



PAPER II · CSE & IT · 300 MARKS

TGPSC HMWSSB MANAGER (ENGINEERING) · PAPER II

Computer Science & Information Technology

Degree-level notes on all 16 units of TGPSC's 2026 syllabus, with every question of TSPSC's 2015 CS & IT paper solved from the official key

English Medium | **2026-27 Edition**

16	147	190+
units of the 2026 syllabus	TSPSC 2015 questions solved	practice MCQs

INSIDE

01 Probability, statistics and engineering mathematics
02 Sets, relations, algebraic structures and linear algebra
03 Combinatorics, graph theory and mathematical logic
04 Digital logic and number representation
05 Computer organisation and architecture
06 Programming in C and data structures
07 Algorithms
08 Theory of computation
09 Compiler design
10 Operating systems
11 Databases
12 Information systems and software engineering
13 Computer networks and network security
14 Web technologies
15 OOP concepts, C++ and Java
16 Data mining



Jupudi Anil Kumar
UPSC faculty & mentor

FREE PYQ PRACTICE
learnwithanil.in

About this book



Jupudi Anil Kumar

UPSC faculty & mentor · 1000+ hours of civil-services teaching

learnwithanil.in · Instagram @learn_with_anil06

This book is for **Paper II (Computer Science Engineering & Information Technology)** of the TGPSC recruitment test for **Manager (Engineering) in the Hyderabad Metropolitan Water Supply & Sewerage Board** (Notification No. 04/OG/HM/2026, 18 September 2026; 130 posts, of which 5% are shared by CSE, IT and ECE graduates). Paper II carries **two-thirds of your marks**, so it decides the merit list.

Paper	Questions	Marks	Time	Language
Paper I: General Studies & General Abilities	150	150	150 min	English and Telugu
Paper II: Computer Science Engineering & IT (degree level)	**150**	**300**	**150 min**	**English**

Selection is on the written marks only (no interview). The notification does not mention negative marking. The exam is tentatively in **December 2026**: check tgpsc.gov.in for the date and hall ticket.

How TSPSC asks it. The only Manager (Engineering) CS & IT paper published so far is the **2015 paper (Notification 13/2015, code 502)**. All its **147 valid questions** (TSPSC dropped Q72, Q96 and Q106) are in this book, each in the chapter where it belongs, with the answer from **TSPSC's final key** and a short explanation. It shows what to expect: GATE-style questions, slightly easier, with much discrete maths and networking, short C programs to trace, and textbook definitions.

Unit of the 2026 syllabus (chapter)	2015 Qs	Unit	2015 Qs
1 Probability, statistics & engineering maths	8	9 Compiler design	9
2 Sets, algebra & linear algebra	9	10 Operating systems	12
3 Combinatorics, graphs & logic	12	11 Databases	11
4 Digital logic	5	12 Information systems & software engineering	6
5 Computer organisation & architecture	13	**13 Computer networks & security**	**25**
6 C programming & data structures	13	14 Web technologies	2
7 Algorithms	15	15 OOP, C++ & Java	1
8 Theory of computation	6	16 Data mining (new in 2026)	0

How each chapter works

- Notes on every unit of the official syllabus (new ones such as data mining too), with **formula boxes**, worked numericals, tables and code you can trace by hand.
- **TSPSC 2015** boxes (gold) put each real question right after the idea it tests, with the official answer marked and explained.
- **Trap** boxes show where marks are lost; a **quick-revision** table and **12 practice MCQs** with explanations close each chapter.

Cover photos (Wikimedia Commons): "Retaining wall of Osman Sagar" by iMahesh, CC BY-SA 4.0; "Code on computer monitor" by Markus Spiske, CC0.

© Jupudi Anil Kumar · For the purchaser's personal use only · Do not share

Contents

01 Probability, statistics and engineering mathematics –

02 Sets, relations, algebraic structures and linear algebra –

03 Combinatorics, graph theory and mathematical logic –

04 Digital logic and number representation –

05 Computer organisation and architecture –

06 Programming in C and data structures –

07 Algorithms –

08 Theory of computation –

09 Compiler design –

10 Operating systems *(in this sample)* 3

11 Databases –

12 Information systems and software engineering –

13 Computer networks and network security –

14 Web technologies –

15 OOP concepts, C++ and Java –

16 Data mining –

© Jupudi Anil Kumar · For the purchaser's personal use only · Do not share

CHAPTER 10

Operating systems

Syllabus unit 10: Processes, Threads, Inter-process communication, Concurrency, Synchronization, Deadlock, CPU scheduling, Memory management and virtual memory, File systems, I/O systems, Protection and security, System calls, Disk scheduling · TSPSC 2015: 12 questions (Q11–20, Q29, Q100)

Operating systems gave **12 of the 147 questions** in TSPSC's 2015 paper, and they came in a block (Q11 to Q20). Most were one-line concept questions (which scheduler, which state, what a semaphore does), plus two that needed working: a page-fault count and a deadlock trace. This chapter follows the syllabus order and covers every idea those questions used, plus the numericals that GATE-style papers love: scheduling averages, page replacement, the Banker's algorithm and disk-head movement.

10.1 What an operating system does

ASKED BY TSPSC

An operating system (OS) is the program that **manages the hardware and gives programs a convenient, protected way to use it**. It is both a **resource manager** (CPU time, memory, disk, devices) and an **extended machine** that hides the hardware behind simple abstractions.

