What is an Operating System (OS)?

An Operating System is a software that:

- Manages hardware (CPU, memory, devices)
- Acts as a bridge between the user & the hardware.
- Provides a platform for running application programs.

imp

Define OS & what does OS do?

→ An operating system is a software that manages hardware and runs applications, acting as a bridge between the user and the computer hardware.

Draw the abstract view of the components of a computer system.

Place of OS in a computer:

fig: Abstract view of the components of a computer system.

CamScanner

imp

Functions of OS:/Services of OS:

i. Role – Government

ii. Bridge between the Hardware & Software

P M F } iii. Process Management

iv. Memory Management

I P S } v. File Management

vi. Input-output Management

vii. Device Management

viii. Security Management

ix. Error Handling.

x. Parallel Processing

PC computer system:

fig: A typical PC computer System.

Utility Software:

is a type of software which is used to faster the computer function.

Ex:
- Antivirus.
- Disk Clean up
- File Compression
- Backup Software

CamScanner

How an OS is initially Loaded into a computer?

→ 1. When the computer is powered on, the BIOS/UEFI runs first (stored in ROM)

2. BIOS runs a small program called the bootstrap loader.

3. The Bootstrap loader locates the OS on the hard disk.

4. It then loads the OS into RAM. (main memory)

5. Once loaded, the OS starts running and takes control of the system.

What will happen if there is no OS on a PC?

If there is no operating system on a computer:

1. The computer won't start properly – it will stop after showing the BIOS screen.

2. We can't run any programs.

3. The system can't communicate with hardware.

4. No user interface.

5. The computer will show an error like "No Bootable Device".

P.T.O

CamScanner

Types of Operating System

* What is a Job?

→ A job is a set of instructions given to the OS to execute. It may be a small program or a large task.

* Uniprocessor System:

Only one job is processed at a time, and all system resources are used by that one job until it's done.

imp

Batch Operating System:

- ~~These jobs are~~ Jobs with simillar needs group into batches.
- These jobs are processed one by one, without direct user interaction.
- A operator sort jobs with similar needs.

fig: Batch Operating System.

Advantages of Batch OS:

- Supports multiple users.
- Less idle time (efficient for grouped jobs)
- Suitable for repeating large task.

Ex: Payroll system, Bank Statement processing.

Disadvantages of Batch OS:

- CPU wastes time during I/O.
- If one job fails, other must wait.
- Slow response time as jobs run one by one.

CamScanner

Multiprogramming Operating System:

- More than one program is kept in main memory at the same time.
- CPU switches between programs to make best use of time.

fig: Multi-Programming

Advantages:

- Better CPU utilization
- Faster System performance
- Less response time.

imp

* Multi-Tasking / Time-sharing Operating System:

- It is a type of multiprogramming OS with every process running in round robin manner.
- Each task is given a fixed time (quantum).
- After the time ends, CPU switches to the next task.
- Used in systems with multiple users or programs running at once.

Advantages of Time-Sharing OS:

- Equal CPU time for each task
- Low CPU idle time
- Efficient resource sharing
  - share hardware resources: CPU, memory, peripherals reducing cost & increase efficiency.
- Better productivity.
  → concurrent work, reduce waiting time.
- Real-time interaction

Disadvantages:

- Security risks. (multiple users sharing)
- Complex to manage
- Higher system overhead due to, scheduling, context switching etc.
- Bugs & errors possible.
- Data communication issues

CamScanner

Example:

- IBM VM/CMS: Virtual machine time-sharing.
- TSO (IBM system/360): Mainframe multi-user OS
- Windows Terminal Services: Remote access to shared apps/resources

Multiprocessing Operating System:

- Uses multiple CPU's at the same time.
- Executes processes in parallel for better performance.

Advantages:

→ Faster execution (higher throughput)

→ Fault-tolerant if one CPU fails, other still work.

fig: Multiprocessing

Multi-User Operating System:

- Allows multiple users to use the system at the same time.
- Can be single-processor (via time-sharing) or multi-processor

fig: TS OS

imp

Distributed Operating System: (loosely coupled systems)

- Multiple independent computers (nodes) connected through a network
- Each has its own CPU & memory but work together.
- Enables remote access to files & software.

Architecture of Distributed OS

fig: Distributed OS

Advantages:

1. No single point of failure (one system fails, other work).
2. Fast data exchange (email, network).
3. Load is shared, improves performance
4. Scalable - easily add more systems.
5. Less delay in processing.

Example: LOCUS.

Disadvantages:

1. If network fails, all communication stops.
2. Expensive & complex setup.
3. Security risks in message passing over public networks.
4. Co-ordination is difficult (scheduling, deadlock handling) across nodes
5. Delay: Networking causes delays in the data transfer & it's difficult to know the correct chronological order.

CamScanner

imp

Networking Operating System: (tightly coupled system)

- Runs on servers to manage data, users, security and shared devices like printers overs a private network.
- Users are aware of each other and connected tightly (tightly couple)

fig: Networking OS.

Advantages:

- Highly stable central servers.
- Easy integration of new tech.
- Centralized security.
- Remote access possible.

Disadvantages:

- Expensive Servers.
- Central point of failure
- Regular maintenance needed.

Ex: Windows servers, UNIX, LINUX, MAC OSX, BSD, NetWare.

