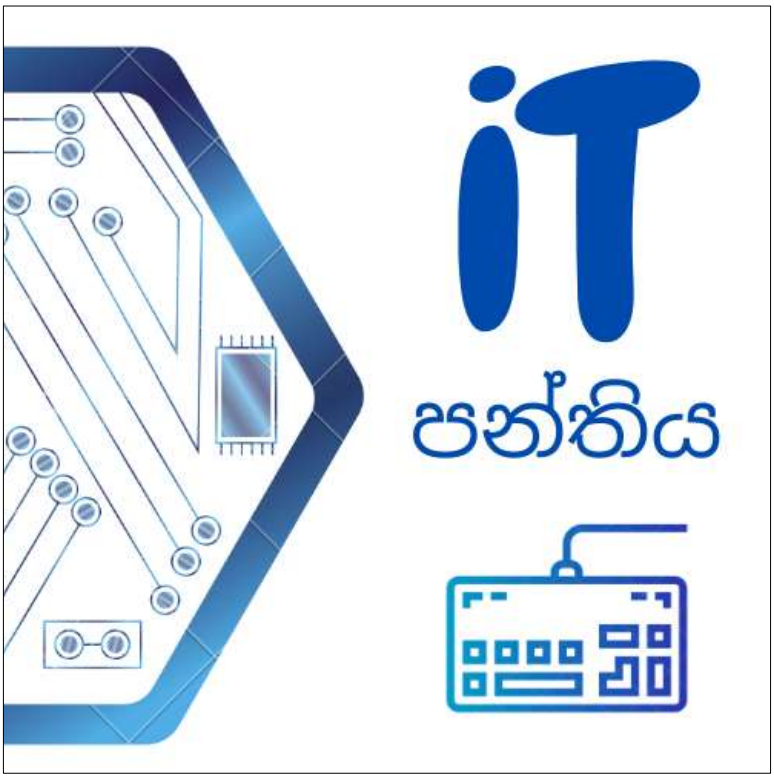
UNIT REVISING TEST - 2026

MCQ Discussion



JULY 22, 2020

P. M. SANJULA NADEESHANI

MCQ Answers

(01)	C	(11)	C	(21)	B	(31)	C
(02)	C	(12)	C	(22)	C	(32)	C
(03)	B	(13)	C	(23)	C	(33)	A
(04)	C	(14)	C	(24)	B	(34)	C
(05)	B	(15)	C	(25)	D	(35)	C
(06)	C	(16)	B	(26)	C	(36)	C
(07)	B	(17)	D	(27)	D	(37)	C
(08)	B	(18)	A	(28)	C	(38)	C
(09)	C	(19)	C	(29)	B	(39)	D
(10)	D	(20)	A	(30)	D	(40)	B

Q1 –

- **Disk partitioning** is the operation that divides a single physical hard drive into multiple **logical drives**, each capable of hosting a separate OS.
- This is critical in **dual-boot setups**, as each OS requires its own partition to operate without interfering with the other.
- **Potential consequence:** If disk partitioning is done on a drive that already has data (without backup), it can lead to **data loss**, especially if existing partitions are resized or overwritten.

📝 **Key Point:** Disk partitioning is essential for dual-boot, but must be handled carefully to avoid overwriting data on active disks.

Q2 –

- A **system diagnosis utility** is used to **scan** the system for errors, monitor hardware status (CPU, RAM, disk), and identify failing components or software conflicts.
- These tools are part of most modern OSs (e.g., Windows' built-in **"System Diagnostics"** or **"Reliability Monitor"**).
- It helps in **preventive maintenance** and **troubleshooting**, which is exactly what the admin is doing in the scenario.

📝 **Key Point:** This tool provides **detailed diagnostics**, which matches the scenario of identifying hidden faults in desktop systems.

Q3 –

Why B is Correct:

- This is a classic case of **external fragmentation**, where parts of a file are scattered across **non-contiguous sectors**.
- This happens when files are modified, deleted, and saved over time, causing gaps in the disk that are later filled in fragmented ways.
- The solution is a **file defragmentation utility**, which reorganizes files on the disk so that they are stored in **contiguous blocks**, speeding up access time.

📝 **Key Point:** The slowness is due to fragmented storage, and defragmentation directly addresses this issue by reducing disk seek time.

