Operating Systems Reviewer

What is Operating system?

A software program that manages computer hardware and provides services for computer programs.

▪ It acts as an intermediary between the hardware and software, enabling the hardware to communicate with the software and vice versa.

▪ The history and evolution of operating systems have been shaped by technological advancements and changing user needs.

A Computer System consists of:
1. Users (people who are using the computer)
2. Application Programs (Compilers, Databases, Games, Video player, Browsers, etc.)
3. System Programs (Shells, Editors, Compilers, etc.)
4. Operating System (A special program which acts as an interface between user and hardware)
5. Hardware (CPU, Disks, Memory, etc)

Types of Operating Systems

Real-time operating systems (RTOS): These operating systems are designed to manage real-time applications, where tasks need to be completed within strict time constraints.
Examples include VxWorks and FreeRTOS.

Network operating systems: These operating systems are designed to manage network resources and provide services to clients over a network.
Examples include Windows Server, Linux, and Novell NetWare.

Types of Operating Systems
Mobile operating systems: These operating systems are designed for smartphones, tablets, and other mobile devices.
Examples include Android (Google), iOS (Apple), and Windows Phone (Microsoft).

Embedded operating systems: These operating systems are designed to run on embedded devices, such as appliances,

automobiles, and industrial machines.
Examples include Embedded Linux, Windows Embedded, and VxWorks.

Types of Operating Systems
Server operating systems: These operating systems are designed to run servers and provide network services.
Examples include Windows Server, Linux Server, and UNIX-based server operating systems.

Multiprocessor operating systems: These operating systems are designed to support multiple processors running in parallel.
Examples include Windows Server, Linux, and UNIX.

Types of Operating Systems
Distributed operating systems: These operating systems are designed to run on a distributed network of computers, sharing resources and coordinating tasks.
Examples include Amoeba, Plan 9, and Inferno.

Standalone operating systems: These operating systems are designed to run on a single computer.
Examples include Windows, macOS, and various distributions of Linux (Ubuntu, Fedora, CentOS).

OS ARCHITECTURE AND COMPONENTS

Kernel: The core component of the operating system that manages the system's resources such as memory, CPU, and input/output devices.
The kernel provides essential services to other parts of the operating system and ensures that programs can run smoothly.
Device drivers: These are software components that allow the operating system to communicate with hardware devices such as printers, scanners, and network cards.
Device drivers convert high-level commands into instructions that the hardware can understand.

OS ARCHITECTURE AND COMPONENTS
File system: The file system is responsible for managing files and directories on storage devices such as hard drives and SSDs.
It organizes and stores data in a structured manner to make it easily accessible to users and applications.
4. User interface: The user interface allows users to interact with the operating system and applications through visual elements such as

windows, icons, and menus.
There are different types of user interfaces, including command-line interfaces (CLI) and graphical user interfaces (GUI), each with its own advantages and disadvantages.

EXAMPLES OF OS ARCHITECTURE
Windows operating system has a layered architecture with a kernel at its core that manages the system resources. It provides device drivers for various hardware components and a file system to organize and store data.
Windows also has a graphical user interface that allows users to interact with the system through visual elements.
There are different types of user interfaces, including command-line interfaces (CLI) and graphical user interfaces (GUI), each with its own advantages and disadvantages.

EXAMPLES OF OS ARCHITECTURE
Linux is an open-source operating system that follows a modular architecture with a monolithic kernel.
It provides device drivers for a wide range of hardware devices and supports multiple file systems.
Linux can be customized and modified to suit different requirements, making it popular among developers and system administrators.

EXAMPLES OF OS ARCHITECTURE
macOS is the operating system developed by Apple for its Macintosh computers. It has a layered architecture like Windows, with a kernel managing system resources and a file system for organizing data.
macOS has a user-friendly graphical user interface known for its sleek design and ease of use.