Real-Time Operating System (RTOS):

- Serve real-time services.
- The time required to process & respond to inputs is very small. (Respone time)
- Used when there are time requirements that are very strict like missile systems, air traffic, robots etc.

- **Types:** 1. Hard Real-Time Systems:
  - → Time constraints are very strict.
  - → Even the shortest possible delay is not acceptable
  - Ex: life saving like: airbag.
  - → Virtual memory rarely found here.

CamScanner

2. **Soft Real-time Systems:** These OSs are for applications where

→ Small delays acceptable (e.g., video streaming)

fig: Real-Time Operating System.

Advantages of RTOS:

- Fast Response time.
- Maximum Consumption
- Task shifting time less.
- Focus on application
- Good for embedded systems
- Error-free and efficient.
- Best memory handling

Disadvantages of RTOS:

- Runs limited tasks
- Needs high system resources.
- Complex to design.
- Needs custom drivers & hardware.
- Not good to set thread priority.

Example: Air traffic control, military systems, industrial robots, weapon systems, robots, scientific experiments etc.

P.T.O

CamScanner

Embedded Operating System:

- Built for specific devices, not general computers (e.g., elevators, microwave ovens).
- Allows the access of device hardware to the software that is running on the top of the OS.

Advantages:

- Fast as it is dedicated to a particular job
- Cheap.
- Use less memory & resources.

Disadvantages:

- Only one task at a time.
- Hard to upgrade or scale.

Mobile OS: for smartphones & tablets.

Ex: Android, iOS.

CamScanner

Kernel in Os

imp

Definition: The kernel is the core part of the operating system.

→ It is the first program loaded after the bootloader when the computer starts.

→ It acts as a bridge between applications and hardware.

fig: Kernel

Main functions of Kernel:

- Manages system resources like CPU, memory & I/O devices.
- Controls tasks like running programs, file access, and device management.
- Enables multitasking in a secure & stable way.
- Facilitates communication between hardware & user applications.
- Prevents unauthorized access to system resources.

imp

System Call: A system call is a mechanism that provides the interface between a process and the operating system.

→ It allows user programs to access OS services like file handling, process management etc.

→ When a process ~~wants to~~ makes a request to the kernel, then it is called System call.

CamScanner

How System calls work:

1. The user program executes a system call instruction.
2. This switches the CPU to kernel mode.
3. OS performs the requested task.
4. Control returns back to the program.

imp

When is a System Call Needed?

A system call is needed whenever a user program wants to access hardware or OS services.

Situations when system calls are needed:

1. To create or terminate a process
   Ex: fork(), exit().
2. To perform file operations.
   Ex: open, read, write, close a file
3. To access I/O devices.
   Ex: reading from keyboard, writing to display.
4. To request memory or manage it.
   Ex: allocate / free memory.
5. To establish communication between processes.
   Ex: creating pipes, sockets, shared memory.
6. To handle user permissions or security.
   Ex: login authentication, access control.

CS CamScanner

imP

Types of System Calls:

1. **Process Control**: Create, terminate or wait for processes.
   fork(), wait(), exit().
2. **File Management**: Open, read, write, close files.
   open(), read(), write().
3. **Device Management**: Request or release devices.
   ioctl(), read(), write().
4. **Information Maintenance**: Get system time, PID, etc
   getpid(), alarm(), sleep().
5. **Communication**: Send and receive messages.
   pipe(), shmget(), mmap().
6. **Protection**:
   chmod(), umask(), chown()

P}
P}
C}
C}
DM}
DM}
FM}
}

fig: System call.

CamScanner

Popular System Calls:

- fork() – Creates a new (child) process.
- wait() – Parent waits for child.
- exec() – Replaces current process with a new program. (overlay)
- exit() – Terminates a process.
- kill() – Sends a signal to terminate or control another process.

Previleged Instructions (OS): These are special CPU instructions that can only be executed in kernel mode (not by normal user programs).

→ They allow the OS to perform critical tasks like managing memory, devices and interrups.

Example of Previleged Instructions:

1. Handling Interrupts.
2. Switching from user mode to kernel mode.
3. I/O management (like accessing devices directly).

* Why Privileged Instructions are Needed?

→ They protect the system from user errors or malicious programs.

→ Prevent users from harming the OS or other running processes.

CS CamScanner

imp

Dual Mode operations in OS:

To ensure the proper execution of the operating system, there are two modes of operation:

1. User mode.
2. Kernel mode.

1. User mode:

- For running normal user applications like Word, browsers etc
- Can't execute privileged instructions.

2. Kernel mode:

- For running core OS functions like memory & device management.
- Can execute all instructions, including previleged ones.

Transition → Happens via system calls or interrupts.

When needed system switches from user mode → kernel mode, & then back.

fig: Dual mode operations in OS

CamScanner

Process in Operating System

A process is a program in execution. For example, when we write a program in C or C++ and compile it, the compiler creates binary code. The original code and binary code are both programs. When we actually run the binary code, it becomes a process.

- A process is an 'active' entity instead of a program, which is considered a 'passive' entity.
- A single program can create many processes when run multiple times; for example, when we open a .exe or binary file multiple times, multiple instances begin (multiple processes are created).

How Does a Process Look Like in Memory?

A process in memory is divided into several distinct sections, each serving a different purpose. Here's how a process typically looks in memory:

- **Text Section**: A text or code segment contains executable instructions. It is typically a read only section
- **Stack**: The stack contains temporary data, such as function parameters, returns addresses, and local variables.
- **Data Section**: Contains the global variable.
- **Heap Section**: Dynamically memory allocated to process during its run time.