OS abstraction	Hardware it stands for	What the OS gives programs
Process / **thread**	**CPU**	the illusion that each program has its own processor
Virtual address space	**Main memory**	a private, large, contiguous memory
File system	**Disk**	named files and folders instead of sectors
Signal	**Interrupt**	a software notice of an event

TSPSC 2015 · PAPER 2 CS & IT · Q29 — Processes, Threads & CPU Scheduling

Listed below are some of the Operating System abstractions and related hardware components:

i. Thread	p) Interrupt
ii. Virtual Address Space	q) Memory
iii. File System	r) CPU
iv. Signal	s) Disk

Which of the following correctly relates the Operating System abstractions and related hardware components?

(1) i-p ii-q iii-r iv-s

(2) i-r ii-s iii-q iv-p

(3) i-r ii-q iii-s iv-p

(4) i-r ii-p iii-q iv-s

Official answer: (3) i-r ii-q iii-s iv-p

Each OS abstraction virtualises a hardware resource: a thread is an abstraction of the CPU, a virtual address space is an abstraction of physical memory, a file system is an abstraction of the disk, and a signal is the software counterpart of a hardware interrupt. So i-r, ii-q, iii-s, iv-p.

Dual mode. The CPU runs in **user mode** (programs) or **kernel mode** (the OS). Privileged instructions (I/O, changing the timer, disabling interrupts) work only in kernel mode. A program asks the OS for a service through a **system call**, which traps into kernel mode and back.

System-call group	UNIX examples	Windows examples
Process control	`fork()`, `exec()`, `exit()`, `wait()`	CreateProcess(), ExitProcess()

© Jupudi Anil Kumar · For the purchaser's personal use only · Do not share

System-call group	UNIX examples	Windows examples
File management	`open()`, `read()`, `write()`, `close()`	CreateFile(), ReadFile()
Device management	`ioctl()`, `read()`, `write()`	SetConsoleMode()
Information	`getpid()`, `alarm()`, `sleep()`	GetCurrentProcessID()
Communication	`pipe()`, `shmget()`, `mmap()`	CreatePipe(), MapViewOfFile()
Protection	`chmod()`, `umask()`, `chown()`	SetFileSecurity()

Kernel designs

Design	Idea	Plus	Minus
Monolithic	the whole OS in one kernel address space (classic UNIX, Linux)	fastest (direct function calls)	a bug anywhere can crash everything
Layered	layer *n* uses only layer *n* − *1* (THE system)	**easy to design and debug** layer by layer	**less efficient**: a call crosses many layers
Microkernel	only IPC, scheduling and basic memory in the kernel; drivers, file system in user space (Mach, QNX)	**more secure and reliable**; adding a service **rarely changes the kernel**	slower (message passing)
Modular / hybrid	core kernel + loadable modules (Linux modules, Windows, macOS)	flexible	—

`TSPSC 2015 · PAPER 2 CS & IT · Q15` `Processes, Threads & CPU Scheduling`

Which of the following statements is incorrect with regard to operating system design?

(1) Layered approach is easy to design and debug

(2) Layered approach is less efficient

(3) Changes to the kernel tend to be fewer with microkernel approach

(4) Microkernels are less secure and reliable

Official answer: (4) Microkernels are less secure and reliable

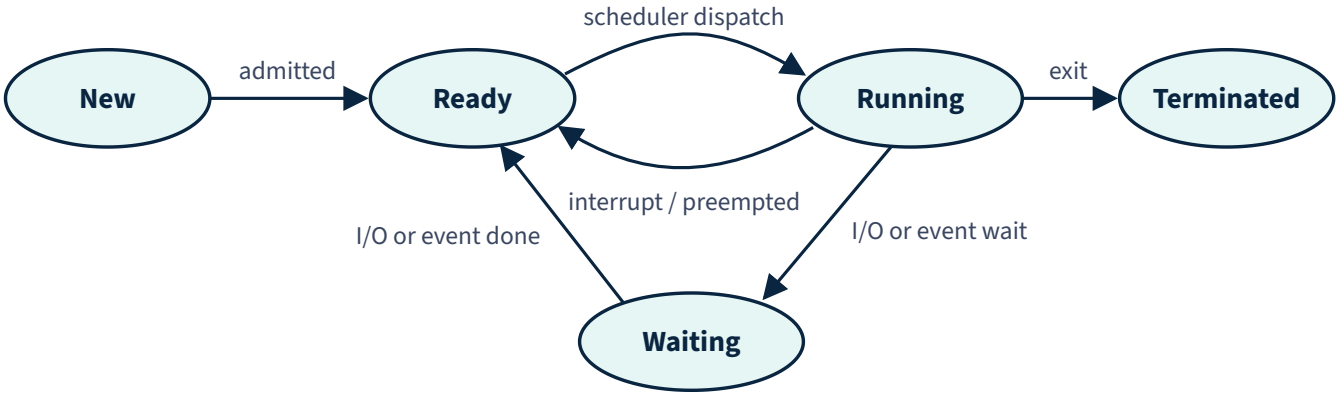
A microkernel keeps only minimal services in the kernel and runs most services as user-level processes, so a failing service does not crash the kernel; microkernels are therefore more secure and reliable, which makes (4) incorrect. The layered approach is simple to construct and debug but less efficient because a call passes through several layers, and in a microkernel new services are added in user space, so the kernel changes less often.

