	Very fast	Medium
Time-sharing systems	Often absent	Present	Present
Transition	New to ready	Ready to running	Swapped-out to ready and back



The long-term scheduler should choose a good mix of I/O-bound and CPU-bound jobs. The short-term scheduler must be fast because it runs very often. The medium-term scheduler reduces the degree of multiprogramming by swapping out a process and later swapping it in, which also helps to avoid thrashing.

Q12. Differentiate among multi programming, multi-processing and

multi-tasking. (2024)

Multiprogramming	Multiprocessing	Multitasking
Several programs kept in memory at once, on one CPU	Two or more CPUs in one system sharing memory	Many tasks run in a time-shared way by switching the CPU quickly
CPU switches when a job waits for I/O	Processes run truly in parallel	Switching is based on a time slice, so the user sees quick response
Goal: maximise CPU utilisation	Goal: higher throughput and reliability	Goal: good interactive response
Needs memory management and CPU scheduling	Needs multiple CPUs and symmetric or asymmetric processing	Built on multiprogramming plus timer interrupts
Example: early mainframe batch systems	Example: server with 8 cores	Example: Windows, Linux running a browser and editor together

Key line: multiprogramming means many programs and one CPU, while multiprocessing means many CPUs.

Q13. What is a race condition? Illustrate with an example why presence of race condition is considered as bad design. (2024)

A **race condition** occurs when two or more processes access and modify shared data concurrently and the final result depends on the particular order in which their instructions execute.

Example: shared variable `counter = 5`. P1 executes `counter++` and P2 executes `counter--`. Each is three machine steps.

Step	P1	P2	counter
1	`R1 = counter` (5)		5
2	`R1 = R1 + 1` (6)		5
3		`R2 = counter` (5)	5
4		`R2 = R2 - 1` (4)	5
5	`counter = R1`		6
6		`counter = R2`	**4**

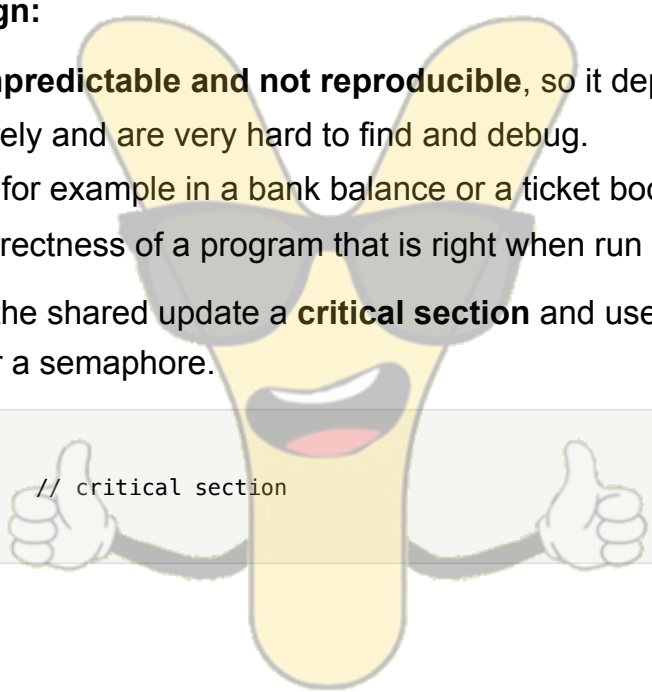
The correct final value is 5, but the interleaving gives **4** (or 6 if the last two steps are swapped).

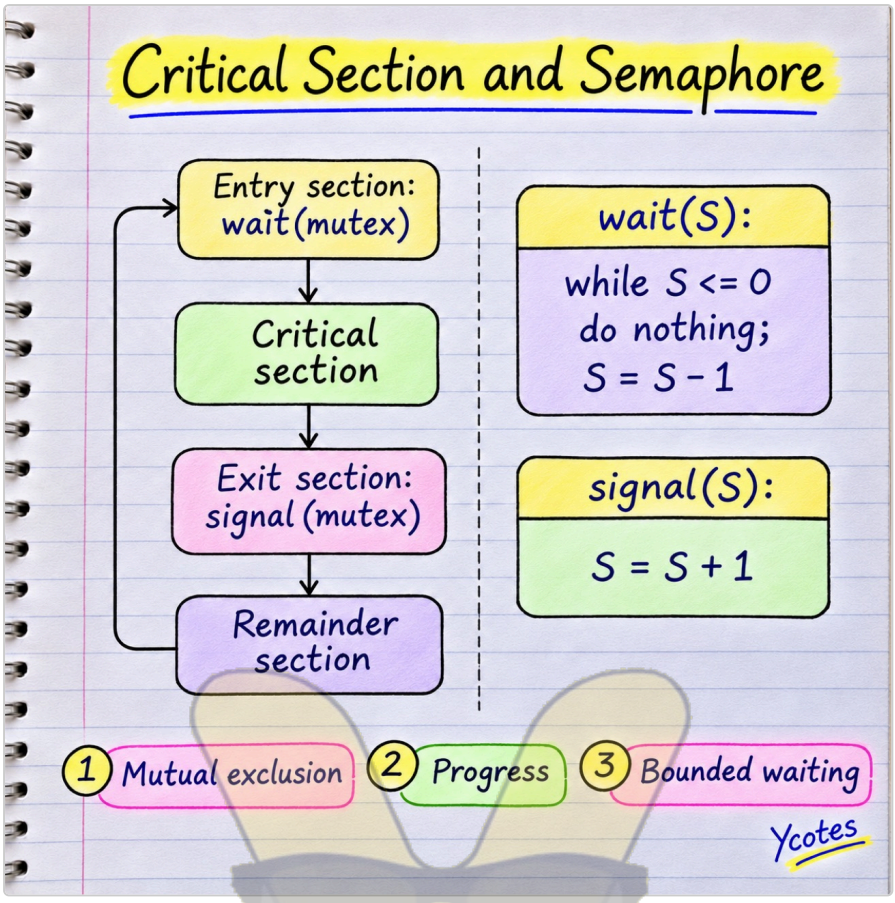
Why it is bad design:

- The output is **unpredictable and not reproducible**, so it depends on timing.
- Bugs appear rarely and are very hard to find and debug.
- It corrupts data, for example in a bank balance or a ticket booking system.
- It breaks the correctness of a program that is right when run alone.

Prevention: make the shared update a **critical section** and use mutual exclusion with a mutex lock or a semaphore.

```
wait(mutex);
counter++;          // critical section
signal(mutex);
```





Q14. What happens when a high-priority thread enters a system with a currently running low-priority thread? Explain with priority inversion example. (2025)

Situation: a low-priority thread L is running and a high-priority thread H becomes ready.

What happens: in a preemptive priority system H **preempts** L at once and runs. L goes back to the ready queue. H runs until it finishes, blocks, or waits for a resource.

Priority inversion: a problem arises when H needs a resource (lock) that L already holds. H blocks, and L runs again to release the lock. If a medium-priority thread M, which does not need the lock, becomes ready now, it preempts L. H then waits for M although H has a higher priority than M.

Time	Event	Running
t1	L starts and locks resource R	L
t2	H arrives, preempts L, requests R and blocks	L (H waiting)
t3	M arrives and preempts L	M
t4	M finishes	L
t5	L releases R	H

So H effectively waits for M, and the priority order is inverted. Here the unbounded delay happens for H. This bug made the Mars Pathfinder reset repeatedly.

Solutions:

- **Priority inheritance:** L temporarily gets the priority of H while it holds the lock, so M cannot preempt it.
- **Priority ceiling:** the lock has a ceiling equal to the highest priority of any user, and a holder runs at that priority.