CamScanner

Process Lifecycle

When you run a program (which becomes a process), it goes through different phases before its completion. These phases, or states, can vary depending on the operating system, but the most common process lifecycle includes two, five, or seven states.

Figure 3.2 Diagram of process state.

Here's a simple explanation of the five-state process model:

- **New:** This state represents a newly created process that hasn't started running yet. It has not been loaded into the main memory, but its process control block (PCB) has been created, which holds important information about the process.

- **Ready:** A process in this state is ready to run as soon as the CPU becomes available. It is waiting for the operating system to give it a chance to execute.

- **Running**: This state means the process is currently being executed by the CPU. Since we're assuming there is only one CPU, at any time, only one process can be in this state.

- **Blocked/Waiting:** This state means the process cannot continue executing right now. It is waiting for some event to happen, like the completion of an input/output operation (for example, reading data from a disk).

CamScanner

- **Exit/Terminate:** A process in this state has finished its execution or has been stopped by the user for some reason. At this point, it is released by the operating system and removed from memory.

Process Control Block (PCB)

A Process Control Block (PCB) is a data structure that is used by an Operating System to manage and regulate how processes are carried out. In operating systems, managing the process and scheduling them properly play the most significant role in the efficient usage of memory and other system resources. In the process control block, all the details regarding the process corresponding to it like its current status, its program counter, its memory use, its open files, and details about CPU scheduling are stored.

With the creation of a process, a PCB is created which controls how that process is being carried out. The PCB is created with the aim of helping the OS to manage the enormous amounts of tasks that are being carried out in the system. The **Process Table** is an array of PCBs, which logically contains a PCB for all of the current processes in the system.

Structure of the Process Control Block

A 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. The diagram helps explain some of these key data items.

CamScanner

Figure: Process control block (PCB)

Process state. The state of the process is stored in the PCB which helps to manage the processes and schedule them. There are different states for a process which are "running," "waiting," "ready," or "terminated."

Process ID/ Number: The OS assigns a unique identifier to every process as soon as it is created which is known as Process ID, this helps to distinguish between processes.

Program counter. While running processes when the context switch occurs the last instruction to be executed is stored in the program counter which helps in resuming the execution of the process from where it left off.

CPU registers. The registers vary in number and type, depending on the computer architecture. They include accumulators, index registers, stack pointers, and general-purpose registers, plus any condition-code information. Along with the program counter, this state information must be saved when an interrupt occurs, to allow the process to be continued correctly afterward when it is rescheduled to run.

CPU-scheduling information. The priority of the processes or the algorithm of scheduling is stored in the PCB to help in making scheduling decisions of the OS.

CamScanner

Memory-management information. This information may include such items as the value of the base and limit registers and the page tables, or the segment tables, depending on the memory system used by the operating system.

Accounting information. This information includes the amount of CPU and real time used, time limits, account numbers, job or process numbers, and so on.

I/O status information. This information includes the list of I/O devices allocated to the process, a list of open files, and so on.

List of Open files: This information includes the list of files opened for a process.

CamScanner

Process table:

The **process table** is a data structure used by the operating system to keep track of all processes. It is the collection of Program control Blocks (PCBs) which contains information about each process, such as its ID (PID), current state (e.g., running, ready, waiting), CPU usage, memory allocation, and open files.

- The process table helps the operating system manage processes efficiently by tracking their progress and resource usage.
- Each entry in the table corresponds to one process, and the operating system updates it as the process moves through different states in its lifecycle.

The states of a process represent the different stages a process goes through during its lifecycle in an operating system. A process can be in states such as **new**, **ready**, **running**, **waiting**, or **terminated**, depending on what it is doing and the resources it needs. These states help the operating system manage processes efficiently, ensuring that tasks are executed smoothly and system resources are used effectively.

Different types of processes in process table can be:

1. New Process
2. Ready Process
3. Running Process
4. Blocked / Waiting Process
5. Terminated Process
6. Zombie Process
7. Orphan Process

CS CamScanner

8. Daemon Process

1. New (Created) Process

A process that has been created but is not yet ready for execution. It remains in this state until the operating system assigns the necessary resources.

2. Ready Process

It is the process that is ready to be loaded into the main memory. This process is ready to run and is waiting to get the CPU time for its execution. Processes that are ready for execution by the CPU are maintained in a queue called a ready queue for ready processes.

3. Running Process

A process that is currently being executed by the CPU. Only one process (or more in multicore systems) can be in the running state at a time.

4. Blocked (Waiting) Process

A process that is waiting for an event to occur, such as input/output (I/O) completion or a resource becoming available. It cannot continue execution until the required condition is met.

5. Terminated Process

A process that has completed its execution or has been explicitly killed. It remains in the process table briefly until the operating system removes it.

6. Zombie Process

A zombie process is a process that has completed its execution but still remains in the process table because its parent process has not yet read its exit status. It is called a "zombie" because it is no longer active or running, but it still exists as a placeholder in the system.

CamScanner

Entry of child process remains in the process table until the parent process retrieves the exit status. During this time, the child process is referred to as a **zombie process**. This happens because the operating system keeps the process table entry to allow the parent to gather information about the terminated child.