PROCESS MANAGEMENT
It is a crucial aspect of operating systems that involves managing the execution of processes, which are instances of executing programs.
The primary goals of process management are to ensure efficient utilization of system resources, provide a fair and secure environment for all processes, and facilitate communication and synchronization between processes.

Process control block (PCB)
Process Control Block (PCB) is a data structure used by the operating system to manage information about a process. The process control keeps track of many important pieces of information needed to manage

processes efficiently.
It contains information about the process, i.e. registers, quantum, priority, etc.

Process scheduling
Process scheduling involves deciding which process should be executed next on the CPU.
The operating system uses scheduling algorithms to manage the execution of multiple processes efficiently.
Some common scheduling algorithms include First-Come-First-Serve (FCFS), Round Robin, Shortest Job Next (SJN), and Priority-based scheduling.

Process synchronization
Process synchronization is critical for ensuring that processes cooperate and communicate effectively.
The operating system provides mechanisms, such as semaphores, mutexes, and condition variables, to allow processes to synchronize their execution and access shared resources.

Remote Procedure Call (RPC) is a software communication protocol that one program uses to request a service from another program located on a different computer and network, without having to understand the network's details.
RPC is used to call other processes on remote systems as if the process were a local system.

The names of the states are not standardized although the process may be in one of the following states during execution.

1. New: A program which is going to be picked up by the OS into the main memory is called a new process.

2. Ready: Whenever a process is created, it directly enters in the ready state, in which, it waits for the CPU to be assigned.
The OS picks the new processes from the secondary memory and put all of them in the main memory

3. Running: One of the processes from the ready state will be chosen by the OS depending upon the scheduling algorithm.

4. Block or wait: From the Running state, a process can make the transition to the block or wait state depending upon the scheduling algorithm or the intrinsic behavior of the process.

When a process waits for a certain resource to be assigned or for the input from the user then the OS move this process to the block or wait state and assigns the CPU to the other processes.

5. Completion or termination: When a process finishes its execution, it comes in the termination state. All the context of the process (Process Control Block) will also be deleted the process will be terminated by the Operating system.

6. Suspend ready: A process in the ready state, which is moved to secondary memory from the main memory due to lack of the resources (mainly primary memory) is called in the suspend ready state.
If the main memory is full and a higher priority process comes for the execution, then the OS must make the room for the process in the main memory by throwing the lower priority process out into the secondary memory

7. Suspend wait: Instead of removing the process from the ready queue, it's better to remove the blocked process which is waiting for some resources in the main memory.
Since it is already waiting for some resource to get available hence it is better if it waits in the secondary memory and make room for the higher priority process.
These processes complete their execution once the main memory gets available and their wait is finished.

History of Operating Systems

1940s: Early systems (ENIAC, UNIVAC) – no OS.

1950s–1960s: First OS (GM-NAA I/O, Atlas Supervisor).

1960s: Time-sharing OS (CTSS at MIT).

1970s: Personal/minicomputers (CP/M, UNIX).

1980s: GUIs (Apple Macintosh, Windows).

1990s–2000s: Open-source OS (Linux, BSD); mobile OS (Android, iOS).

Types of Operating Systems

Real-time OS (VxWorks, FreeRTOS)

Network OS (Windows Server, Linux, Novell NetWare)

Mobile OS (Android, iOS, Windows Phone)

Embedded OS (Embedded Linux, Windows Embedded)

Server OS (Windows Server, UNIX, Linux Server)

Multiprocessor OS

Distributed OS (Amoeba, Plan 9, Inferno)

Standalone OS (Windows, macOS, Linux distros)

OS Architecture & Components

Kernel: Core, manages resources.

Device Drivers: Interface with hardware.

File System: Organizes/stores data.

User Interface: CLI / GUI.

Examples:

Windows – layered, GUI-based.

Linux – modular, monolithic kernel.

macOS – layered, sleek GUI.