10.2 Processes

ASKED BY TSPSC

A **process** is a program in execution: the code (text section) plus its current activity, i.e. program counter, registers, **stack** (function calls, local variables), **data** section (globals) and **heap** (dynamic memory). The OS keeps everything about a process in its **process control block (PCB)**: state, PC, registers, scheduling information, memory-management information (page tables), accounting and I/O status (open files).

© Jupudi Anil Kumar · For the purchaser's personal use only · Do not share



Five-state process model

Transition	Happens when
Ready → Running	the short-term scheduler dispatches the process
Running → Ready	the process is **preempted**: its time slice ends or a **higher-priority process becomes ready**
Running → Waiting	it starts **I/O** or waits on an event or a **semaphore**
Waiting → Ready	the I/O finishes or the event occurs (never directly to Running)

TSPSC 2015 · PAPER 2 CS & IT · Q13 — Processes, Threads & CPU Scheduling

In the process state transition diagram, transition from running to ready state indicates that

(1) the running process is preempted by a higher priority process

(2) the running process initiated an input/output operation

(3) the running process is blocked for a semaphore operation

(4) that the OS is not ready to service the running process

Official answer: (1) the running process is preempted by a higher priority process

A process goes from running back to ready when it is preempted, i.e. it is still able to run but loses the CPU, for example because a higher-priority process has become ready or its time slice has expired. Starting an I/O operation or blocking on a semaphore moves the process from running to the waiting (blocked) state, not to ready.

Schedulers

Scheduler	Chooses	Runs	Controls
Long-term (job) scheduler	which jobs enter memory from the job pool	rarely (seconds, minutes)	the **degree of multiprogramming**
Short-term (CPU) scheduler	which ready process gets the CPU	very often (milliseconds)	CPU allocation
Medium-term scheduler	which process to swap out and back in	when memory is short	the mix of processes in memory

© Jupudi Anil Kumar · For the purchaser's personal use only · Do not share

TSPSC 2015 · PAPER 2 CS & IT · Q19 — Processes, Threads & CPU Scheduling

Which scheduler controls the degree of multiprogramming?

(1) Short-term scheduler
(2) Long-term scheduler
(3) Middle-term scheduler
(4) Disk Scheduler

Official answer: (2) Long-term scheduler

The long-term (job) scheduler decides which jobs are admitted from the job pool into main memory, so it controls the degree of multiprogramming, i.e. the number of processes in memory. The short-term scheduler only picks which ready process gets the CPU next.

A **context switch** saves the PCB of the running process and loads the PCB of the next one. It is pure overhead; the **dispatcher** does it, and the time it takes is the **dispatch latency**.

Creating processes: fork() and exec()

```
pid_t pid = fork();          /* one call, two returns           */
if (pid == 0)                /* child: fork() returned 0         */
    execlp("/bin/ls", "ls", NULL);   /* replace the child's image */
else if (pid > 0)            /* parent: fork() returned the child's PID */
    wait(NULL);              /* wait for the child to finish     */
```

`fork()` gives the child a **copy** of the parent's address space (stack, heap, data), made lazily with **copy-on-write**, so later changes in one are not seen by the other. Open files and **shared-memory segments** stay shared. `exec()` replaces the process image with a new program. A child that has finished but whose parent has not yet called `wait()` is a **zombie**; a child whose parent has died is an **orphan** (adopted by `init` / `systemd`).

TSPSC 2015 · PAPER 2 CS & IT · Q16 — Processes, Threads & CPU Scheduling

When a process creates a new process using the fork() operation, then which of the following structures are shared between the parent process and the child process?

(1) Stack
(2) Heap
(3) Shared Memory Segments
(4) Stack, Heap and Shared Memory Segments

Official answer: (3) Shared Memory Segments

fork() gives the child its own copy of the parent's address space (stack, heap and data, usually implemented with copy-on-write), so a later change by one process is not seen by the other. Shared memory segments attached by the parent stay attached in the child and refer to the same physical memory, so only they are shared.

COUNTING PROCESSES MADE BY FORK()

n `fork()` calls in sequence (no conditions) give **2^n processes in all**, i.e. **$2^n - 1$ new (child) processes**.
`fork(); fork(); fork();` prints "hello" 8 times if `printf("hello")` follows them.

Inter-process communication (IPC). Cooperating processes exchange data by **shared memory** (fast; the processes must synchronise themselves) or **message passing** (`send` / `receive` through the kernel: pipes, message queues, sockets; easier, slower). Pipes are one-way; a named pipe (FIFO) works between unrelated processes.

© Jupudi Anil Kumar · For the purchaser's personal use only · Do not share

10.3 Threads

ASKED BY TSPSC

A **thread** is a unit of CPU use inside a process: it has its own **thread ID, program counter, register set and stack**, and shares the process's **code, data, heap and open files** with the other threads. Threads are cheaper than processes to create and switch, they share memory without IPC, and a blocked thread need not stop the others.

	User-level threads	Kernel-level threads
Managed by	a thread library in user space (green threads)	the OS kernel
Kernel knows them?	**No**: it sees one process	Yes
Create / switch cost	**cheap** (no system call)	**more expensive** (trap to kernel)
One thread blocks on I/O	the whole process blocks (many-to-one)	only that thread blocks
Run on several CPUs at once	no	yes

Mapping models: **many-to-one** (user threads on one kernel thread), **one-to-one** (Linux, Windows), **many-to-many** (Solaris before 9).