Q15. Simulate a scenario where two processes enter the critical section at the same time. How can mutual exclusion solve this. (2025)

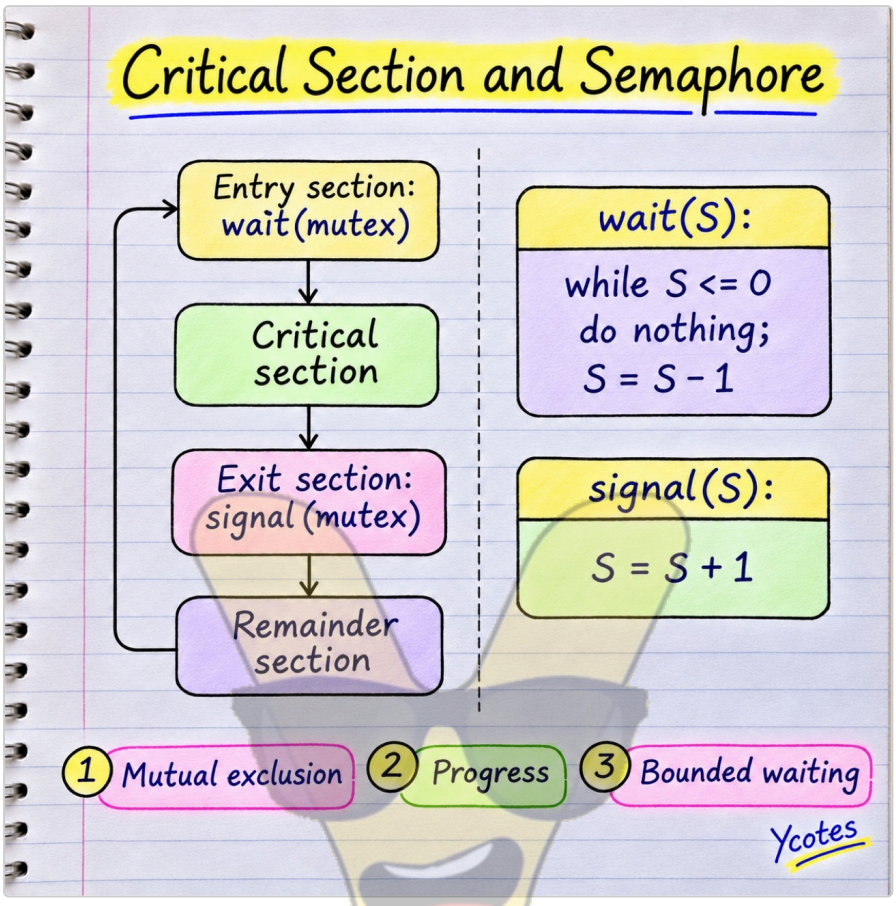
Scenario: two processes P1 and P2 share `balance = 100`. Both execute `balance = balance + 50` inside their critical section at the same time, without any protection.

Step	P1	P2	balance in memory
1	reads 100		100
2		reads 100	100
3	computes 150		100
4		computes 150	100
5	writes 150		150
6		writes 150	**150**

The correct answer is 200 but one update is lost. This happens because both are in the critical section at once.

Mutual exclusion solution with a semaphore (`mutex = 1`):

```
wait(mutex);                // entry section
balance = balance + 50;     // critical section
signal(mutex);              // exit section
```



Step	P1	P2
1	`wait(mutex)` succeeds, mutex = 0	
2	enters critical section	`wait(mutex)` blocks, mutex = 0
3	writes 150, `signal(mutex)`	still blocked
4		wakes up, reads 150, writes **200**

Only one process is in the critical section at a time, so the result is always 200. Mutual exclusion, progress and bounded waiting are all satisfied.

PART–C Answers (≈500 Words Each)

Q1. Calculate the average waiting time & turnaround time of the system using preemptive SJF and Round Robin (tq = 2). Processes (Process Id, Arrival time, Burst time): P1 0 3; P2 1 5; P3 2 2; P4 3 4; P5 4 1; P6 5 3. (2019)

Method: TAT = CT - AT and WT = TAT - BT. Average = sum / number of processes.

(a) Preemptive SJF (SRTF)

Time	0-3	3-5	5-6	6-9	9-13	13-18
Process	P1	P3	P5	P6	P4	P2

Process	AT	BT	CT	TAT = CT - AT	WT = TAT - BT
P1	0	3	3	3	0
P2	1	5	18	17	12
P3	2	2	5	3	1
P4	3	4	13	10	6
P5	4	1	6	2	1
P6	5	3	9	4	1

Average WT = (0 + 12 + 1 + 6 + 1 + 1) / 6 = 21/6 = **3.5 ms**; Average TAT = (3 + 17 + 3 + 10 + 2 + 4) / 6 = 39/6 = **6.5 ms**

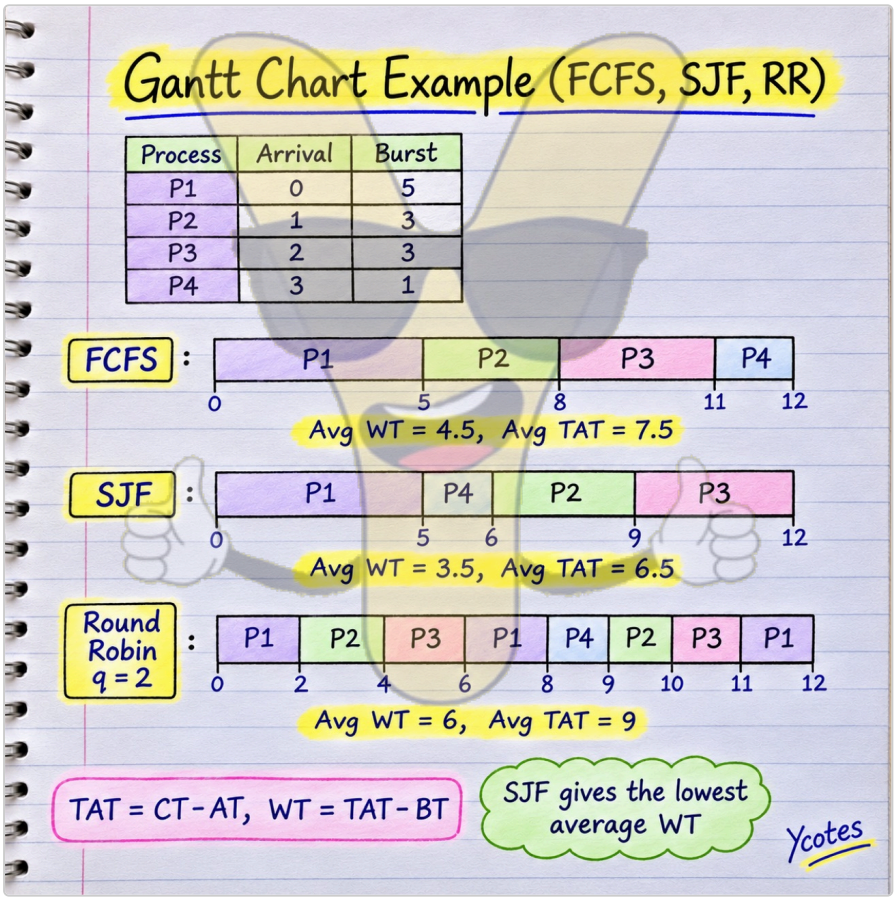
At every arrival the burst of the new process is compared with the remaining time of the running process. Here no process is actually preempted: every arrival has a burst not smaller than the remaining time of the running process (at t = 4, P5 = 1 ties with P3 remaining = 1, so P3 continues).

(b) Round Robin, quantum = 2 (a process arriving during a slice joins the queue before the preempted process)

Time	0-2	2-4	4-6	6-7	7-9	9-10	10-12	12-14	14-16	16-17	17-18
Process	P1	P2	P3	P1	P4	P5	P2	P6	P4	P2	P6

Process	AT	BT	CT	TAT = CT - AT	WT = TAT - BT
P1	0	3	7	7	4
P2	1	5	17	16	11
P3	2	2	6	4	2
P4	3	4	16	13	9
P5	4	1	10	6	5
P6	5	3	18	13	10

Average WT = (4 + 11 + 2 + 9 + 5 + 10) / 6 = 41/6 = **6.83 ms**; Average TAT = (7 + 16 + 4 + 13 + 6 + 13) / 6 = 59/6 = **9.83 ms**



Conclusion: SRTF (average WT 3.5 ms, TAT 6.5 ms) is better than Round Robin (average WT 6.83 ms, TAT 9.83 ms), but RR gives a better response time.

Q2. Compute average turnaround time and average waiting time for (a) FIFO (b) SJF (c) Round Robin. Process, Burst time, Arrival time: P1 4 2;

P2 2 1; P3 5 1; P4 3 3. (2021)

Assumption: the table lists Process, Burst time, Arrival time, so P1 has BT 4 and AT 2, and so on. The CPU is idle from 0 to 1. The paper gives no quantum for RR, so q = 2 ms is assumed.