EARLY MEMORY MANAGEMENT

1. Single-User Contiguous Scheme

- Simplest scheme: whole memory given to one user/process.
- No multiprogramming; OS resides in the lowest memory portion.
- Example:
    - 64 KB total memory
    - 4 KB → OS
    - 60 KB → one program
- To run another program → stop current, reload new one.
- No fragmentation (only one process).

---

2. Fixed Partition Scheme

- Memory divided into fixed-size partitions at boot time.
- Each partition holds one process.
- Internal fragmentation occurs when process size < partition size.
- Example: 64 KB RAM → 4 partitions of 16 KB.
    - A (10 KB) → 6 KB wasted
    - B (12 KB) → 4 KB wasted
    - C (16 KB) → perfect fit
    - D (20 KB) → cannot load
- Simple but wasteful.

---

3. Dynamic Partitions

- Memory allocated exactly as process requires.
- Reduces internal fragmentation but leads to **external fragmentation**.

Internal Fragmentation

- Wasted space inside a fixed partition.
- Example: Partitions of 16 KB, processes of 10 KB, 12 KB, 16 KB.
    - Allocated: 48 KB
    - Used: 38 KB
    - Waste: 10 KB

External Fragmentation

- Free blocks scattered, not contiguous.
- Example: Free blocks = 6 KB, 4 KB, 10 KB → total 20 KB.
- New process needs 14 KB → can’t fit in one block, though total is enough.

---

4. Best-Fit and First-Fit Allocation

- **First-Fit**: allocates first free block large enough. Faster.
- **Best-Fit**: allocates smallest sufficient block. Less waste, slower.

Example: Free blocks = 5 KB, 10 KB, 20 KB, 15 KB.

- Process: 12 KB.

    - First-Fit → 20 KB block (8 KB left).
    - Best-Fit → 15 KB block (3 KB left).

---

5. Deallocation

- When process finishes, memory is freed.
- Can create **holes** (external fragmentation).
- Example: Process A (10 KB) ends → leaves 10 KB hole.

---

6. Relocatable Dynamic Partitions

- Processes can be moved/compacted to merge free blocks.
- Requires relocation hardware.
- Example:
    - Free blocks = 5 KB + 10 KB → total 15 KB, but not contiguous.
    - New process needs 14 KB.
    - With compaction → merge into 15 KB block → process fits.

Summary Table

Scheme	Fragmentation	Multiprogramming	Limitation
Single-User Contiguous	None	No	Only one process
Fixed Partition	Internal	Yes	Partition size limit
Dynamic Partitions	External	Yes	External fragmentation
Best-Fit Allocation	Minimal internal	Yes	Slower search
First-Fit Allocation	More waste	Yes	Faster, less efficient
Deallocation	External (holes)	Yes	Needs compaction
Relocatable Dynamic Partitions	Minimal	Yes	Needs relocation hardware

Introduction to Memory Management

Memory management controls computer memory allocation and deallocation, ensuring efficient use and system stability. It is crucial for running multiple processes and supporting virtual memory functionality.

Efficiently allocate and deallocate memory, maximize system performance, ensure process isolation, prevent fragmentation, and provide seamless virtual memory support for multitasking environments.

Memory allocation types include static, dynamic, and automatic. Static allocates fixed memory during compile-time, dynamic allocates at runtime, and automatic manages memory within function scopes.

What is Virtual Memory?

Virtual memory is a memory management technique that allows a computer to compensate for physical memory shortages by temporarily transferring data from RAM to disk storage, creating an illusion of a larger memory.

Virtual vs Physical Memory

Virtual memory extends physical memory by using disk space, enabling larger address spaces and multitasking. Physical memory is actual RAM, faster but limited in size compared to virtual memory.

Benefits of Virtual Memory

Virtual memory increases available memory, enables multitasking, isolates processes for security, and allows efficient use of physical memory by loading only necessary data into RAM.

Paged Memory Allocation

Paging Concept and Mechanism

Paging divides memory into fixed-size pages, mapping them to physical frames through page tables, enabling efficient and flexible memory allocation without external fragmentation.