7. Orphan Process

An orphan process is a child process currently performing its execution, whose parent has finished its execution and has terminated, leaving the process table without waiting for the child process to finish. Orphan processes are still active and continue to run normally, but they no longer have their original parent process to monitor or control them.

8. Daemon Process

A **daemon process** is a background process that runs independently of any user control and performs specific tasks for the system. Daemons are usually started when the system starts, and they run until the system stops. A daemon process typically performs system services and is available at all times to more than one task or user. Daemon processes are started by the root user or root shell and can be stopped only by the root user.

Process Management

Process Management for a single tasking or batch processing system is easy as only one process is active at a time. With multiple processes (multiprogramming or multitasking) being active, the process management becomes complex as a CPU needs to be efficiently utilized by multiple processes. Multiple active processes can may share resources like memory and may communicate with each other. This further makes things complex as an Operating System has to do process synchronization.

Please remember the main advantages of having multiprogramming are system responsiveness and better CPU utilization. We can run multiple processes in interleaved manner on a single CPU. For example, when the current process is getting busy with IO, we assign CPU to some other process.

CS CamScanner

CPU-Bound vs I/O-Bound Processes

A CPU-bound process requires more CPU time or spends more time in the running state. An I/O-bound process requires more I/O time and less CPU time. An I/O-bound process spends more time in the waiting state.

Process planning is an integral part of the process management operating system. It refers to the mechanism used by the operating system to determine which process to run next. The goal of process scheduling is to improve overall system performance by maximizing CPU utilization, minimizing throughput time, and improving system response time.

Process Management Tasks

Process management is a key part in operating systems with multi-programming or multitasking.

- **Process Creation and Termination:** Process creation involves creating a Process ID, setting up Process Control Block, etc. A process can be terminated either by the operating system or by the parent process. Process termination involves clearing all resources allocated to it.

- **CPU Scheduling:** In a multiprogramming system, multiple processes need to get the CPU. It is the job of Operating System to ensure smooth and efficient execution of multiple processes.

- **Deadlock Handling:** Making sure that system does not reach a state where two or processes cannot proceed due to a cycling dependency on each other.

- **Inter-Process Communication:** Operating System provides facilities such as shared memory and message passing for cooperating processes to communicate.

- **Process Synchronization:** Process Synchronization is the coordination of execution of multiple processes in a multiprogramming system to ensure that they access shared resources (like memory) in a controlled and predictable manner.

CamScanner

Process Scheduling

The objective of multiprogramming is to have some process running at all times so as to maximize CPU utilization. The objective of time sharing is to switch a CPU core among processes so frequently that users can interact with each program while it is running. To meet these objectives, the process scheduler selects an available process (possibly from a set of several available processes) for program execution on a core. Each CPU core can run one process at a time. For a system with a single CPU core, there will never be more than one process running at a time, whereas a multicore system can run multiple processes at one time. If there are more processes than cores, excess processes will have to wait until a core is free and can be rescheduled. The number of processes currently in memory is known as the degree of multiprogramming.

Balancing the objectives of multiprogramming and time sharing also requires taking the general behavior of a process into account. In general, most processes can be described as either I/O bound or CPU bound. An I/O -bound process is one that spends more of its time doing I/O than it spends doing computations. A CPU-bound process, in contrast, generates I/O requests infrequently, using more of its time doing computations.

A common representation of process scheduling is a queueing diagram, such as that in following Figure. Three types of queues are present: the job queue, ready queue and a set of wait queues. The circles represent the resources that serve the queues, and the arrows indicate the flow of processes in the system.

All processes are initially in the Job Queue. Then, a new process is initially put in the Ready queue by scheduler. It waits in the ready queue until it is selected for execution (or dispatched). Once the process is assigned to the CPU and is executing, one of the following several events can occur:

Fig. Queueing-diagram representation of process scheduling

CamScanner

Fig. Queueing-diagram representation of process scheduling

- The process could issue an I/O request and then be placed in an I/O wait queue.
- The process could create a new child process and then be placed in a wait queue while it awaits the child's termination.
- The process could be removed forcibly from the core, as a result of an interrupt or having its time slice expire, and be put back in the ready queue.

Types of Scheduling Queues

We will look at the numerous types of Scheduling Queues used in computer systems in the following sub-topics.

CamScanner

Job Queue (In Disk)

This queue is known as the job queue, it contains all the processes or jobs in the list that are waiting to be processed. Job: When a job is created, it goes into the job queue and waits until it is ready for processing.

- Contains all submitted jobs.
- Processes are stored here in a wait state until they are ready to go to the execution stage.
- This is the first and most basic state that acts as a default storage of new jobs added to a scheduling system.
- Long Term Scheduler picks a process from Job Queue and moves to ready queue.

Ready Queue (In Main Memory)

The Stand-by queue contains all the processes ready to be fetched from the memory, for execution. When the process is initiated, it joins the ready queue to wait for the CPU to be free. The operating system assigns a process to the executing processor from this queue based on the scheduling algorithm it implements.

- Contains processes (mainly their PCB s) waiting for the CPU to execute various processes it contains.
- They are controlled using a scheduling algorithm like FCFS, SJF, or Priority Scheduling.
- Short Term Scheduler picks a process from Ready Queue and moves the selected process to running state.

Block or Device Queues (In Main Memory)

The processes which are blocked due to unavailability of an I/O device are added to this queue. Every device has its own block queue.