Here's a detailed explanation of why **C, B, and C** are the correct answers for questions (4), (5), and (6), based on the **real-world scenarios** described:

Q4 –

- After creating a partition, it must be **formatted with a file system** (e.g., FAT32, NTFS, ext4) before it becomes usable.
- If the partition is created but not formatted, the OS has **no structure to organize files**, making it **unwritable**.
- This is a common issue during OS installations when the partitioning is done but the formatting step is skipped.

📝 **Key Point:** A partition is just a raw section of the disk. Formatting installs the file system, making it usable for storage.

Q5 –

- **Anti-virus software** actively scans files **before** they are opened or executed.
- It uses pattern recognition, heuristic scanning, and real-time monitoring to **detect malware**.
- It is **preferred** in high-security environments because it offers **proactive protection**, unlike backups (which only restore) or task managers (which intervene after infection).

📝 **Key Point:** The need is for real-time **threat prevention**, not damage control — which is exactly the role of antivirus software.

Q6 –

- **Formatting** will **erase all data** (including corrupted files) and prepare the USB drive for **fresh use**.

- It's the standard method to “clean” a drive when it’s going to be reused, especially in cases where **some files can’t be deleted normally due to corruption**.
- The **critical caution** here is that formatting is **destructive** — it removes everything — so important data must be **backed up first**.

📝 **Key Point:** Formatting ensures full clearance and reusability but should be done only after confirming that all important files have been saved elsewhere.

Q7 –

🧠 **Applied Explanation:**
In real systems like embedded devices or older OS memory managers, processes are given blocks of fixed size (e.g., 1 KB per block). If an application only needs 600 bytes, the remaining 424 bytes are **unused but still reserved**.
This exact situation is known as **internal fragmentation** — where the **waste is inside allocated space**, not between blocks.
It’s a common issue when using static memory allocation techniques.

Q8 –

Think of a cloud-based server farm where logs on the main server must **match logs on the backup**. If a failure happens, the backup must have **up-to-date, identical records**.
Tools like **rsync**, **OneDrive sync**, or **Google Backup & Sync** ensure that changes made on one side are **automatically mirrored** to the other — this is **data synchronization**.
Without it, logs would drift out of sync, making audits and debugging unreliable.

Q9 –

When files are **edited repeatedly**, especially large project files, they often get **split into fragments** across different sectors of the hard drive.
This happens a lot with large Word docs, video projects, or databases.
As the OS keeps writing new data in **available gaps**, it causes **file fragmentation**.
Eventually, accessing such files becomes slow because the drive head jumps around to gather all parts.
A **disk defragmentation tool** solves this by **reorganizing file chunks to be contiguous**, boosting performance.

Q10 –

This is a textbook example of **external fragmentation** — although **enough total memory** is available, it's **scattered in small chunks**, so no single **continuous block** of 50KB exists.
In real-world systems (especially older OS memory managers), this happens when programs start and stop frequently, leaving small unusable holes in memory.
A viable solution is **compaction**, which **reorganizes** memory to bring all free blocks together, or **paging**, which allows programs to be split across memory locations.

Q11 -

This describes **contiguous allocation**, where all file blocks are placed in one continuous sequence — perfect for **real-time systems** where **quick access** is critical.
For example, in media streaming or time-series databases, direct access speeds up data fetching.
However, as the file grows, it may hit space limitations or lack a large enough continuous space — causing **external fragmentation** or requiring file relocation.

Q12 –

This scenario is ideal for **linked allocation**, where each block points to the next, making **dynamic file growth** easy — new blocks can be added **anywhere** without moving the file.
However, the **drawback** is clear in systems like log storage or document editing apps — to access the 100th block, the system must **traverse 99 previous blocks**, slowing down random access.

Q13 –

This points directly to **Indexed Allocation**, where each file has an **index block** containing pointers to all its data blocks.
In real-world systems like older file systems (e.g., some Linux ext2 setups), even a tiny file (say 100 bytes) still requires an entire block to store the pointer structure.
This causes **overhead** and **waste**, especially when many small files exist — for example, config files or log files in `/etc` or `/var/log`.

Q14 –

This is the hallmark of **Contiguous Allocation**.
For example, in real-world file systems like early FAT12 or some parts of DVD ISO structures, files are stored as one **continuous sequence**, and the system simply notes **where the file begins and how long it is**.
This method allows for **quick access** and is easy to manage with minimal metadata — just a start point and length.
However, it struggles with files that grow or shrink.

Q15 –

This matches **Linked Allocation**, where each block has a pointer to the next.
Real-world example: consider how some legacy email storage systems or older backup systems stored attachments — even if the file blocks were scattered, they maintained integrity via chaining.
But to access the 10th block, the system has to **traverse all previous 9**, leading to performance degradation — like reading a book one page at a time in different rooms of a library.