Page Table Structure
The page table maps virtual pages to physical frames, storing frame addresses and status bits like valid, dirty, and access permissions to manage memory efficiently and support virtual memory operations.

Advantages and Disadvantages of Paging
Paging reduces fragmentation and allows efficient memory use but can introduce overhead, increased complexity, and slower access due to page table lookups.

Address Translation in Paging
Paging divides memory into fixedsize pages; logical addresses are split into page numbers and offsets, which the page table translates to physical addresses during address translation.

Page Size and Its Impact
Page size affects internal fragmentation and table size. Larger pages reduce page table entries but increase wasted space; smaller pages improve memory use but require larger page tables and more overhead.

Internal Fragmentation Issues
Paged memory allocation can cause internal fragmentation due to fixed-size pages, where the last page of a process may not be fully utilized, wasting memory space within allocated pages.

Implementation of Paged Memory
Address Translation in Paging
Paging divides memory into fixedsize pages; logical addresses are split into page numbers and

offsets, which the page table translates to physical addresses during address translation.

Page Size and Its Impact
Page size affects internal fragmentation and table size. Larger pages reduce page table entries but increase wasted space; smaller pages improve memory use but require larger page tables and more overhead

Internal Fragmentation Issues
Paged memory allocation can cause internal fragmentation due to fixed-size pages, where the last page of a process may not be fully utilized, wasting memory space within allocated pages.

Introduction to Demand Paging

Concept and Working Principle
Demand paging loads pages into memory only when referenced, reducing memory use. It uses page faults to fetch pages from secondary storage, optimizing system performance and resource allocation.

Page Fault Handling
Page fault handling occurs when a requested page is not in memory, triggering loading from disk, updating page tables, and resuming process execution to ensure efficient memory use.

Advantages of Demand Paging
Demand paging reduces initial load time and memory usage by loading pages only when

needed, improving system responsiveness and allowing efficient use of physical memory resources.

Demand Paging Operation
Steps in Demand Paging
Demand paging loads pages into memory only when referenced, reducing memory use. Steps include page fault detection, locating the page on disk, loading it into a free frame, updating tables, and resuming execution.

Copy-on-Write Mechanism
Copy-on-Write delays copying shared pages until a modification occurs, optimizing memory use by allowing multiple processes to share pages initially, reducing unnecessary duplication.

Performance Considerations
Demand paging improves memory utilization but may cause page faults, leading to increased latency. Performance depends on fault rate, replacement algorithms, and disk access speed.

Page Replacement Algorithms
FIFO (First-In-First-Out)
FIFO replaces the oldest page in memory first, leading to possible suboptimal performance due to lack of consideration for page usage frequency or recency.

LRU (Least Recently Used(
LRU (Least Recently Used)

replaces the page that has not been used for the longest time, effectively minimizing page faults by leveraging recent access history to optimize memory utilization.

Optimal Page Replacement
Optimal Page Replacement replaces the page that will not be used for the longest time, minimizing page faults but requiring future knowledge, making it ideal yet impractical for real systems.

Concepts in Page Replacement
Thrashing and Its Causes
Thrashing occurs when excessive page faults cause frequent swapping, severely degrading system performance. It is often caused by insufficient memory allocation or running too many processes simultaneously.

Page Replacement Performance Metrics
Page replacement performance is evaluated by metrics like page fault rate, effective memory access time, and overhead. Lower page faults and overhead indicate better replacement policy efficiency.

Page Replacement Performance Metrics
Page replacement performance is evaluated by metrics like page fault rate, effective memory access time, and overhead. Lower page faults and overhead indicate better replacement

policy efficiency.

Basics of Segmentation

Segmentation Concept

Segmentation divides memory into variable-sized segments based on logical units like functions or data structures, enhancing memory protection and sharing by mapping each segment independently in virtual memory.