TSPSC 2015 · PAPER 2 CS & IT · Q17 — Processes, Threads & CPU Scheduling

Which of the following statements is incorrect with respect to threads?

(1) User-level threads are unknown by the kernel

(2) Kernel threads are more expensive

(3) User-level threads are more expensive

(4) Thread creation uses fewer resources than process creation

Official answer: (3) User-level threads are more expensive

User-level threads are managed by a thread library in user space, so the kernel does not know about them and creating or switching them needs no system call; they are cheaper than kernel threads, whose every operation needs a trap into the kernel. So "user-level threads are more expensive" is the incorrect statement; the others are true.

10.4 CPU scheduling

Criteria. Maximise **CPU utilisation** and **throughput** (processes finished per unit time); minimise **turnaround time**, **waiting time** and **response time**.

THE THREE TIMES

Turnaround time (TAT) = completion time − arrival time
Waiting time (WT) = TAT − burst time (time spent in the ready queue)
Response time = first time on the CPU − arrival time

Algorithm	Rule	Preemptive?	Note
FCFS	in order of arrival	no	**convoy effect**: short jobs wait behind a long one
SJF	shortest next CPU burst	no	**optimal average waiting time**; long jobs can **starve**
SRTF (preemptive SJF)	shortest *remaining* time; a new shorter job preempts	yes	optimal among preemptive policies
Priority	highest priority first	either	**starvation** of low priority; cured by **aging**

© Jupudi Anil Kumar · For the purchaser's personal use only · Do not share

Algorithm	Rule	Preemptive?	Note
Round robin (RR)	each gets a time quantum q in turn	yes	good response; q too large → FCFS, too small → too many context switches
Multilevel queue / feedback	separate queues; MLFQ moves jobs between queues	yes	most general, most complex

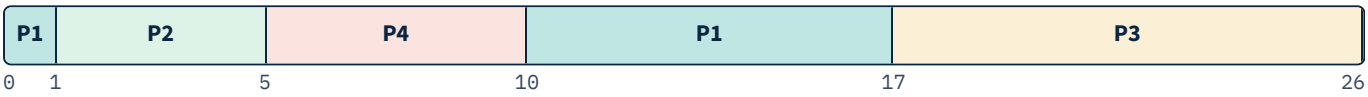
Worked example. Four processes arrive as below (times in ms).

Process	Arrival	Burst
P1	0	8
P2	1	4
P3	2	9
P4	3	5

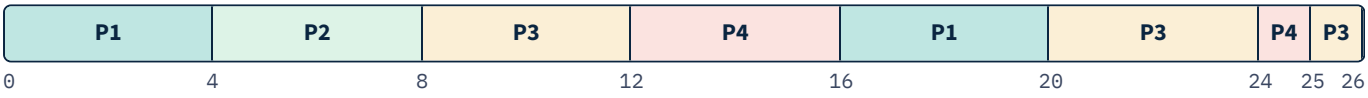
FCFS



SRTF (PREEMPTIVE SJF)



ROUND ROBIN, Q = 4



Policy	Waiting times P1, P2, P3, P4	Average waiting time
FCFS	0, 7, 10, 18	**8.75 ms**
SJF (non-preemptive): P1, P2, P4, P3	0, 7, 15, 9	**7.75 ms**
SRTF	9, 0, 15, 2	**6.5 ms** (the lowest)
RR, q = 4	12, 3, 15, 17	**11.75 ms**

In SRTF, P1 runs for 1 ms; at t = 1, P2 (4 ms) is shorter than P1's remaining 7 ms, so it preempts. P4 (5) runs next because at t = 5 it is shorter than P1 (7) and P3 (9).

> WAITING TIME IS NOT THE START TIME
>
> In a preemptive schedule a process can wait several times. Always use **WT = completion − arrival − burst**. For P1 under SRTF: 17 − 0 − 8 = 9 ms, not 0.

10.5 Synchronisation

ASKED BY TSPSC

When processes share data, the result can depend on the order in which they run (a **race condition**). The code that touches shared data is the **critical section**. A correct solution must give:

1. **Mutual exclusion**: at most one process in its critical section;

© Jupudi Anil Kumar · For the purchaser's personal use only · Do not share

2. **Progress**: if none is inside, the choice of who enters next cannot be postponed forever by processes not trying to enter;
3. **Bounded waiting**: there is a limit on how many times others can enter before a waiting process gets its turn.

Peterson's solution (two processes) uses `flag[2]` and `turn` and satisfies all three; it assumes loads and stores are not reordered. Hardware gives atomic **test-and-set** and **compare-and-swap** instructions for locks.

Semaphores. A semaphore *S* is an integer touched only by two **atomic** operations:

```
wait(S)   /* P, down */ :  while (S <= 0) ;  /* busy wait, or block */   S--;
signal(S) /* V, up   */ :  S++;
```

A **binary semaphore** (0/1) works as a mutex lock; a **counting semaphore** counts free units of a resource. A semaphore is a **synchronisation tool**: it enforces mutual exclusion and ordering between processes; it does not schedule processes or detect deadlocks.

COUNTING SEMAPHORE ARITHMETIC

Final value = initial value − (number of P/wait operations) + (number of V/signal operations). A **negative** value $-k$ (blocking implementation) means k processes are waiting.