CS CamScanner

Schedulers are a special type of operating system software that manages process scheduling in a variety of ways. Its main function is to select the jobs that are to be submitted to the system and decide which process will run.

Types of Process Schedulers

There are three types of process schedulers:

1. Long Term or Job Scheduler

Long Term Scheduler loads a process from disk to main memory for execution. The new process to the 'Ready State'.

- It mainly moves processes from Job Queue to Ready Queue.
- It controls the Degree of Multi-programming, i.e., the number of processes present in a ready state or in main memory at any point in time.
- It is important that the long-term scheduler make a careful selection of both I/O and CPU-bound processes. I/O-bound tasks are which use much of their time in input and output operations while CPU-bound processes are which spend their time on the CPU. The job scheduler increases efficiency by maintaining a balance between the two.
- In some systems, the long-term scheduler might not even exist. For example, in time-sharing systems like Microsoft Windows, there is usually no long-term scheduler. Instead, every new process is directly added to memory for the short-term scheduler to handle.
- Slowest among the three (that is why called long term).

2. Short-Term or CPU Scheduler

CPU Scheduler is responsible for selecting one process from the ready state for running (or assigning CPU to it).

- STS (Short Term Scheduler) must select a new process for the CPU frequently to avoid starvation.

CamScanner

- The CPU scheduler uses different scheduling algorithms to balance the allocation of CPU time.
- It picks a process from ready queue.
- Its main objective is to make the best use of CPU.
- It mainly calls dispatcher.
- Fastest among the three (that is why called Short Term).

The dispatcher is responsible for loading the process selected by the Short-term scheduler on the CPU (Ready to Running State). Context switching is done by the dispatcher only. A dispatcher does the following work:

- Saving context (process control block) of previously running process if not finished.
- Switching system mode to user mode.
- Jumping to the proper location in the newly loaded program.

Time taken by dispatcher is called dispatch latency or process context switch time.

CamScanner

3. Medium-Term Scheduler

Medium Term Scheduler (MTS) is responsible for moving a process from memory to disk (or swapping).

- It reduces the degree of multiprogramming (Number of processes present in main memory).
- A running process may become suspended if it makes an I/O request. A suspended processes cannot make any progress towards completion. In this condition, to remove the process from memory and make space for other processes, the suspended process is moved to the secondary storage. This process is called swapping, and the process is said to be swapped out or rolled out. Swapping may be necessary to improve the process mix (of CPU bound and IO bound)
- When needed, it brings process back into memory and pick up right where it left off.
- It is faster than long term and slower than short term.

CamScanner

What is a Dispatcher?

Dispatcher is a special type of program whose work starts after the scheduler. When the scheduler completes its task of selecting a process, it is the dispatcher that moves the process to the queue it needs to go to.

The dispatcher is the module that hands over control of the CPU to the process that has been selected by the short-term scheduler.

The following things happen in this function:

1. **switching context:** Cores the current process and restores the state of the process to be run next.
2. **Switching to User Mode:** Makes sure that it runs in the user mode and not kernel mode, which is for security and privilege break-
3. **Jumps to the correct location in the user program** from where the program can be restarted.

Advantages of a Dispatcher

- **Fast Process Switching:** Evaluates circumstances where a procedure shifts from the ready queue to the execution phase in a way that causes less delay.
- **Efficient CPU Time Allocation:** Is important in making sure that processes receive CPU time, hence giving the necessary support for multitasking to occur.

Disadvantages of a Dispatcher

- **Dispatch Latency:** Although the time taken is considerably small, this lateness in dispatching the requests can slow down the system.
- **Dependent on Scheduler:** The dispatcher hence cannot work on her own since she is reliant on the decisions made by the scheduler.

CS CamScanner

- maximize the CPU Utilization.
- Minimize response time & waiting time.

CPU Scheduling

What is CPU Scheduling:

- CPU Scheduling is how the operating system decides which process gets to use the CPU.
- It is the core idea behind multiprogramming - allowing the system to switch between processes and increase productivity.
- In modern systems, threads are scheduled, not entire program.
- "CPU" usually means a core when talking about running processes.

CPU Scheduling in Single-Core & Multi-Core Systems:

Main idea:

→ In a single-core CPU, only one process runs at a time.

→ Other processes must wait for their turn when the CPU is ~~busy~~ busy.

- Why Multiprograming ?
  - Goal: Always keep the CPU busy to maximize CPU usage.
  - Without multiprogramming:
    - If a process waits for I/O
    - The CPU sits idle.
  - With multiprogramming:
    - Multiple processes are kept ready in memory.
    - If one process waits, the CPU immediately switches to another process.

CamScanner

- In Multi-core Systems:
  - → The same idea is applied to all CPU cores.
  - → Each core can work on a different process, maximizing efficiency.

- Important Concept: CPU Burst & I/O Burst:
  - → A CPU burst: The time a process spends doing calculations (CPU work)
  - → An I/O burst: The time a process waits for i/o.

Processes keep switching between CPU bursts & I/O bursts.

⋮

load store
add store
read from file } CPU burst

[Wait for I/O] } I/O burst

Store increment
index
write to file } CPU burst

[Wait for I/O] } I/O burst

load store
add store
read from file } CPU burst

[Wait for I/O] } I/O burst

⋮

fig: Alternating sequence of CPU & I/O bursts.

CamScanner

CPU - I/O Burst Cycle:

- What Happens During Process Execution?
  - → Processes repeatedly switch between CPU execution & I/O waiting.