Segment Table and Address Translation

Segmented memory uses a segment table storing base and limit for each segment. Logical addresses translate by adding offset to the segment base, ensuring address validity via segment limit checks.

Advantages of Segmentation

Segmentation improves memory utilization, simplifies protection by isolating segments, supports modular programming, and enables dynamic sharing and growth of segments without external fragmentation

Segmentation in Practice

Protection and Sharing in Segmentation

Segmentation enables fine-grained protection by assigning access rights to each segment, facilitating controlled sharing among processes while preventing unauthorized access and enhancing memory security.

Segmentation vs Paging

Segmentation divides memory into variable-sized segments, enhancing logical division and protection, while paging uses fixed-size pages, simplifying allocation but potentially causing internal fragmentation.

Examples and Use Cases

Segmented memory allocation is used in modern operating systems for efficient code modularity, supporting dynamic loading, and protection by isolating segments such as stack, heap, and code during execution.Examples and Use Cases
Segmented memory allocation is used in modern operating systems for efficient code modularity, supporting dynamic loading, and protection by isolating segments such as stack, heap, and code during execution.

Flexibility and Efficiency
Provides flexibility by allowing dynamic memory allocation and efficient use of physical memory; however, managing segments or pages adds overhead and complexity to memory management.

Complexity in Management
Managing segmented and demand-paged memory increases complexity due to tracking multiple segments and pages, handling fragmentation, and ensuring efficient page replacement and segment protection mechanisms.

Real-world Implementations
Real-world implementations of segmented/demand-paged memory improve efficient memory use and process isolation but face challenges like increased complexity, fragmentation, and higher overhead for managing dynamic allocation.

Understanding Cache Memory
Cache Memory Concept
Cache memory is a small, fast storage located close to the CPU that stores frequently accessed data, improving system speed

and efficiency by reducing access time to main memory in virtual memory management.

Types of Cache (L1, L2, L3)
Cache memory enhances virtual memory performance by storing frequently accessed data. Ll is fastest and smallest, L2 is larger with moderate speed, and L3 is shared among cores, improving overall efficiency.

Cache Mapping Techniques
Cache mapping techniques determine how data is stored and accessed in cache, including direct, associative, and setassociative mapping, optimizing speed and efficiency in virtual memory management.

Cache Memory Interaction with Virtual Memory

Translation Lookaside Buffer (TLB)
The Translation Lookaside Buffer (TLB) caches recent virtual-to-physical address translations, significantly speeding up memory access by reducing the need for repeated page table lookups in virtual memory systems.

Cache Coherence and Consistency
Cache coherence ensures data uniformity across multiple cache levels, while consistency guarantees synchronized updates between cache and virtual memory, preventing stale or conflicting data during memory access.

Performance Impact of Cache on Virtual Memory Systems
Cache memory reduces latency by storing frequently accessed virtual pages, improving virtual memory performance. However, cache misses can increase page faults, slightly impacting overall system efficiency.

Dead-Locks Locks in in
Operating Systems

What is a Dead-Lock ?
Deadlock is a situation where a set of processes are blocked because each process is holding a resource and waiting for another resource acquired by some other process.

Conditions for Deadlock in OS
Mutual-Exclusion
When a Process is Accessing a Shared Variable, the Process is said to be in a CRITICAL SITUATION

One Process at a time can Use a resource.

Hold and Wait

A Process holding at least one resource is waiting to acquire additional resources held by another processes.

Printer needs scanned copy to Print and scanner wants a copy to scan.

No Preemption
A Resource can be released only voluntarily by the process holding it, after that process has completed its task.

Number of resources can be feasibly removed from a processor holding it.

Resource-Allocation Graph

If Graph contains no cycles
→ No Deadlock
If graph contains a Cycle
→ If only one instance per resource type, then deadlock.
→ If several Instances per resource type, possibility of

dead lock.