TSPSC 2015 · PAPER 2 CS & IT · Q18 — Synchronisation & Deadlocks

Semaphore can be used for _____.

(1) detecting deadlock
(2) process synchronization
(3) process scheduling
(4) defining priority of the processes

Official answer: (2) process synchronization

A semaphore is an integer variable accessed only through the atomic operations wait (P) and signal (V); it is a tool for process synchronization, for example to enforce mutual exclusion on a critical section or to order events between processes. It does not detect deadlocks, schedule processes or set priorities.

Classic problems. Producer–consumer (bounded buffer: semaphores `mutex = 1`, `empty = n`, `full = 0`), **readers–writers** (many readers or one writer), **dining philosophers** (five philosophers, five chopsticks: picking up left then right can deadlock). A **monitor** is a language construct that lets only one process be active inside it, with **condition variables** (`wait`, `signal`) for waiting.

© Jupudi Anil Kumar · For the purchaser's personal use only · Do not share

Using semaphores in the wrong order causes trouble. Two processes that take the same two semaphores **in opposite orders** can each hold one and wait for the other forever:

TSPSC 2015 · PAPER 2 CS & IT · Q20 — Synchronisation & Deadlocks

Let S and Q be two semaphores initialized to 1, where P_0 and P_1 are the processes defined as follows:

```
Process P1      Process P2
wait (S);       wait(Q);
wait(Q);        wait(S);
    ---;            ----;
signal(S);      signal(Q);
signal(Q);      signal(S);
```

Which one of the following statements is correct with respect to the above construct?

(1) Causes starvation
(2) Causes livelock
(3) Causes a deadlock
(4) Does not ensure mutual exclusion

Official answer: (3) Causes a deadlock

The two processes acquire the semaphores in opposite orders. If the first process executes wait(S) (S becomes 0) and is then preempted, and the second executes wait(Q) (Q becomes 0), the first now blocks on wait(Q) and the second blocks on wait(S). Each holds what the other needs (circular wait), so neither can proceed: a deadlock.

Problem	Meaning
Deadlock	a set of processes each waits for a resource held by another in the set; nobody moves
Starvation	a process waits indefinitely while others keep being served (e.g. low priority, LIFO semaphore queue)
Livelock	processes keep changing state in response to each other but make no progress
Priority inversion	a high-priority process waits for a lock held by a low-priority one that a medium-priority one keeps preempting; fixed by **priority inheritance**

10.6 Deadlocks

Four necessary conditions (Coffman), all must hold together: mutual exclusion, **hold and wait**, **no preemption** and **circular wait**. In a **resource-allocation graph**, a cycle means deadlock if every resource has one instance; with several instances a cycle means only that a deadlock is *possible*.

Strategy	How	Example
Prevention	make one condition impossible	request all resources at once (no hold-and-wait); number the resources and request in increasing order (no circular wait)
Avoidance	grant a request only if the state stays **safe**	**Banker's algorithm** (needs each process's maximum demand in advance)
Detection and recovery	let deadlocks happen, find them (wait-for graph), then abort or preempt	used where deadlocks are rare
Ignore	pretend they never happen ("ostrich")	most desktop OSs

Banker's algorithm: worked example. Five processes, resources A = 10, B = 5, C = 7.

Process	Allocation (A B C)	Max (A B C)	Need = Max − Allocation
P0	0 1 0	7 5 3	7 4 3
P1	2 0 0	3 2 2	1 2 2

© Jupudi Anil Kumar · For the purchaser's personal use only · Do not share

Process	Allocation (A B C)	Max (A B C)	Need = Max − Allocation
P2	3 0 2	9 0 2	6 0 0
P3	2 1 1	2 2 2	0 1 1
P4	0 0 2	4 3 3	4 3 1

Available = (10, 5, 7) − total allocated (7, 2, 5) = **(3, 3, 2)**. Safety check: P1's need (1 2 2) ≤ (3 3 2), so P1 can finish and return its allocation: Work = (5 3 2). Then P3 → (7 4 3), P4 → (7 4 5), P0 → (7 5 5), P2 → (10 5 7). Every process can finish, so the state is **safe**, with safe sequence **⟨P1, P3, P4, P0, P2⟩** (⟨P1, P3, P4, P2, P0⟩ also works).

MINIMUM RESOURCES THAT RULE OUT DEADLOCK

If *n* processes each need at most *k* units of one resource type, deadlock is impossible when the number of units is at least **n(k − 1) + 1**. Three processes needing 2 units each: 3 × 1 + 1 = 4 units.

10.7 Memory management

ASKED BY TSPSC

The memory-management part of an OS **keeps track of which parts of memory are in use and by whom**, **decides which processes (or parts) to move in and out of memory**, and **allocates and frees memory as needed**. Managing free space on disk and **storage allocation** on disk belong to **secondary-storage (disk) management**, not memory management.

TSPSC 2015 · PAPER 2 CS & IT · Q14 — Memory Management & Virtual Memory

Which of the following activities is not done by memory management?

(1) Keep an account of which parts of memory are currently being used by each of the processes

(2) Decide which processes are to be brought into memory when memory space becomes available

(3) Allocate and de-allocate memory space as needed

(4) Storage allocation

Official answer: (4) Storage allocation

The memory-management activities of an OS are keeping track of which parts of main memory are in use and by whom, deciding which processes (or parts of them) to move into and out of memory, and allocating and de-allocating memory space as needed. Storage allocation (allocating space on disk) is part of secondary-storage (disk) management, together with free-space management and disk scheduling.