(a) FIFO (FCFS)

Time	0-1	1-3	3-8	8-12	12-15
Process	Idle	P2	P3	P1	P4

Process	AT	BT	CT	TAT = CT - AT	WT = TAT - BT
P1	2	4	12	10	6
P2	1	2	3	2	0
P3	1	5	8	7	2
P4	3	3	15	12	9

Average WT = (6 + 0 + 2 + 9) / 4 = 17/4 = **4.25 ms**; Average TAT = (10 + 2 + 7 + 12) / 4 = 31/4 = **7.75 ms**

(b) SJF (non-preemptive)

Time	0-1	1-3	3-6	6-10	10-15
Process	Idle	P2	P4	P1	P3

Process	AT	BT	CT	TAT = CT - AT	WT = TAT - BT
P1	2	4	10	8	4
P2	1	2	3	2	0
P3	1	5	15	14	9
P4	3	3	6	3	0

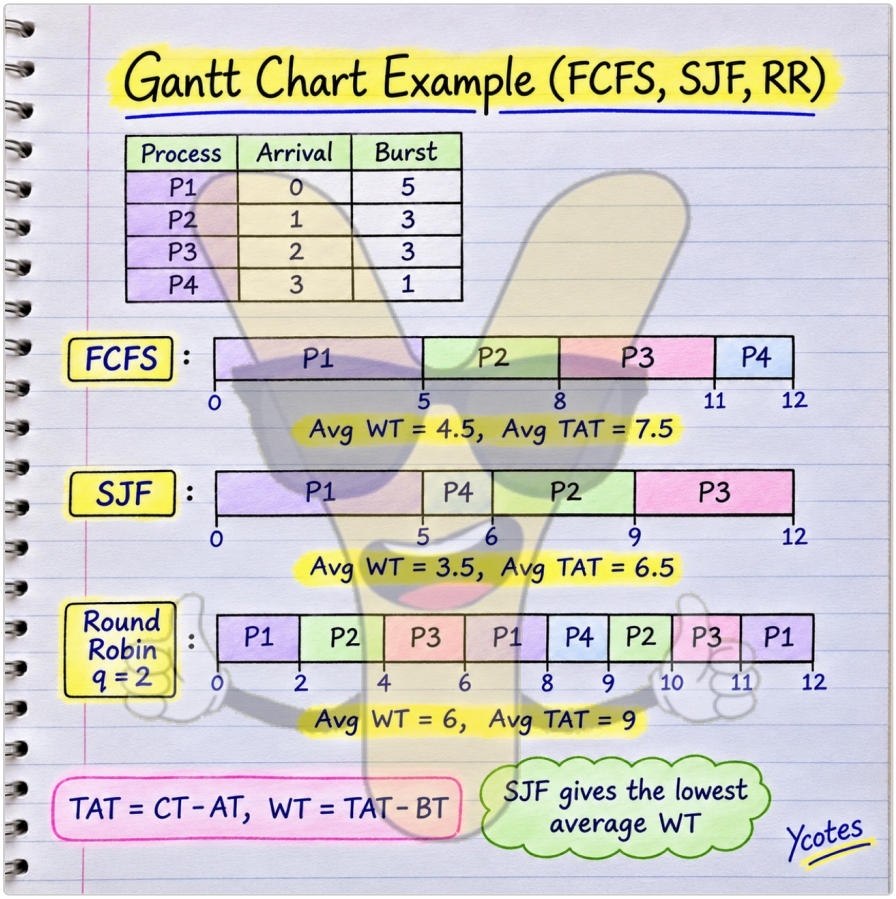
Average WT = (4 + 0 + 9 + 0) / 4 = 13/4 = **3.25 ms**; Average TAT = (8 + 2 + 14 + 3) / 4 = 27/4 = **6.75 ms**

(c) Round Robin (quantum 2 ms)

Time	1-3	3-5	5-7	7-9	9-11	11-13	13-14	14-15
Process	P2	P3	P1	P4	P3	P1	P4	P3

Process	AT	BT	CT	TAT = CT - AT	WT = TAT - BT
P1	2	4	13	11	7
P2	1	2	3	2	0
P3	1	5	15	14	9
P4	3	3	14	11	8

Average WT = (7 + 0 + 9 + 8) / 4 = 24/4 = **6 ms**; Average TAT = (11 + 2 + 14 + 11) / 4 = 38/4 = **9.5 ms**



Algorithm	Average WT (ms)	Average TAT (ms)
FIFO	4.25	7.75
SJF	3.25	6.75
RR (q = 2)	6	9.5

SJF gives the minimum average waiting time.

Q3. For the following set of process, find the average waiting time and turn around time using Gantt chart for (a) SJF (b) Priority scheduling. Process, Burst time (ms), Priority: P1 5 5; P2 3 4; P3 8 3; P4 2 1; P5 1 2. (2022)

Assumption: all processes arrive at time 0, scheduling is non-preemptive and a smaller priority number means a higher priority.

(a) SJF

Time	**0-1**	**1-3**	**3-6**	**6-11**	**11-19**
Process	P5	P4	P2	P1	P3

Process	**AT**	**BT**	**CT**	**TAT = CT - AT**	**WT = TAT - BT**
P1	0	5	11	11	6
P2	0	3	6	6	3
P3	0	8	19	19	11
P4	0	2	3	3	1
P5	0	1	1	1	0

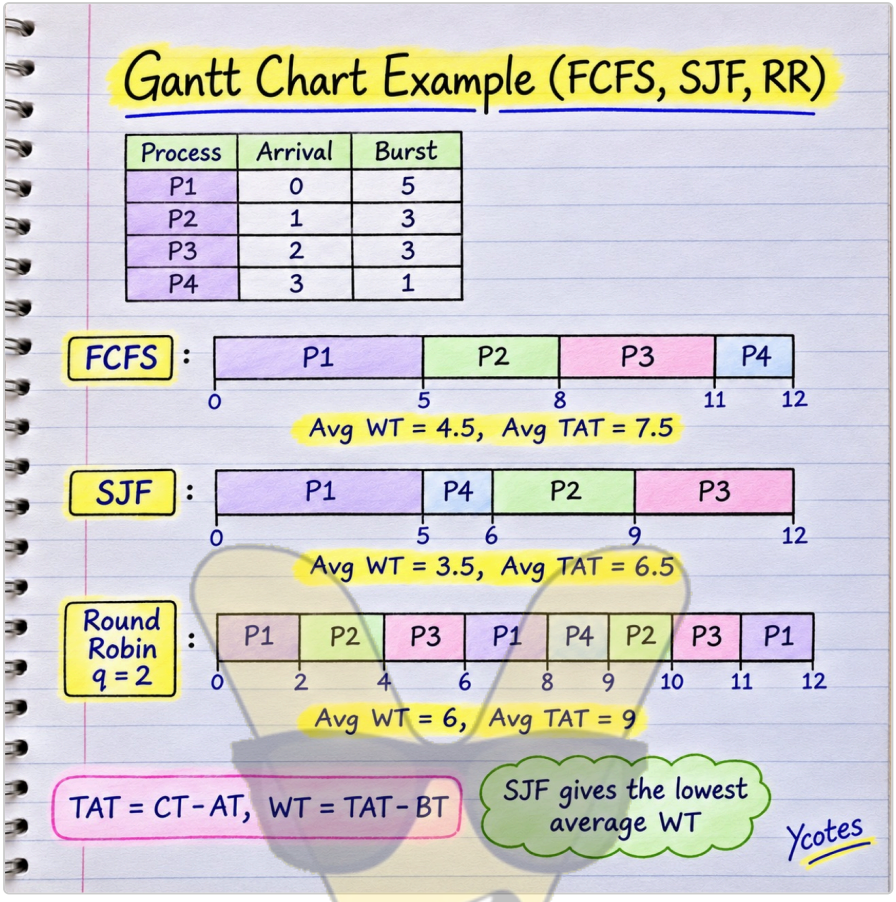
Average WT = (6 + 3 + 11 + 1 + 0) / 5 = 21/5 = **4.2 ms**; Average TAT = (11 + 6 + 19 + 3 + 1) / 5 = 40/5 = **8 ms**

(b) Priority scheduling

Time	**0-2**	**2-3**	**3-11**	**11-14**	**14-19**
Process	P4	P5	P3	P2	P1

Process	**AT**	**BT**	**Priority**	**CT**	**TAT = CT - AT**	**WT = TAT - BT**
P1	0	5	5	19	19	14
P2	0	3	4	14	14	11
P3	0	8	3	11	11	3
P4	0	2	1	2	2	0
P5	0	1	2	3	3	2

Average WT = (14 + 11 + 3 + 0 + 2) / 5 = 30/5 = **6 ms**; Average TAT = (19 + 14 + 11 + 2 + 3) / 5 = 49/5 = **9.8 ms**



Conclusion: SJF (average WT 4.2 ms, TAT 8 ms) is better than priority scheduling (average WT 6 ms, TAT 9.8 ms) because SJF is optimal for average waiting time.