RECOVERY FROM DEADLOCK: RESOURCE PREEMPTION
Selecting a victim - minimize cost
Rollback - return to some safe state, restart process for that state
Starvation - same process may always be picked as victim, include number of rollback in cost factor

PROCESS MANAGEMENT CONCEPTS
ˆ Concept of a process
ˆ Terminology
ˆ Job (also known as program) is an inactive unit such as a file in disk
ˆ This entity contains at least two types of elements: program code and a set of data
ˆ Process (also called task) is an executed program code on the processor, that is, a part of an active entity
ˆ a Thread is a portion of a process that can be run independently
ˆ In multiprogramming environment, the processor is used by multiple programs/processes
ˆ Process manager needs to arrange the execution of the programs/processes (to make a schedule for them) to promote the most efficient use of resources (including hardware and software) and to avoid competition and deadlock

Job and Process Scheduling
ˆ Schedulers
ˆ Job scheduler
ˆ Initialise each job
ˆ Only concerned with selecting jobs from a queue of incoming jobs
ˆ Places them in a process queue (READY queue)
ˆ Process scheduler
ˆ Determines which jobs get the CPU, when, and for how long.
ˆ Also decides when processing should be interrupted
ˆ decides when each step will be executed
ˆ recognises when a job has concluded and should be

terminated
ˆ Hierarchy
ˆ Job scheduler is the high-level scheduler
ˆ Process scheduler is the low-level scheduler

Process Scheduler
ˆ Common trait among most computer programs: they alternate between CPU cycles and IO cycles
ˆ Process scheduler is a low-level scheduler that assigns CPU the execute the processes of those jobs placed in the ready queue by the job scheduler.
ˆ Although CPU cycles vary from program to program, there are some general types of jobs:
ˆ IO-Bound jobs e.g. printing a series of documents. Many brief CPU cycles and long IO cycles.
ˆ CPU-Bound jobs e.g. finding the first 300 prime numbers. Long CPU cycles and shorter IO cycles.
ˆ Total statistical effect approximates a Poisson distribution.
ˆ Highly interactive environment has a third level - begin a middle-level scheduler - finding it advantagious to remove active jobs from memory; allowing jobs to complete faster

PROCESS STATUS
Two-state model
ˆ Process queue
ˆ Processes are divided into threads
ˆ The threads form a "ready queue" waiting for being processed in the processor
ˆ Two-state model
ˆ A process is in "running" state if it is executed on the processor
ˆ It is in "not running" state otherwise

PROCESS STATUS
ˆ Five-state model
ˆ States
ˆ Running: the process is currently executed on the processor
ˆ Ready: a process is prepared to execute
ˆ Waiting: a process waits for resources

ˆ Hold: a process is just created
ˆ Exit: a process is released from pool of executable programs by OS
ˆ State transitions:
1 HOLD to READY:
ˆ When OS is prepared to take an additional process. Initiated by job scheduler; availability of memory and devices checked.
2 READY to RUNNING:
ˆ Process scheduler chooses a ready process to execute.
3 RUNNING to EXIT:
ˆ Currently executed process has completed or a run time error. Initiated by job or process scheduler.
4 RUNNING to WAIT:
ˆ Handled by process scheduler, initiated by job instruction for I/O or other resources request.
5 WAIT to READY:
ˆ Handled by process scheduler, initiated when I/O or other resources become available.
6 RUNNING to READY:
ˆ Handled by process scheduler; maximum allowable time is reached or other criterion e.g. a priority interrupt.