Address binding. A program uses **logical (virtual) addresses**; the **memory-management unit (MMU)** turns them into **physical addresses** at run time. With a relocation (base) register: physical = logical + base, checked against a limit register.

Contiguous allocation gives each process one block. With **variable (dynamic) partitions**, free holes are chosen by **first fit** (first hole big enough; fast), **best fit** (smallest hole big enough; leaves tiny holes) or **worst fit** (largest hole).

Fragmentation	Where	Cure
External	free memory is split into many small holes between allocated blocks (variable partitions, segmentation)	**compaction** (move blocks together), or paging
Internal	wasted space *inside* an allocated block (fixed partitions, the last page of a process)	smaller blocks or pages

© Jupudi Anil Kumar · For the purchaser's personal use only · Do not share

TSPSC 2015 · PAPER 2 CS & IT · Q11 Memory Management & Virtual Memory

Which of the following techniques can be used to counter the problem of external fragmentation that occurs in dynamic partitioning memory management?

(1) Compaction (2) Segmentation

(3) Swapping (4) Splitting

Official answer: (1) Compaction

In dynamic (variable) partitioning, free memory gets broken into many small holes between allocated partitions (external fragmentation). Compaction shuffles the allocated partitions together so that all the free holes merge into one large contiguous block, which counters this problem. Segmentation itself suffers from external fragmentation, and swapping or splitting do not merge the holes.

Paging divides logical memory into fixed **pages** and physical memory into **frames** of the same size, so a process need not be contiguous: **no external fragmentation** (only internal, in the last page). A logical address is split into a **page number** (index into the page table) and an **offset**.

PAGING ARITHMETIC

Page size 2^d bytes → offset = **d bits**; logical address of *m* bits → page number = **m − d bits**, page-table entries = 2^{m-d}.

Example: 32-bit addresses, 4 KB (2^{12}) pages → 12-bit offset, **20-bit page number**, $2^{20} \approx 1$ million entries; at 4 bytes each the page table takes **4 MB**.

A page table in memory doubles every memory access, so a small fast cache of recent translations, the **TLB (translation look-aside buffer)**, is used.

EFFECTIVE ACCESS TIME WITH A TLB

EAT = h (t + m) + (1 − h)(t + 2m), where *h* = TLB hit ratio, *t* = TLB lookup time, *m* = memory access time.
h = 0.8, t = 20 ns, m = 100 ns: EAT = 0.8 × 120 + 0.2 × 220 = **140 ns**.

Large page tables are handled by **multilevel (hierarchical) paging**, **hashed page tables** or an **inverted page table** (one entry per **physical frame**, searched by process ID and page number). **Segmentation** splits a program into logical units (code, stack, data) of different sizes, each with a base and limit; it matches the programmer's view but suffers external fragmentation. Intel x86 combines segmentation with paging.

10.8 Virtual memory

ASKED BY TSPSC

Virtual memory lets a process run when **only part of it is in memory**. With **demand paging**, a page is loaded only when it is first touched; touching a page that is not in memory causes a **page fault**: the OS finds the page on disk, picks a free frame (or a victim), reads the page in, updates the page table and restarts the instruction.

EFFECTIVE ACCESS TIME WITH PAGE FAULTS

EAT = (1 − p) × memory access + p × page-fault service time. With 200 ns memory and an 8 ms fault, even p = 1/1000 gives about 8.2 µs: page faults must be very rare.

To count page faults, first turn addresses into page numbers (address ÷ page size), then run the replacement policy:

© Jupudi Anil Kumar · For the purchaser's personal use only · Do not share

TSPSC 2015 · PAPER 2 CS & IT · Q12 Memory Management & Virtual Memory

The addresses generated by a particular program in a pure demand paging system with 100 records per page and one free main memory frame are as follows
0100, 0200, 0420, 0489, 0510, 0530, 0599, 0120, 0220, 0230, 0260, 0320, 0370
The number of page faults generated in order to execute the program is

(1) 13 (2) 8
(3) 7 (4) 10

Official answer: (3) 7
With 100 records per page, page number = address ÷ 100, so the reference string is 1, 2, 4, 4, 5, 5, 5, 1, 2, 2, 2, 3, 3. With only one frame, a fault occurs every time the page changes: 1 (F), 2 (F), 4 (F), 4, 5 (F), 5, 5, 1 (F), 2 (F), 2, 2, 3 (F), 3. That is 7 page faults.

Page replacement. Reference string 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1 with **3 frames**:

Algorithm	Replaces	Page faults
FIFO	the page loaded earliest	**15**
Optimal (OPT, Belady's MIN)	the page that will not be used for the longest time (needs the future; used as a benchmark)	**9** (the minimum)
LRU	the page not used for the longest time in the past	**12**
LFU / MFU, Second chance (clock)	least / most frequently used; FIFO with a reference bit	—

Belady's anomaly: with FIFO, more frames can give *more* faults. For 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5, FIFO gives 9 faults with 3 frames but **10 with 4**. LRU and OPT are **stack algorithms** and never show it.

Thrashing is when a process spends more time paging than executing because it has fewer frames than its working set; CPU utilisation collapses as the degree of multiprogramming rises. Cures: the **working-set model** (keep each process's recently used pages in memory) and **page-fault-frequency** control; reduce the degree of multiprogramming.