Q4. Consider the following set of process, arrival time and CPU burst time in milliseconds. Calculate average waiting time and turn-around time using (i) FCFS (ii) SJF (iii) Round Robin (Quantum = 2). Draw Gantt chart for each. Process, Arrival, Burst: P1 0 5; P2 1 3; P3 2 3; P4 3 1. (2023)

Steps: draw the Gantt chart, read CT, TAT = CT - AT, WT = TAT - BT, then average.

(i) FCFS

Time	**0-5**	**5-8**	**8-11**	**11-12**
Process	P1	P2	P3	P4

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

Average WT = (0 + 4 + 6 + 8) / 4 = 18/4 = **4.5 ms**; Average TAT = (5 + 7 + 9 + 9) / 4 = 30/4 = **7.5 ms**

(ii) SJF (non-preemptive)

Time	0-5	5-6	6-9	9-12
Process	P1	P4	P2	P3

Process	AT	BT	CT	TAT = CT - AT	WT = TAT - BT
P1	0	5	5	5	0
P2	1	3	9	8	5
P3	2	3	12	10	7
P4	3	1	6	3	2

Average WT = (0 + 5 + 7 + 2) / 4 = 14/4 = **3.5 ms**; Average TAT = (5 + 8 + 10 + 3) / 4 = 26/4 = **6.5 ms**

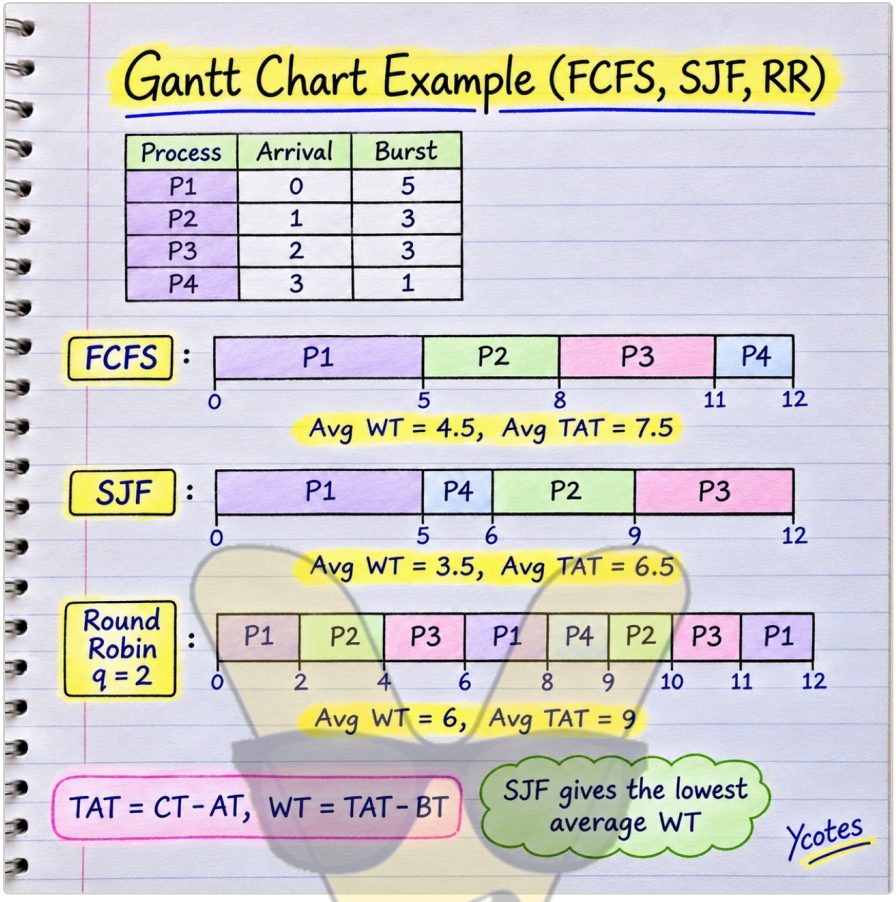
At t = 5 the ready processes are P2 (3), P3 (3) and P4 (1), so P4 is chosen first.

(iii) Round Robin, quantum = 2 (a process arriving during a slice joins the queue before the preempted process)

Time	0-2	2-4	4-6	6-8	8-9	9-10	10-11	11-12
Process	P1	P2	P3	P1	P4	P2	P3	P1

Process	AT	BT	CT	TAT = CT - AT	WT = TAT - BT
P1	0	5	12	12	7
P2	1	3	10	9	6
P3	2	3	11	9	6
P4	3	1	9	6	5

Average WT = (7 + 6 + 6 + 5) / 4 = 24/4 = **6 ms**; Average TAT = (12 + 9 + 9 + 6) / 4 = 36/4 = **9 ms**



Algorithm	Average WT (ms)	Average TAT (ms)
FCFS	4.5	7.5
SJF	3.5	6.5
RR (q = 2)	6	9

SJF gives the lowest average waiting time.

Q5. Consider four processes with CPU burst time in ms - Process, Burst time, Arrival time: P0 15 0.0; P1 20 1.0; P2 3 2.0; P3 7 2.0. Consider SRTF and Round Robin (Quantum = 5ms). Illustrate the Gantt chart. Which algorithm will give the minimum average waiting time. (Feb 2024)

Total burst time = 15 + 20 + 3 + 7 = 45 ms, so the last completion time is 45 ms in both charts.

SRTF (preemptive SJF) (P0 is preempted at t = 2 when P2 arrives with burst 3)

Time	0-2	2-5	5-12	12-25	25-45
Process	P0	P2	P3	P0	P1

Process	AT	BT	CT	TAT = CT - AT	WT = TAT - BT
P0	0	15	25	25	10
P1	1	20	45	44	24
P2	2	3	5	3	0
P3	2	7	12	10	3

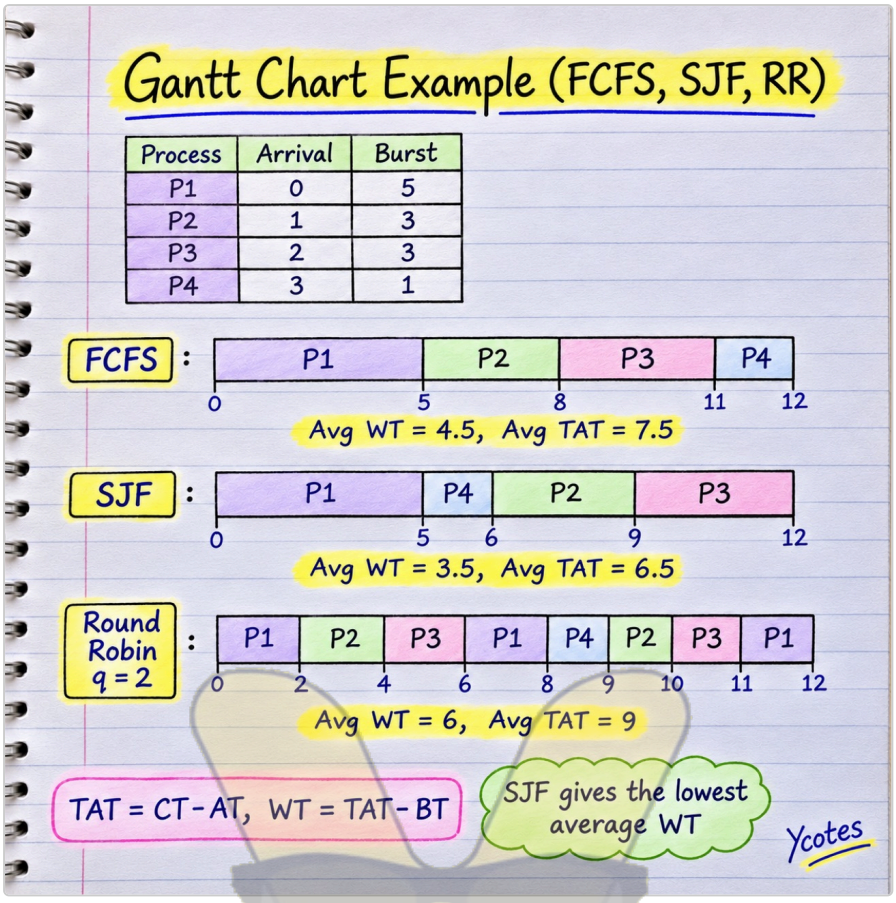
Average WT = (10 + 24 + 0 + 3) / 4 = 37/4 = **9.25 ms**; Average TAT = (25 + 44 + 3 + 10) / 4 = 82/4 = **20.5 ms**