    CPU burst → I/O burst → CPU burst → I/O burst → ... → terminate.

- A process starts with a CPU burst (doing calculations).
  - → Then waits for I/O
  - → After I/O is done, it goes back to CPU work.
  - → Finally, after many bursts, it ends. (terminate).

- CPU Burst Duration:
  - → Measured across many systems.
  - → Most processes have many short CPU bursts (quick calculations)
  - → Few processes have long CPU bursts (heavy calculations)

    Many small bursts, few large bursts.

fig: Histogram of CPU-burst calculations.

CamScanner

Types of Programs:

→ I/O-bound programs:

→ Many short CPU bursts.

→ Example: Program that read/write data often (e.g. media players).

→ CPU-bound programs:

→ Few but long CPU bursts.

→ Example: Heavy calculation apps (e.g., video rendering).

CPU Scheduler

What is a CPU scheduler?

- When the CPU becomes free, the OS must pick one process from the ready queue to run next.

→ CPU Scheduler = Makes the decisions which process gets the CPU.

- Ready Queue:

→ The Ready Queue holds all processes that are ready to run.

→ It's not always a simple FIFO.

→ Can be organized as:

- FIFO queue
- Priority queue
- Tree
- Unordered linked list

→ Each entry in the queue is a Process Control Block (PCB)

– it holds process information.

CamScanner

imp

When does CPU Scheduling happen? (decisions)

CPU scheduling happens when something changes with a running process:

1. Process moves from running → waiting (e.g., waiting for I/O)
2. Process moves from running → ready (e.g., an interrupt occurs)
3. Process moves from waiting → ready (e.g., I/O finishes)
4. Process terminates (finishes execution)

- In case 1 and 4: No choice - Must pick another process.
- In case 2 and 3: OS can choose to preemt the current process or not.

imp

Types of Scheduling:

Nonpreemtive Scheduling (Cooperative):

- Once a process gets the CPU, it keeps it until it finishes or waits for I/O.
- No interruption by OS.
- Simple to design, but not good for real-time or fast responses.

* Rare today — mainly used in very old or simple systems.

P.T.O

CamScanner

Preemtive Scheduling:

→ Os can take the CPU away from a running process and give it to another.

→ Most modern OS (Windows, Linux, macOS) use preemtive scheduling

→ Allows better response time & supports multitasking.

Problem:

- Race conditions can happen:

→ If two processes share data and one is interrupted in the middle of updating it, the data can get corrupted.

Preemtive Kernal Vs Nonpreemtive Kernal:

Nonpreemtive Kernal (~~Modern OS~~)	Preemtive Kernal (Modern OS)
→ The kernal finishes system calls before switching processes.	→ Kernal can be interrupted.
→ Simple design, but bad for real-time tasks.	→ Needs extra care: uses tools like mutex locks to protect shared data.

Dispatcher & Dispatch Lateney:

- **Dispatcher:**

→ Part of the OS that gives CPU control to a process.

→ Switches between processes (save current process state, load next process state)

- **Dispatch Lateney:**

→ Small delay between stopping one process & starting another.

should be fast to reduce delay.

(Example: P0's state and restoring P1's state takes) a little time.

CS CamScanner

fig: The role of the dispatcher.

- **Handling Interrupts:**
  - → Interrupts can happen anytime (keyboard input, network packet, etc).
  - → The OS must be ready to accept interrupts to avoid losing data.
  - → When working with critical parts of code (like device drivers), interrupts are temporarilly disabled:
    → Disable interrupts → critical work → Enable interrupts.

Dispatcher's main Job:

→ Switch context (save old process info, load new process info.

→ Switch to use mode (from system mode to normal program running mode.

→ Jump to the process's correct starting point (where it left off or needs to start).

CamScanner

How often does a Context Switch Happen?

- Context Switch = Switching CPU from one process to another.
- We can measure it using vmstat in linux.

Context Switch Types:

- Voluntary Context Switch:
  - → Happens when the process willingly gives up CPU.
  - → Example: Waiting for I/O.
- Non Voluntary Context Switch:
  - → Happens when OS forcefully takes CPU away.
  - → Example: Time slice over or a higher priority process preempts it.

imp

What is CPU scheduling?

→ CPU scheduling is how the Operating system decides which process gets to use the CPU while others wait. Whenever the CPU becomes idle, the OS quickly picks a process from the ready queue.

Why do we need CPU Scheduling?

1. Keep the CPU Busy:
   - Make sure the CPU doesn't stay idle when there are processes waiting.
2. Improve Resource Utilization:
   - Use CPU, memory and devices efficiently without wasting time.

CamScanner

3. Handle Multiprogramming:

- In multiprogramming, many processes exit.
- If too many I/O-bound processes are picked by mistake, the CPU can stay idle (bad).
- Good scheduling selects the right mix of CPU-bound and I/O-bound processes to keep CPU active.

imp

Main Scheduling Criteria:

Different CPU scheduling algorithms (like FCFS, SJF, etc) have different strengths and weakness. To choose the best algorithm we need to compare them using specific criteria.

1. CPU Utilization:
   - Goal: Keep CPU as busy as possible.
   - Ideal Range:
     → 40% (lightly loaded system) to 90% (heavily loaded system)

2. Throughput:
   - Definition: Number of processes completed per time unit.
   - Example:
     → Long processes → maybe 1 process every few seconds.
     → Short processes → maybe tens of processes per second.