Q16 –

This exactly describes **Indexed Allocation** — commonly used in Unix-like systems (e.g., ext2/ext3), where an **index block** (like an index page in a book) stores all data block addresses.
This **eliminates external fragmentation** because blocks can be placed anywhere, and it **supports direct access** to any block without traversing others.
In real life, think of how an e-book allows you to jump straight to Chapter 5 using its table of contents — no need to flip through each previous page.

Q17 –

This is a classic **Linked Allocation** method — used in MS-DOS's FAT12/FAT16 systems. While it offers **flexibility in adding/removing data** (you can easily update pointers), you **can't directly jump** to block 10 without **reading blocks 1 to 9** in sequence.
Imagine reading a WhatsApp message thread — if it's a long conversation with replies linked, you have to scroll down from the top to find a specific reply.

Q18 –

This is the simplest addressing strategy, typical of **Contiguous Allocation**.
The system just needs to know the starting point and how far to go — like counting 5 seats from the front row in a theater.
Many CD-ROM file systems and older OS bootloaders use this method for **speed and simplicity**.
The benefit? **Minimal overhead**, and **very fast access** to any block.
The drawback? You must allocate all blocks up front, and file expansion can be tricky.

Q19 –

This is a classic description of **Indexed Allocation**. In this method, each file has a **separate index block** that stores the addresses of all its data blocks.

In a real-world example, imagine you're keeping a full table of contents for a tiny note. The index (pointer list) is **overkill** — it wastes memory if your file only contains 1–2 blocks.
This is especially inefficient on systems with many small files, like **IoT logs** or **configuration scripts**, where the index size may outweigh the data.

Q20 –

This clearly indicates **External Fragmentation**, a well-known issue with **Contiguous Allocation**. The system requires **all blocks to be together** when saving a file.
In real-life, think of trying to park a large bus in a parking lot where many small cars have left random gaps — you have enough total space, but it's scattered. The bus (large file) won't fit.

This problem is common in older file systems like FAT16 or systems storing large media files like **video editing software**.

Q21 –

When a process is waiting for I/O (like disk or printer), it enters the **Blocked (or Waiting) state** — it's **not ready for execution** until the I/O is complete.

In the real world, consider downloading a large file. While waiting, your device doesn't freeze — it lets other apps work (i.e., CPU moves on). The **Interrupt Vector Table (IVT)** is used to manage these hardware I/O requests and must be updated so the system knows which interrupt to handle when I/O is done.

Q22 –

In the **Seven-State Process Model**, if a process is in the "Ready" state but moved out of main memory due to RAM shortage (i.e., **swapping**), it enters the **Suspend Ready** state. This means it's still logically ready for execution, but **physically not in memory**.

🧩 **Real-World Scenario:**
Imagine a media editing app was queued to run, but another app (like a browser with 100 tabs) consumes all RAM. The OS pushes the media app to **virtual memory (e.g., pagefile)** — it's ready, but **"parked"** on disk.

Q23 –

During a **context switch**, the OS saves the **complete state** of the process/thread so it can resume later. This includes:

- **Program Counter (PC):** the address of the next instruction
- **CPU Registers**
- **Memory Map**
  These are stored in the **Process Control Block (PCB)**, a key data structure in all OSs.

🧩 **Real-World Scenario:**
Consider switching between apps on your phone (e.g., WhatsApp ↔ Google Maps). The OS must remember **exactly where you left off** in each app. That memory snapshot is saved in the **PCB**.

Q24 –

If memory isn't available, the OS doesn't put the process directly in the Ready state. Instead, it transitions:

- From **New → Suspend Ready**, meaning it's prepared for execution but **held on disk**.
- The process will be swapped **into RAM** once memory is available.

🧩 **Real-World Scenario:**
Think of a download manager queuing a new file. If your drive is full, the download is paused and **"waiting"** until enough space is cleared. Similarly, the process is **waiting in secondary storage**.

Q25 –

In preemptive multitasking, when a higher-priority process arrives:

- The currently running process is **interrupted** (not killed).
- It's **moved to the Ready state** (not Blocked or Terminated).
- The system performs a **context switch**, saving the state of the preempted process so it can resume later.

🧩 **Real-World Analogy:**
Imagine you're talking to someone (running process), and your boss calls (higher-priority process). You pause your chat (context switch), note where you left off (save state), and attend the boss (switch to the new process). The person you were talking to isn't gone — just waiting (in Ready state) until you return.