Round Robin, quantum = 5 ms (P2 and P3 arrive at t = 2, before the first slice of P0 ends)

Time	0-5	5-10	10-13	13-18	18-23	23-28	28-30	30-35	35-45
Process	P0	P1	P2	P3	P0	P1	P3	P0	P1

Process	AT	BT	CT	TAT = CT - AT	WT = TAT - BT
P0	0	15	35	35	20
P1	1	20	45	44	24
P2	2	3	13	11	8
P3	2	7	30	28	21

Average WT = (20 + 24 + 8 + 21) / 4 = 73/4 = **18.25 ms**; Average TAT = (35 + 44 + 11 + 28) / 4 = 118/4 = **29.5 ms**

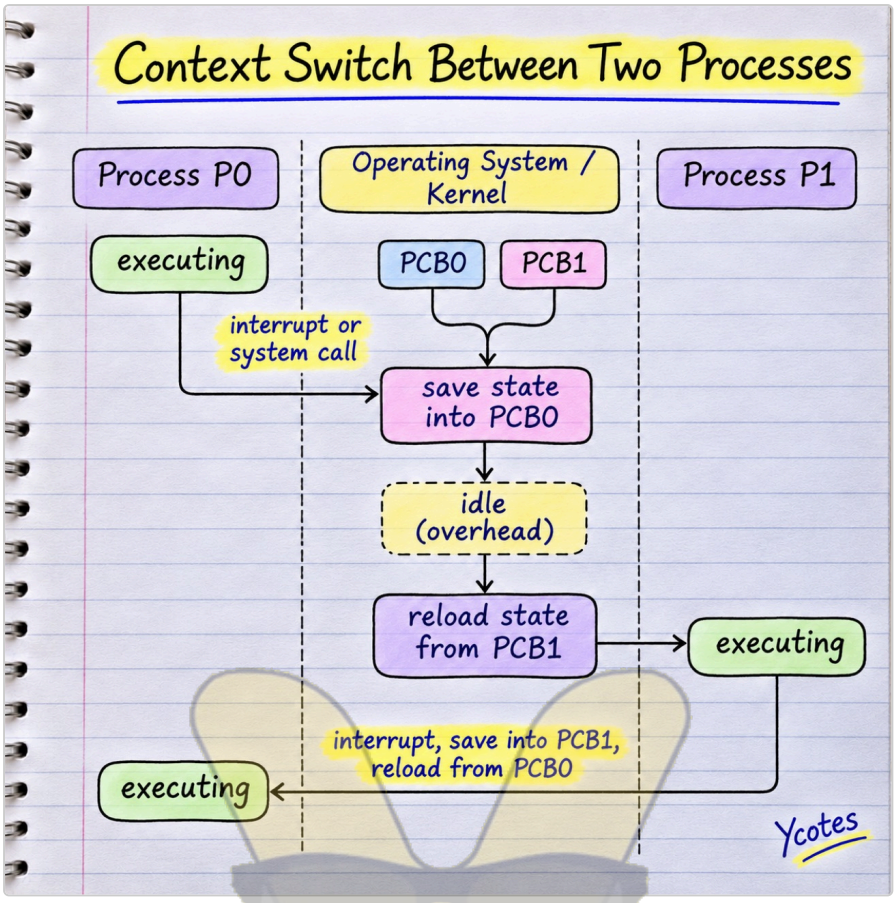


Algorithm	Average WT (ms)	Average TAT (ms)
SRTF	9.25	20.5
RR (q = 5)	18.25	29.5

Answer: SRTF gives the minimum average waiting time (9.25 ms against 18.25 ms).

Q6. (a) Describe the actions taken by Kernel to context switch between processes. (b) For a given set of processes calculate the average wait time and average turn around time using FCFS, SJF and RR. Process, Burst, Priority: P1 8 4; P2 6 1; P3 1 2; P4 9 2; P5 3 3. (2024)

(a) Actions taken by the kernel to context switch from process P0 to P1:



1. An interrupt (timer or I/O) or a system call occurs and the CPU enters kernel mode.
2. The kernel saves the program counter, CPU registers and state of P0 into **PCB0**.
3. It changes the state of P0 to ready or waiting and puts it in the proper queue.
4. The short-term scheduler selects P1 from the ready queue.
5. The kernel loads the program counter, registers and memory-management information (page table) of P1 from **PCB1**.
6. The state of P1 is set to running, the CPU returns to user mode and P1 resumes. The dispatcher does the last steps.

A context switch is pure overhead, because no useful work is done during it. Its cost depends on the number of registers, the PCB size and the memory hardware (TLB flush).

(b) *Assumption: all processes arrive at time 0. The priority column is not needed for FCFS, SJF and RR. The paper gives no quantum, so q = 4 ms is assumed.*

FCFS

Time	0-8	8-14	14-15	15-24	24-27
Process	P1	P2	P3	P4	P5

Process	AT	BT	CT	TAT = CT - AT	WT = TAT - BT
P1	0	8	8	8	0
P2	0	6	14	14	8
P3	0	1	15	15	14
P4	0	9	24	24	15
P5	0	3	27	27	24

Average WT = (0 + 8 + 14 + 15 + 24) / 5 = 61/5 = **12.2 ms**; Average TAT = (8 + 14 + 15 + 24 + 27) / 5 = 88/5 = **17.6 ms**

SJF

Time	0-1	1-4	4-10	10-18	18-27
Process	P3	P5	P2	P1	P4

Process	AT	BT	CT	TAT = CT - AT	WT = TAT - BT
P1	0	8	18	18	10
P2	0	6	10	10	4
P3	0	1	1	1	0
P4	0	9	27	27	18
P5	0	3	4	4	1

Average WT = (10 + 4 + 0 + 18 + 1) / 5 = 33/5 = **6.6 ms**; Average TAT = (18 + 10 + 1 + 27 + 4) / 5 = 60/5 = **12 ms**

Round Robin, quantum = 4 ms

Time	0-4	4-8	8-9	9-13	13-16	16-20	20-22	22-27
Process	P1	P2	P3	P4	P5	P1	P2	P4

Process	AT	BT	CT	TAT = CT - AT	WT = TAT - BT
P1	0	8	20	20	12
P2	0	6	22	22	16
P3	0	1	9	9	8
P4	0	9	27	27	18
P5	0	3	16	16	13

Average WT = (12 + 16 + 8 + 18 + 13) / 5 = 67/5 = **13.4 ms**; Average TAT = (20 + 22 + 9 + 27 + 16) / 5 = 94/5 = **18.8 ms**

Algorithm	Average WT (ms)	Average TAT (ms)
FCFS	12.2	17.6
SJF	6.6	12
RR (q = 4)	13.4	18.8

SJF gives the minimum average waiting and turnaround time.