Process control block (PCB): Any process is uniquely characterised by a list of elements:
ˆ Identifier: a unique identifier which distinguishes a process from others
ˆ State: a process can have different states
ˆ Priority: this shows the priority level of a process relative to other process
ˆ Program counter: the address of the instruction to be executed next of a process (program code)
ˆ Memory pointers: include those point to program code and data associated of a process, plus any memory blocks shared with other processes
ˆ Context data: These are date that are presented in registers in the processor while a process is running
ˆ I/O status information: includes I/O requests and the availability of files and I/O devices
ˆ Accounting information: includes time and number countered

Process State
ˆ This contains all of the information needed to indicate the current state of the job such as:

process status word current instruction counter and register contents when a job isn't running.
register contents if the job has been interrupted and is waiting.
main memory information including the address where the job is stored.
resources information about all resources allocated to the job, being hardware units or files.
process priority used by systems with priority scheduling algorithm

Accounting
ˆ Information for billing purposes and performance measurement. Typical charges include:
ˆ Amount of CPU time used from beginning to end of execution.
ˆ Total time the job was in the system until exited.
ˆ Main storage occupancy - how long the job stayed in memory until it finished execution.
ˆ Secondary storage used during execution.
ˆ System programs used (compilers, editors, utilities, etc.)
ˆ Number and type of IO operations, including IO tranmission time (includes use of channels, control units, devices)
ˆ Time spent waiting for IO completion
ˆ Number of input records read, and number of output records written

Process scheduling algorithms
FCFS - First-come-first-served
ˆ Batch environment
ˆ Jobs/processes arrive at the times that are too close to one another
ˆ Those jobs/processes are collected in batches
ˆ First-come-first-serve (FCFS)
ˆ A READY queue for storing ready processes that are initialised by Job scheduler
ˆ When a process is in RUNNING state, its execution will be completed in one go, that is, these is no waiting state in this algorithm
ˆ Average turnaround time (τ)
ˆ Total time needed for every process completed divided by the number of processes in the queue
ˆ Depends on how the processes are queued in the READY queue

Process scheduling algorithms
SJF - Shortest job First
ˆ Shortest job first [next] (SJF) [SJN]
ˆ Process that uses the shortest CPU cycle will be selected for running first

Process scheduling algorithms
SJF - Shortest job First
ˆ Optimal in terms of occupying the least CPU time

Process scheduling algorithms
SRT - Shortest remaining time
ˆ Real-time environment
ˆ Shortest remaining time (SRT)
ˆ Process that the mostly close to complete will be process first, and even this process can be pre-empted if a newer process in READY queue has a shorter complete time

Priority scheduling
ˆ Allowing process with the highest priority to be processed first
ˆ The processing is not interrupted unless the CPU cycle of the process is completed or a natural wait occurs
ˆ If more than one process has the same priority, the one which joins the ready queue first is processed first
ˆ Priorities:
ˆ Memory requirements - jobs requiring large amounts of memory could be allocated lower priorities than those requiring small amounts of memory
ˆ Number and type of device - jobs requiring many devices would be allocated lower priorities
ˆ Total CPU time - jobs having long CPU cycle, or estimated run time, would be given lower priorities
ˆ Amount of time already spent in the system - some systems increase priority of jobs that have been in the system for an unusually long time. This is known as aging

process scheduling algorithms - comparison
ˆ FCFS - nonpreemptive - batch - unpredictable turnaround - Easy to implement
ˆ SJN - nonpreemptive - batch - indefinite postponement - minimise average waiting
ˆ Priority scheduling - nonpreemptive - batch - indefinite postponement - fast completion of important jobs

ˆ SRT - Preemptive - batch - overhead from context switching - fast completion of short jobs
ˆ Round robin - Preemptive - interactive - requires selection of good time quantum - fair CPU allocation and reasonable response times for interactive users
ˆ Multiple-level queues - Preemptive/nonpreemptive - batch/interactive - overhead from monitoring queues - flexible, counteracts indefinite postponement with aging, fair treatment of CPU-bound jobs by incrementing time quantums on lower priority queues

Preemptive Scheduling
In preemptive scheduling, the operating system can interrupt a running process to allocate the CPU to another process usually due to priority rules or time-sharing policies.A process may be moved from Running → Ready state before it finishes.

Non-Preemptive Scheduling
In non-preemptive scheduling, once a process starts using the CPU, it runs until it finishes or moves to a waiting state. The OS cannot forcibly take away the CPU.