Q26 –

A **context switch** does not perform useful computation — instead, it:

- Saves the current process's state (registers, memory map, etc.)
- Loads the next process's state from the PCB
  This is essential but consumes **CPU time** that could have been spent executing useful instructions.

🧩 **Real-World Analogy:**
Switching tabs on your browser doesn't load a webpage — it **just prepares** the tab. Similarly, context switching doesn't execute the program — it **just sets up** the environment. Too many switches = wasted time.

Q27 –

Once an I/O request is complete:

- The **I/O Subsystem** (e.g., disk controller) raises an **interrupt**.
- This interrupt is captured by the OS.
- The OS then **moves the process from Blocked → Ready**.

🧩 **Real-World Analogy:**
You send a document to print (Blocked state). The printer finishes, and a **"Print Complete"** popup (interrupt) appears, letting you return to your task. The printer (I/O subsystem) **notifies** your OS it's done.

Q28 –

In the **Ready → Running** transition:

- The process is already loaded in **main memory**, prepared with all required resources.
- The CPU just **schedules** the process to run next.
- There's **no swapping or I/O wait** involved.

🧩 **Real-World Analogy:**
Think of a sprinter already at the start line (Ready). The whistle blows, and they start running instantly. No gear changes, no warm-up, just **immediate action**.

✅ So, **C is correct** because this transition:

- Requires **no memory fetch** or secondary storage access.
- Involves only **context switching and CPU assignment**, making it **fastest** among the listed transitions.

Q29 –

Such a process is in the **Blocked** state. During this time:

- The OS **suspends its execution**.
- CPU cycles are **freed up** and can be given to another process.

🧩 **Real-World Analogy:**
Imagine you're waiting for someone to reply to a message. Instead of sitting idle, you do something else (like scrolling Instagram). That's what the CPU does — gives time to another task while the first one waits.

✅ So, **B is correct** because:

- A **Blocked** process is not wasting CPU time.
- This ensures **efficient CPU utilization** by letting others run while it waits.

Q30 –

Adaptive scheduling algorithms:

- Adjust **dynamically** based on process behavior.
- Can **favor I/O-bound processes**, which are usually **shorter** and more **responsive**.
- Prevent **CPU-bound processes** from monopolizing resources.

🧩 **Real-World Analogy:**
Think of a cashier line. If quick customers (I/O-bound) are always stuck behind large orders (CPU-bound), the line slows down. A smart cashier calls quick customers to a **fast lane** — that's adaptive scheduling in action.

✅ So, **D is correct** because:

- It helps balance performance.
- Reduces wait times for **short, interactive tasks**.
- Ensures **fairness** and prevents bottlenecks.

Q31 –

The **FAT (File Allocation Table)** system stores the entire file allocation table in memory (RAM), allowing the system to:

- **Quickly locate any cluster**, even if scattered.
- Reduce disk seeks for each access.

This setup explains how the system can **tolerate non-contiguous files** but still **perform fast lookups**.

🧩 **Real-World Analogy:**
Imagine looking for items in a warehouse. Even if items are spread out (non-contiguous), you have a **digital map** on your tablet (cached FAT) that tells you exactly where to go — no time wasted.

✅ So, **C is correct** because:

- FAT's **in-memory cluster mapping** enables quick access.
- Even with minor delays from non-contiguity, lookup remains efficient.

Q32 –

- 2 clusters × 512 bytes = 1024 bytes allocated.
- File size = 980 bytes → Waste = 1024 − 980 = **44 bytes**.
- Since the wasted space is within the last cluster and **can't be used for other files**, it's called **slack space** (or internal fragmentation).

🧩 **Real-World Analogy:**
Imagine packing a 980 ml drink into two 512 ml bottles. You use **1024 ml** worth of bottle capacity, but you only needed **980 ml**. The leftover **44 ml of unused space** is slack — wasted but unavoidable in that packaging system.

✅ So, **C is correct** because:

- The **44-byte difference** is slack.
- It represents **unused cluster space** in the last cluster of the file.

Q33 -

- In **FAT32**, the **Boot Sector** includes a **Flags field**, which specifies:
  - Which FAT copy is active (for use).
  - Whether the system should alternate or stick to one FAT.
- This dynamic selection is **not available** in FAT12/16, which default to **always using the first FAT**.

🧩 **Real-World Analogy:**
Think of FAT32 as having two backup batteries. The **control panel (Boot Sector)** tells the device which battery (FAT copy) to use. Unlike older models (FAT12/16), it can **switch dynamically** based on the situation.