Q7. Five processes P1 to P5 arrive at time 0 with burst times P1 = 6 ms, P2 = 8 ms, P3 = 7 ms, P4 = 3 ms, P5 = 4 ms. Using Shortest Job First (non-preemptive) calculate (a) average waiting time (b) average turnaround time. (2025)

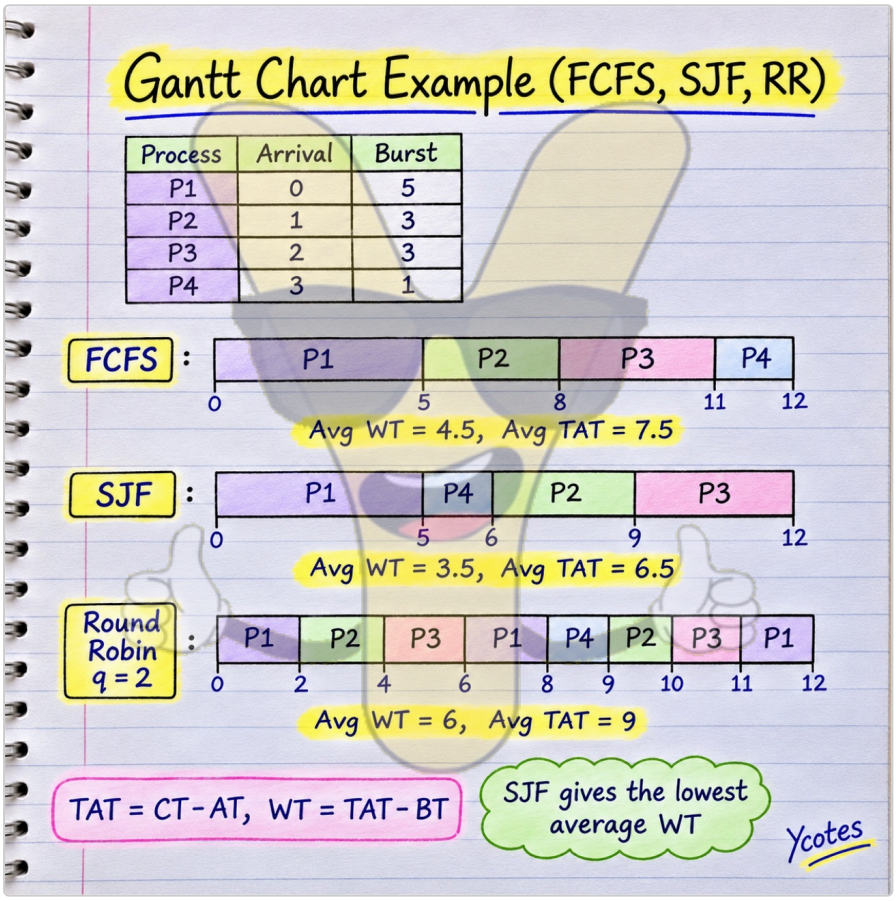
Assumption: all processes arrive at time 0, so the shortest burst runs first.

Non-preemptive SJF

Time	0-3	3-7	7-13	13-20	20-28
Process	P4	P5	P1	P3	P2

Process	AT	BT	CT	TAT = CT - AT	WT = TAT - BT
P1	0	6	13	13	7
P2	0	8	28	28	20
P3	0	7	20	20	13
P4	0	3	3	3	0
P5	0	4	7	7	3

Average WT = (7 + 20 + 13 + 0 + 3) / 5 = 43/5 = **8.6 ms**; Average TAT = (13 + 28 + 20 + 3 + 7) / 5 = 71/5 = **14.2 ms**



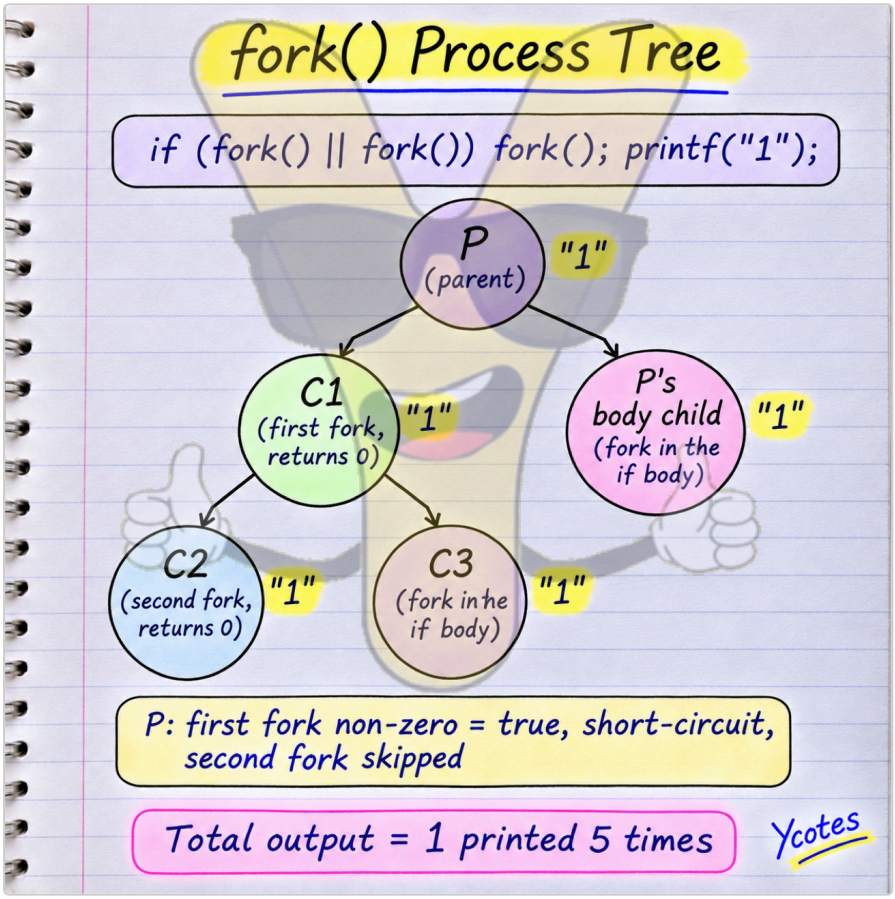
(a) Average waiting time = 8.6 ms.
(b) Average turnaround time = 14.2 ms.

Q8. What is Fork system call? What will be the output of the following code and justify the output? #include<stdio.h> #include<unistd.h> int

main() { if (fork() || fork()) fork(); printf("1"); return 0; } (2024)

Fork system call: `fork()` creates a new **child process** that is an exact copy of the parent. Both continue from the instruction after `fork()`. It returns **0 in the child**, the **child's PID (non-zero) in the parent**, and -1 on failure. A program with n unconditional forks creates 2^n processes in total.

```
#include<stdio.h>
#include<unistd.h>
int main() {
    if (fork() || fork())
        fork();
    printf("1");
    return 0;
}
```



Solution: the operator `||` short-circuits, so the second operand is evaluated only if the first one is 0.

1. **Parent P:** the first `fork()` returns a non-zero value (true), so the second `fork()` is **not** executed. The condition is true, so P executes the body `fork()` and creates child **C4**.

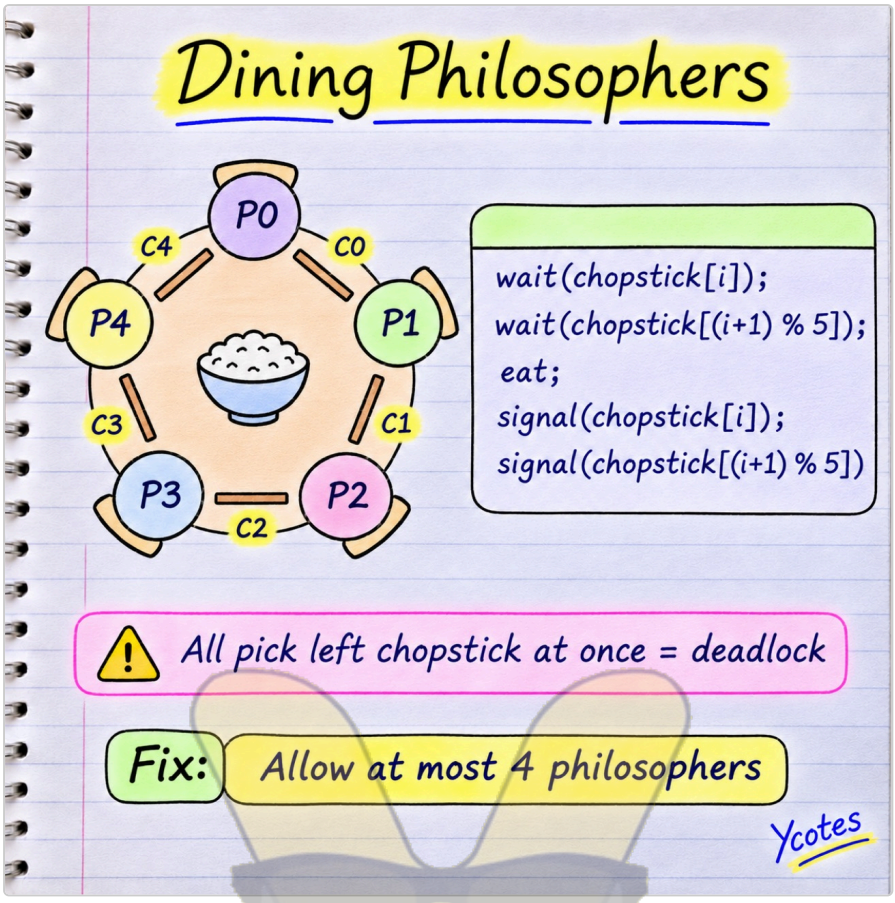
2. **Child C1** (created by the first fork): its `fork()` returned 0 (false), so it executes the second `fork()`, which creates **C2**. In C1 that call returns non-zero, so the condition is true and C1 executes the body `fork()`, creating **C3**.
3. **Child C2** (created by the second fork): the second `fork()` returned 0, so the condition is false and C2 skips the body.
4. C3 and C4 are created inside the body and just continue.