😂 **MEME BREAK** **Counting page faults in the exam hall**

😫 **Trying to remember the addresses and the frames in your head** ✗

😎 **Write the reference string first, then one column per frame** ✓

👉 Q12 of TSPSC 2015 is 7 faults once 0100 … 0370 become pages 1, 2, 4, 4, 5 …

10.9 File systems, I/O and disk scheduling

File allocation methods

Method	How	Good	Bad
Contiguous	each file in consecutive blocks (start, length)	fast sequential and direct access	**external fragmentation**; files cannot grow easily

© Jupudi Anil Kumar · For the purchaser's personal use only · Do not share

Method	How	Good	Bad
Linked	each block points to the next (FAT keeps the links in a table)	no external fragmentation	slow direct access; a broken pointer loses the rest
Indexed	an index block lists all the file's blocks (UNIX inode: 12 direct, single, double, triple indirect)	fast direct access	index block overhead

Directories can be single-level, two-level, **tree-structured** or acyclic-graph (shared files through links). Free space is tracked with a **bit vector**, a linked free list, grouping or counting. **I/O** is done by **programmed I/O** (CPU polls), **interrupt-driven I/O** or **DMA** (a controller moves whole blocks to memory); buffering, caching and **spooling** (a printer queue on disk) smooth speed differences.

Disk scheduling. Seek time dominates, so the OS orders the requests. Queue: 98, 183, 37, 122, 14, 124, 65, 67; head at **53**; cylinders 0–199.

Algorithm	Order served	Total head movement
FCFS	98, 183, 37, 122, 14, 124, 65, 67	**640** cylinders
SSTF (shortest seek first)	65, 67, 37, 14, 98, 122, 124, 183	**236** (can starve far requests)
SCAN (elevator), moving towards 0	37, 14, [0], 65, 67, 98, 122, 124, 183	53 + 183 = **236**
LOOK, moving towards 0	37, 14, 65, 67, 98, 122, 124, 183	39 + 169 = **208**
C-SCAN, moving towards 199	65, 67, 98, 122, 124, 183, [199], jump to 0, 14, 37	146 + 199 + 37 = 382 (183 if the return sweep is not counted)

READ THE QUESTION'S CONVENTION

SCAN goes to the **end of the disk** before turning; LOOK turns at the **last request**. For C-SCAN and C-LOOK, books differ on counting the return jump: use the convention the question states, and if it is silent, count it.

RAID: level 0 striping (speed, no redundancy), 1 mirroring, 5 block striping with distributed parity (survives one disk failure), 6 double parity (two failures), 10 = mirroring + striping.

10.10 Protection and security

ASKED BY TSPSC

Protection controls which **domains** (users, processes) may do which **operations** on which **objects** (files, devices, memory). The model is the **access matrix**: rows = domains, columns = objects, entries = rights (read, write, execute, owner…).

Stored as	Is	Example
Access-control list (ACL)	a **column**: for one object, the domains and their rights	UNIX permission bits, Windows ACLs
Capability list	a **row**: for one domain, the **objects and the operations allowed** on them	a list of capabilities (unforgeable tickets) a process holds
Global table	⟨domain, object, rights⟩ triples	rarely used (too big)

© Jupudi Anil Kumar · For the purchaser's personal use only · Do not share

TSPSC 2015 · PAPER 2 CS & IT · Q100 File Systems & Disk Scheduling

List of objects together with the operations allowed on these objects is called _____.

(1) Capability list (2) access list
(3) available list (4) access matrix

Official answer: (1) Capability list

In the access-matrix protection model, a capability list belongs to a domain (one row of the matrix): it lists the objects together with the operations the domain may perform on each. An access list is a column (for one object, the domains and their rights), and the access matrix is the whole table.

Security guards against misuse from outside: authentication (passwords, OTP, biometrics), malware (virus, worm, Trojan horse, trap door, logic bomb), buffer-overflow attacks, denial of service; the principle of **least privilege** gives each program only the rights it needs.

Quick revision

Ask	Answer
Controls degree of multiprogramming	Long-term scheduler
Running → ready means	Preempted (higher priority ready / time slice over)
Shared after fork()	Shared-memory segments, open files (not stack or heap)
Cheaper threads	User-level threads (kernel does not know them)
Optimal average waiting time	SJF (SRTF if preemptive)
Starvation cure	Aging
Semaphore is for	Process synchronisation (mutual exclusion, ordering)
Opposite order of two wait()s	Deadlock
Deadlock avoidance	Banker's algorithm (safe state)
External fragmentation cure	Compaction (or paging)
Page number bits, 32-bit address, 4 KB page	20
Belady's anomaly	FIFO only
Thrashing cure	Working-set model; reduce multiprogramming
Objects + allowed operations for a domain	Capability list

Practice questions

10.1 Which of the following is NOT one of the four necessary conditions for a deadlock?

(1) Mutual exclusion (2) Hold and wait
(3) Circular wait (4) Starvation

10.2 A system uses 32-bit logical addresses and a page size of 4 KB. How many bits of the logical address form the page number?

(1) 12 (2) 20
(3) 22 (4) 32

© Jupudi Anil Kumar · For the purchaser's personal use only · Do not share

10.3 Processes P1, P2 and P3 arrive at time 0 in that order with CPU bursts of 24, 3 and 3 ms. What is the average waiting time under FCFS?