CamScanner

3. Turnaround Time:

Definition: Total time from submission to completion of a process.

Formula: $\boxed{\text{Turnaround time} = \text{Completion time} - \text{Arrival time}}$

4. Waiting Time:

Definition: Time a process spends waiting in the ready queue. Here, only waiting is counted, not execution or I/O.

$$\boxed{\text{Waiting Time} = \text{Turnaround time} - \text{Burst time}}$$

5. Response Time:

Definition: Time from submission of a request to the first response. It's about starting to respond, not finishing the output.

Example:

- We click a button → response is when the system reacts, not when the full task is done.

From here we want to optimize:

→ Maximize:

- CPU Utilization
- Throughput

→ Minimize:

- Turnaround Time
- Waiting Time
- Response Time.

CamScanner

Terminologies Used in CPU Scheduling:

1. **Arrival Time (AT):**
   Point of time
   → The time when a process enters the ready queue (ready to be scheduled).

2. **Completion Time (CT):**
   Point of time
   → The time when a process finishes its complete execution.

3. **Burst Time (BT):**
   Duration of time
   → The total time a process needs the CPU to execute fully (excluding waiting).

4. **Turnaround Time (TAT):**
   Duration of time
   → Total time taken from arrival to completion of the process.

   $TAT = CT - AT$

5. **Waiting Time (WT):**
   Duration of time
   → Time spent in the ready Queue.

   $WT = TAT - BT$

6. **Response Time (RT):**
   Duration of time
   RT = First time the process gets CPU for execution − Arrival time.

ESE-19 Final A 3(b)

Differentiate between preemptive & non-preemptive scheduling?

Feature	Preemtive Scheduling	Non-preemptive Scheduling
~~Definition:~~ Uses:	Used when a process switches from running state to ready or waiting to ready state	Used when a process terminates, or switches from running state to waiting state.
CPU Taken From process?	Yes, can take CPU away before completion.	No, process keeps CPU until it finishes or waits.
Example algorithms:	Round Robin, Preemptive Priority Scheduling	FCFS, Non-preemptive SJF
Response time	Generally better (faster first respone)	Can be slower if a long process is running.
Complexity	More complex to implement	Simple.

Are turnaround time & response time the same?

→ No they are different.

Turnaround Time → Total time from arrival to completion (whole process time)

Response Time → Time from arrival to first response. (when process starts, not finishes.)

Ans

CamScanner

CPU Scheduling

(Waiting Time = Response Time)

- **Preemptive**
  - (PS) Priority Scheduling
  - (SRTF) Shortest Remaining Job First
  - LRTF (Longest Remaining Job First
  - Round Robin
- **Non Preemptive**
  - FCFS
  - SJF
  - LJF
  - HRRN (Highest Respone Ratio Next.

Convoy effect. Longer process makes shorter processes wait longer.

no priority or (burst time) consideration

FCFS

→ First come, First Serve.

→ Processes are served in the order they arrive.

→ Non-preemptive. (Once a process starts running, it can't be interrupted.

How FCFS Works:

1. Arrival: Processes arrive & get added to a queue in order.
2. Execution:
   → CPU peaks the first process in the queue.
   → Executes it completely. (Until it finishes / waits for I/O)
3. Repeat: CPU peaks the next process & executes it.

This continues untill all processes are finished.

CamScanner

Example: Process with Same Arrival time:

Consider the following table of arrival time & burst time for three processes P_1, P_2 & P_3.

=>

Process	Arrival (AT) Time	Burst Time (BT)	Completion Time (CT)	Turn Around Time (TAT)	Waiting Time (WT)	RT
P_1	0	5	5	5	0	0
P_2	0	3	8	8	5	5
P_3	0	8	16	16	8	8

Ready Queue:

P_1	P_2	P_3

∴ Avgerage Turn around time = 9.67

∴ Average waiting time = 4.33

Ans

Gantt chart:

P_1	P_2	P_3

0 5 8 16

Example: Process with Different Arrival Times.

Process	Arrival Time (AT)	Burst Time (BT)	Completion Time (CT)	Turn Around Time (TAT)	WT	RT
P_1	2ms	5 ms	8ms	6ms	1ms	1ms
P_2	0ms	3ms	3ms	3ms	0ms	0ms
P_3	4ms	4ms	12ms	8ms	4ms	4ms

Ready Queue:

P_2	P_1	P_3

∴ Avg. TAT = 5.67

∴ Avg. WT = 1.67

Ans

Gantt chart:

P_2	P_1	P_3

0 3 8 12

CamScanner

Longer processes may keep getting postponed if short ones keep arriving } Starvation

Shortest Job First (SJF) Scheduling

→ shortest next CPU Burst algo.

- Best average waiting time.
- Difficult to predict exact burst time in advance

What is SJF?

→ SJF = Shortest Job First.

→ The process with the smallest burst time (CPU time needed) is selected first to execute.

→ Goal: Finish short processes quickly & reduce average waiting time.

Types of SJF:

1. Non-Preemptive SJF:
   - Once a process starts, it runs till completion (can't be interrupted)
2. Preemptive SJF (also called Shortest Remaining Time First (SRTF)):
   - If a new process arrives with shorter remaining time, the current running process is preempted. (interrupted).