Process	Created by	Condition result	Body fork
P	main	true	creates C4
C1	P, first fork	true	creates C3
C2	C1, second fork	false	skipped
C3	C1, body	-	-
C4	P, body	-	-

Every one of the 5 processes reaches `printf("1")` once. **Output: 11111** (the digit 1 printed five times; the order of the processes is not fixed).

Q9. What is Dining Philosophers Problem? Explain the solution of this problem by using a suitable example. (2021, Feb 2024)

Dining Philosophers Problem: five philosophers sit around a round table. There is a bowl of rice in the centre and **five chopsticks**, one between each pair of neighbours. A philosopher alternates between thinking and eating. To eat, a philosopher needs **both** the left and the right chopstick, and may pick up only one at a time. When finished, both are put down.



It is a classic model of allocating several resources among several processes without **deadlock** and **starvation**.

Basic semaphore solution: one semaphore for each chopstick, `semaphore chopstick[5]`, all initialised to 1.

```
do {
    wait(chopstick[i]);              // left chopstick
    wait(chopstick[(i+1) % 5]);      // right chopstick
    eat;
    signal(chopstick[i]);
    signal(chopstick[(i+1) % 5]);
    think;
} while (1);
```

This guarantees that no two neighbours eat together, because a chopstick is held by only one philosopher.

Flaw (example): if all five philosophers become hungry together and each picks up the left chopstick, then each holds one chopstick and waits for the right one forever. All five are blocked, which is **deadlock** (hold and wait plus circular wait).

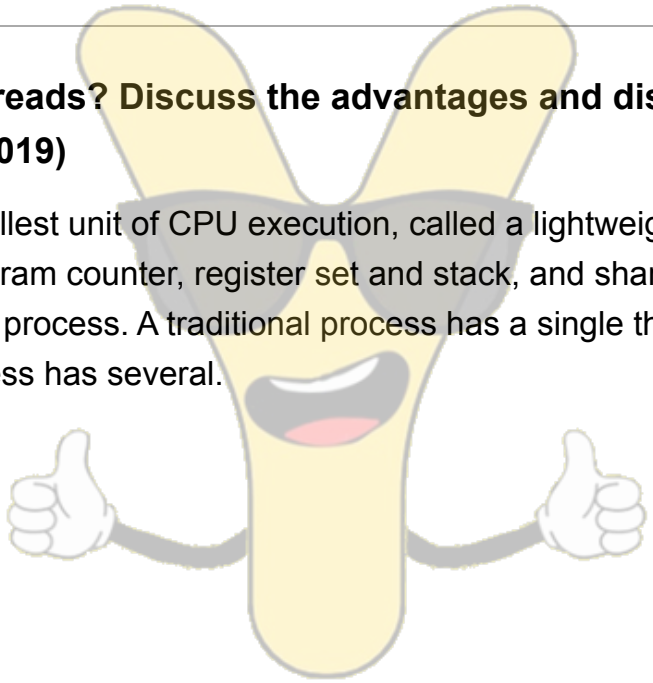
Remedies:

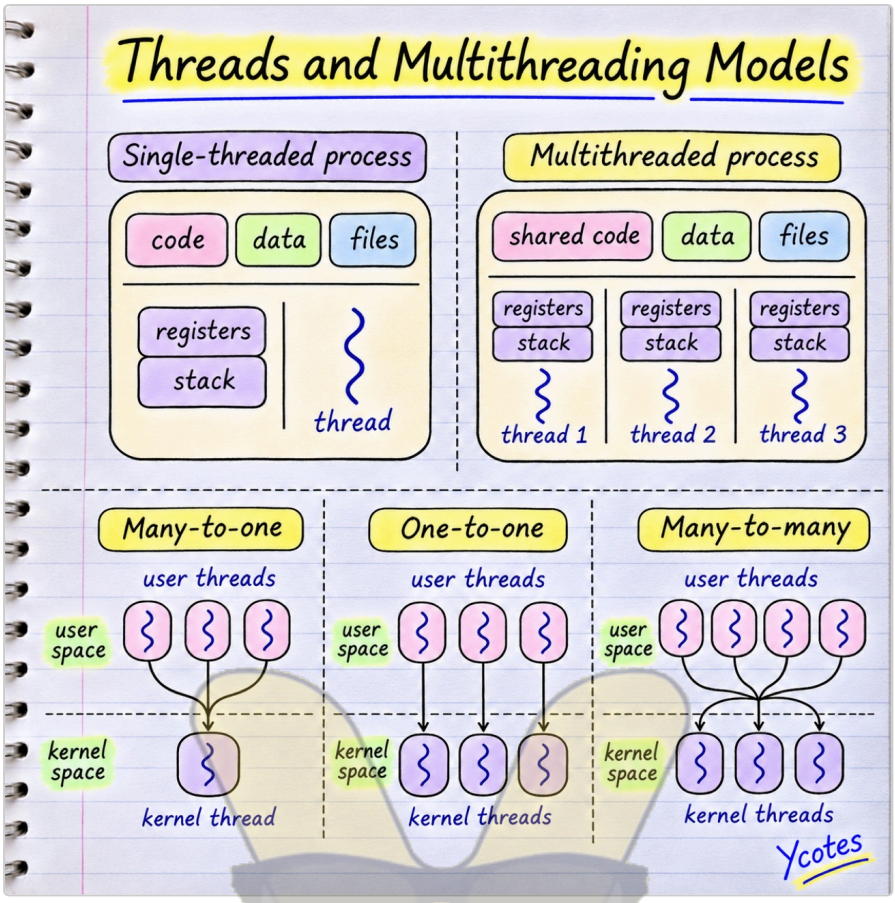
1. Allow at most **four** philosophers at the table at the same time (a counting semaphore initialised to 4).
2. Allow a philosopher to pick up chopsticks only if **both** are available, checked inside a critical section.
3. Use an **asymmetric** solution: odd-numbered philosophers pick the left chopstick first, even-numbered ones the right first.
4. Use a **monitor** with states thinking, hungry and eating; a philosopher eats only when neither neighbour is eating.

Example with remedy 1: with `room = 4`, at most four philosophers compete for five chopsticks, so at least one gets two chopsticks, eats and releases them, and the others proceed. Deadlock is avoided, although starvation of a particular philosopher is still possible unless the queue is fair.

Q10. What are threads? Discuss the advantages and disadvantages of using threads. (2019)

A **thread** is the smallest unit of CPU execution, called a lightweight process. It has its own thread ID, program counter, register set and stack, and shares the code, data and open files of its process. A traditional process has a single thread; a multithreaded process has several.





Multithreading models:

- **Many-to-one:** many user threads map to one kernel thread. One blocking call blocks all threads and there is no parallelism.
- **One-to-one:** each user thread has a kernel thread. It gives true parallelism but has more overhead (Linux, Windows).
- **Many-to-many:** many user threads are multiplexed over a smaller or equal number of kernel threads.

Advantages of threads:

1. **Responsiveness:** a program continues even if part of it is blocked or doing a long operation.
2. **Resource sharing:** threads share memory and resources of their process by default, so no IPC is needed.
3. **Economy:** creation and context switching of threads cost much less than for processes.
4. **Scalability:** threads of one process run in parallel on different processors or cores.
5. Better utilisation of multiprocessor architectures.