(1) 17 ms
(2) 3 ms
(3) 13 ms
(4) 27 ms

10.4 Belady's anomaly, in which adding page frames can increase the number of page faults, can occur with

(1) LRU replacement
(2) Optimal replacement
(3) FIFO replacement
(4) None of these

10.5 The TLB hit ratio is 90%, a TLB lookup takes 10 ns and a memory access takes 100 ns. What is the effective memory-access time?

(1) 110 ns
(2) 119 ns
(3) 210 ns
(4) 120 ns

10.6 Which CPU-scheduling algorithm can make long processes wait indefinitely if short processes keep arriving?

(1) First-come, first-served
(2) Round robin
(3) Shortest job first
(4) None of these

10.7 A counting semaphore S is initialised to 7. Then 20 wait (P) operations and 15 signal (V) operations are performed on it. What is the final value of S?

(1) 2
(2) 12
(3) −2
(4) 0

10.8 When CPU utilisation falls sharply as more processes are admitted, because the processes spend most of their time servicing page faults, the system is said to be

(1) starving
(2) thrashing
(3) deadlocked
(4) in Belady's anomaly

10.9 How many child processes are created by the following code (excluding the original process)?

```
int main() { fork(); fork(); fork(); return 0; }
```

(1) 3
(2) 6
(3) 7
(4) 8

10.10 Which of the following is shared by all the threads of a process?

(1) Stack
(2) Register set
(3) Program counter
(4) Code section, global data and open files

10.11 Three processes share 'm' units of a resource, and each process needs at most 2 units. What is the minimum value of m for which a deadlock can never occur?

(1) 3
(2) 4
(3) 5
(4) 6

10.12 Which file-allocation method suffers from external fragmentation?

(1) Contiguous allocation
(2) Linked allocation
(3) Indexed allocation
(4) None of these

© Jupudi Anil Kumar · For the purchaser's personal use only · Do not share

ANSWERS AND EXPLANATIONS

10.1 (4) The four Coffman conditions are mutual exclusion, hold and wait, no preemption and circular wait; all four must hold at once for a deadlock. Starvation is a different problem: one process waits indefinitely while others are served, and it can happen without any deadlock (e.g. low priority under priority scheduling).

10.2 (2) A 4 KB page is 2^{12} bytes, so the offset takes 12 bits; the rest, 32 − 12 = 20 bits, is the page number, giving 2^{20} page-table entries.

10.3 (1) Under FCFS, P1 runs 0–24, P2 24–27 and P3 27–30, so the waiting times are 0, 24 and 27 and the average is 51 ÷ 3 = 17 ms. Had the short jobs run first (P2, P3, P1) the average would be (0 + 3 + 6) ÷ 3 = 3 ms: this gap is the convoy effect.

10.4 (3) FIFO is not a stack algorithm, so the set of pages in memory with n frames is not always a subset of the set with n + 1 frames, and more frames can give more faults (1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5: 9 faults with 3 frames, 10 with 4). LRU and Optimal are stack algorithms and never show the anomaly.

10.5 (4) On a hit: 10 + 100 = 110 ns; on a miss the page table must also be read: 10 + 100 + 100 = 210 ns. EAT = 0.9 × 110 + 0.1 × 210 = 99 + 21 = 120 ns.

10.6 (3) SJF always picks the shortest next burst, so a steady stream of short jobs can postpone a long job forever (starvation). FCFS serves in arrival order and round robin gives everyone a time slice in turn, so neither starves a process. Aging fixes the problem.

10.7 (1) Each wait decrements S and each signal increments it: 7 − 20 + 15 = 2. The order does not change the final value.

10.8 (2) Thrashing happens when processes do not have enough frames for their working sets, so they keep faulting and the paging device becomes the bottleneck. The usual cure is the working-set model or reducing the degree of multiprogramming.

10.9 (3) Each fork() doubles the number of processes: 1 → 2 → 4 → 8. Eight processes exist at the end, of which one is the original, so $2^3 - 1 = 7$ are newly created children.

10.10 (4) Each thread needs its own stack, registers and program counter to run independently; the code, global data, heap and open files belong to the process and are shared by all its threads, which is why threads can communicate without IPC.

10.11 (2) In the worst case each process holds one unit and waits for its second: 3 units are taken and none is free, which can deadlock. One more unit lets some process finish and release both units, so $m = n(k - 1) + 1 = 3 \times 1 + 1 = 4$.

10.12 (1) Contiguous allocation needs a run of consecutive free blocks for each file, so free space breaks into holes too small to use, as in dynamic memory partitioning. Linked and indexed allocation can use any free block anywhere on the disk.

© Jupudi Anil Kumar · For the purchaser's personal use only · Do not share

That was the free sample.

You have read **Chapter 10**. The full book covers **all 16 units** of TGPSC's 2026 Paper II syllabus for CSE & IT: probability and engineering maths, discrete maths, digital logic, computer organisation, C and data structures, algorithms, theory of computation, compilers, operating systems, databases, software engineering, computer networks, web technologies, C++ and Java, and data mining.

All 147 questions of TSPSC's 2015 CS & IT paper are solved inside the chapters, with the official answer, plus **12 practice MCQs per chapter** with explanations.

The book is included in the **₹149 All-in-One Pass**, with every previous paper and mock test of every exam on the site, for 12 months.

learnwithanil.in/tgpsc-hmwssb-manager/notes/

Jupudi Anil Kumar
UPSC faculty & mentor