✅ So, **A is correct** because:

- FAT32's **Boot Sector** manages which FAT copy is used.
- This offers **greater fault tolerance** and flexibility than FAT12/FAT16.

Q34 –

Linked allocation requires scanning each block to reach the desired one, which is inefficient for random access.
FAT (File Allocation Table) solves this by:

- Keeping a **centralized allocation table** in memory.
- This table holds pointers to the next cluster in the chain for each file.
- Since it's **cached in RAM**, the system can **quickly look up any cluster**, enabling efficient random access.

🧩 **Real-World Analogy:**
Imagine reading a book where each page tells you the number of the next page — you'd have to flip through one at a time.
Now imagine having a **table of contents** where you can instantly find the page number you need — that's the FAT.

✅ **Why C is correct:**

- It directly addresses the linked allocation weakness.
- RAM-cached tables enable **efficient, non-linear data access**.

Q35 -

- The **FAT table acts like a map** that holds pointers for each cluster.
- When a file starts at cluster X, the OS checks the FAT table entry at X to see the **next cluster number**, and so on.
- This chain continues until it reaches a special **end-of-file marker** in the FAT table.

🧩 **Real-World Analogy:**
It's like navigating using a **treasure map** where the first clue leads to the next clue, and the FAT table holds all these clues in memory.

✅ **Why C is correct:**

- The FAT table's pointer-based chaining lets the system **track clusters step-by-step**, without needing to store full sequences elsewhere.

Q36 -

1. **Determine required clusters:**
   - Each cluster = 512 bytes
   - 3 clusters = 3 × 512 = **1,536 bytes**
   - 1,536 is the smallest multiple of 512 that fits 1,400
2. **Calculate slack space:**
   - 1,536 − 1,400 = **136 bytes of slack**

🧩 **Real-World Analogy:**
Think of packing 1,400 ml of liquid into bottles that hold 512 ml each:

- You'd need 3 bottles (3 × 512 = 1,536 ml total).
- You'll end up with **136 ml of unused space**, which is slack.

✅ **Why C is correct:**

- 3 clusters are needed.
- The **extra space** in the last cluster (136 bytes) is wasted as **slack**, since it can't be used by other files.

Q37 –

FAT16 directory entries **do not support user-level security features**. The metadata typically includes:

- File name and extension
- Starting cluster number
- File size in bytes
- Creation/modification timestamps

However, FAT16 was designed as a **simple file system** (common in USB drives and memory cards) without built-in user permissions.

🧩 **Real-World Application:**
Think of an old flash drive you plug into a Windows 98 system — you could see or modify any file without user login or access control.

✅ **Why C is correct:**

- **Ownership and permission levels** are features of **modern file systems** like NTFS or ext4 — **not FAT16**.

Q38 –

In FAT12, FAT16, and FAT32:

- Cluster 0 and cluster 1 are **reserved**
  - Cluster 0 often used to hold media descriptor

    - Cluster 1 is typically marked as **end-of-chain**
- Actual file data begins at **cluster 2**

🧩 **Real-World Analogy:**
Imagine pages in a book where page 1 is a cover and page 2 is an index. The actual story starts from page 3 (cluster 2).

✅ **Why C is correct:**

- Every valid data cluster starts from **cluster number 2**, making this a technical truth about FAT architecture.

Q39 –

Larger clusters mean:

- Fewer clusters → fewer FAT entries → lower FAT overhead ✅
- But more **wasted space** at the end of files, especially small ones ❌
  This waste is called **internal fragmentation**, or **slack space**.

🧩 **Real-World Scenario:**
If every locker in your storage room holds 512 items, but most of your belongings are under 100 items in size, you'll have a lot of **unused space per locker** — that's **inefficiency**.

✅ **Why D is correct:**

- Larger cluster size = fewer entries ✅
- But increases **slack** → classic **internal fragmentation** ❌

Q40 –

This setup reflects how **FAT chains file clusters**:

- FAT[5] = 6 → file continues in cluster 6
- FAT[6] = end-of-chain → this is the last cluster

This means clusters 5 and 6 form a **file's data chain**.

🧩 **Real-World Analogy:**
Imagine a scavenger hunt where clue 5 says "Go to clue 6," and clue 6 says "This is the end." This means both clues (clusters) are part of the journey (file content).

✅ **Why B is correct:**

- This is **standard FAT chaining** behavior: files are made up of **linked clusters**, and the table keeps the links.