Disadvantages of threads:

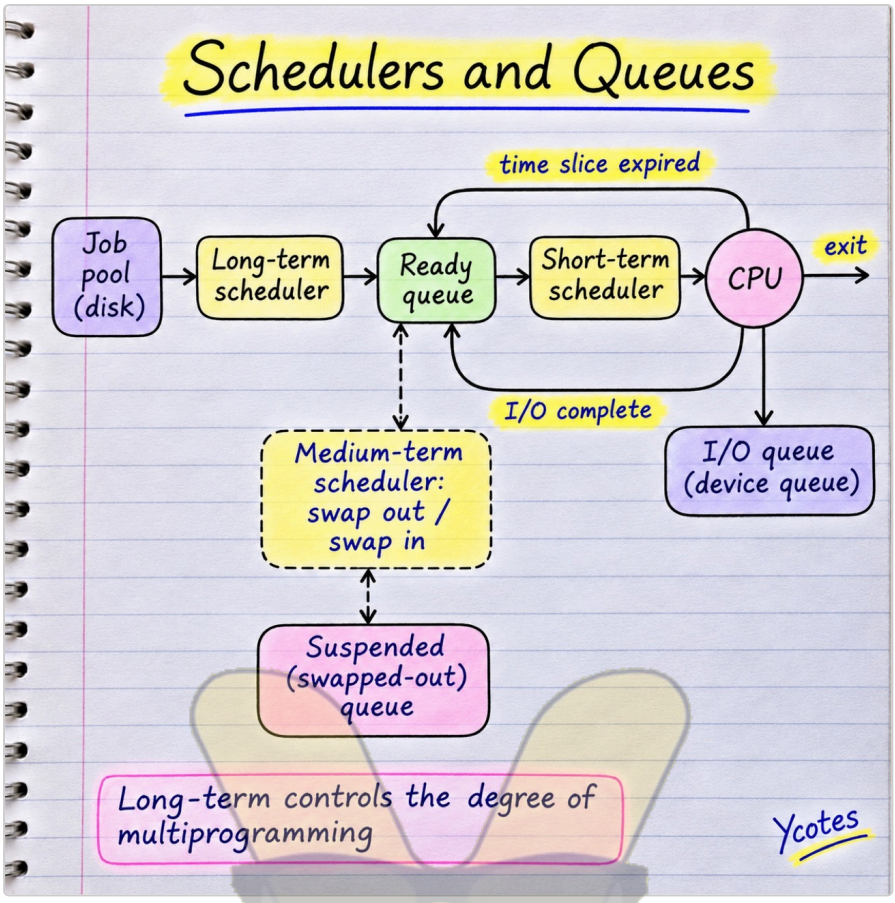
1. **No isolation:** a faulty thread can corrupt data or crash the whole process.
2. **Synchronisation needed:** shared data leads to race conditions and deadlocks if locks are not used correctly.
3. **Difficult to debug and test**, because errors depend on timing.
4. In many-to-one models one blocking system call blocks the whole process.
5. A security problem: all threads can access all the data of the process.

Applications: web servers, word processors, browsers, games and parallel computation.

Q11. (a) Explain the difference between long term, short term and medium term schedulers. (b) Explain the layered approach of the Operating System. (2022)

(a) Long-term, short-term and medium-term schedulers

	Long-term	Short-term	Medium-term
Also called	Job scheduler	CPU scheduler	Swapper
Function	Selects jobs from the pool into memory	Selects a ready process for the CPU	Swaps processes out and in
Frequency	Rare	Very frequent (milliseconds)	Intermediate
Controls	Degree of multiprogramming	CPU allocation	Memory usage
Speed	Slowest	Fastest	Medium
Time-sharing systems	Minimal or absent	Present	Present



(b) Layered approach of the operating system

In the layered approach the OS is divided into a number of **layers**. The bottom layer (layer 0) is the hardware and the top layer (layer N) is the user interface. Each layer is built on top of the layers below it and uses only their services.

Layer	Function
5	User programs
4	I/O buffering and device drivers
3	Operator-console communication
2	Memory management
1	CPU scheduling
0	Hardware

Advantages: simple construction and debugging, because each layer is tested using only the lower verified layers; modular design; easy to change one layer without affecting others.

Disadvantages: careful definition of the layers is difficult, because a layer can use only lower layers; less efficient, because a request passes through many layers, which adds overhead.

A comparison: a **microkernel** keeps only the minimum functions in the kernel and runs other services in user space, which is more secure but slower because of message passing.

Q12. Write in short - (i) Virtual machine (iv) Mutual exclusion. (2023)

(i) Virtual machine

A **virtual machine (VM)** is a software layer that makes one physical computer appear as several separate computers, each running its own operating system. A **virtual machine monitor (hypervisor)** runs on the bare hardware (or on a host OS) and gives every VM a virtual copy of the CPU, memory, disk and I/O devices.

- The VM and its OS cannot see or harm other VMs, so there is complete **protection**.
- Several different operating systems can run on the same machine at the same time.
- Used for server consolidation, testing, cloud computing and software development.

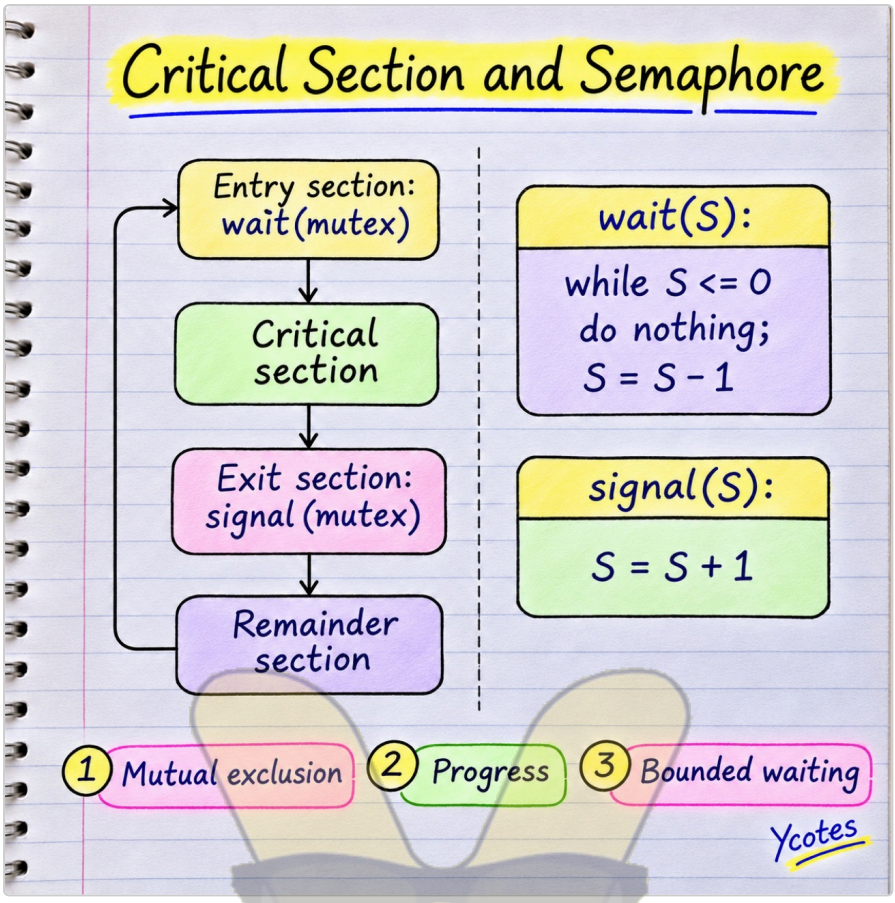
Examples: VMware, VirtualBox, Hyper-V, and the Java Virtual Machine (JVM) for running Java bytecode.

Advantages: isolation, easy backup and migration, better hardware use.
Disadvantage: a performance overhead and difficulty of sharing resources between VMs.

(iv) Mutual exclusion

Mutual exclusion means that when one process is using a shared resource or executing its critical section, no other process is allowed to use it at the same time. It is the first requirement of the critical-section problem and it prevents race conditions.

```
wait(mutex);
   critical section
signal(mutex);
```



It is implemented by lock variables, **Peterson's algorithm**, hardware instructions such as test-and-set, **semaphores** and monitors. The other two requirements of a correct solution are **progress** and **bounded waiting**.