Mid-18,19,20 Briefly explain how the SJF algorithm works?

1. Check all processes that are in ready queue.
2. Choose the one with the smallest burst time.
3. Execute it.
4. Repeat for the next shortest process.

CamScanner

Example: SJF

Process	AT	Burst Time	CT	TAT	WT	RT
P_1	1	3	6	5	2	2
P_2	2	4	10	8	4	4
P_3	1	2	3	2	0	0
P_4	4	4	14	10	6	6

Ready Queue:

P_1	~~P_3~~	~~P_2~~	~~P_4~~

$\therefore$ Avg. TAT $= \frac{25}{4}$

$\therefore$ Avg WT $= \frac{12}{4} = 3$

Ans

Gantt Chart:

idle	P_3	P_1	P_2	P_4

0 1 3 6 10 14

Example:

Process	AT	BT	CT	TAT	WT	RT
P_1	1	7	8	7	0	0
P_2	3	3	13	10	7	7
P_3	6	2	10	4	2	2
P_4	7	10	31	24	14	14
P_5	9	8	21	12	4	4

Ready Queue:

~~P_1~~	~~P_2~~	~~P_3~~	~~P_4~~	~~P_5~~

$\therefore$ Avg. TAT $= \frac{57}{5}$

$\therefore$ Avg. WT $= \frac{27}{5}$

Ans

Gantt Chart:

idle	P_1	P_3	P_2	P_5	P_4

0 1 8 10 13 21 31

CamScanner

Book Example:

Process	AT	BT	CT	TAT	WT	RT
P_1	0	6	9	9	3	3
P_2	0	8	24	24	16	16
P_3	0	7	16	16	9	9
P_4	0	3	3	3	0	0

Ready Queue:

~~P_1~~	~~P_2~~	~~P_3~~	~~P_4~~

$\therefore$ Avg. TAT $= \frac{52}{4}$

$\therefore$ Avg WT $= \frac{28}{4}$

Gantt Chart:

P_4	P_1	P_3	P_2

0 3 9 16 24

Ans

Shortest Remaining Job/ Time First:

Book:

Process	AT	BT	CT	TAT	WT	RT
P_1	0	~~8~~ 7 0	17	17	9	0
P_2	1	~~4~~ 0	5	4	0	0
P_3	2	9 0	26	24	15	15
P_4	3	~~5~~ 0	10	7	2	2

Ready Queue:

~~P_1~~	~~P_2~~	~~P_3~~	~~P_4~~

Gantt Chart:

P_1	P_2	P_4	P_1	P_3

0 1 5 10 17 26

$\therefore$ Avg. TAT $= \frac{52}{4}$ Ans

$\therefore$ Avg. WT $= \frac{26}{4}$ Ans

CamScanner

Final 19 CA

3(e) Schedule the following processes according to the shortest job first scheduling algorithm. Time quantum: 4.

⇒

Process	AT	BT	CT	TAT	WT	RT
P_1	0	7	12	12	5	5
P_2	0	5	5	5	0	0
P_3	11	6	23	12	6	17
P_4	12	2	14	2	0	12
P_5	12	3	17	5	2	14

Time required for completion
मतलब
Burst time.

Ready Queue:

~~P_1~~	~~P_2~~	~~P_2~~	P_1	P_3	~~P_4~~	~~P_5~~

Gantt chart:

P_2	P_1	P_2	P_1	P_4	P_5	P_3

0 4 8 9 12 14 17 23

∴ Avg. TAT $= \frac{36}{5}$

∴ Avg WT $= \frac{13}{5} = 2.6$

Ready Queue:

~~P_1~~	~~P_2~~	~~P_2~~	~~P_1~~	~~P_3~~	~~P_3~~	~~P_4~~	~~P_5~~	~~P_3~~

Gantt chart:

P_2	P_2	P_1	P_1	P_1	P_4	P_5	P_3	P_3

0 4 5 9 12 14 17 21 23

Ans

CamScanner

Final A (3)c 20

Schedule the following process according to the shortest job first algorithm in a preemptive environment. Time quantum: 3

⇒

Process	AT	BT	CT	TAT	WT	RT
P_1	1	7	18	17	10	10
P_2	1	3	4	3	0	0
P_3	1	5	9	8	3	3
P_4	8	2	11	3	1	1

Ready Queue:

~~P_1~~	~~P_2~~	~~P_3~~	~~P_3~~	P_4	~~P_3~~	~~P_1~~	~~P_1~~	~~P_1~~

Gantt chart:

idle	P_2	P_3	P_3	P_3	P_4	P_1	P_1	P_1

0 1 4 7 8 9 11 14 17 18

∴ Avg. TAT = $\frac{31}{4}$

∴ Avg. WT = 3.5

Ans

Final 18(A)

3(e) Schedule the following processes according to the shortest remaining time next algorithm. Time quantum: 3

=>

Process	AT	BT	CT	TAT	WT	RT
P_1	5	~~8~~ ~~5~~ ~~2~~ 0	22	17	9	9
P_2	5	10	32	27	17	17
P_3	5	~~7~~ ~~4~~ ~~1~~ 0	14	9	2	0
P_4	8	~~2~~ 0	10	2	0	0

Ready Queue:

~~P_1~~	~~P_2~~	~~P_3~~	~~P_4~~	~~P_3~~	~~P_3~~	~~P_1~~	~~P_1~~	~~P_2~~

Grantt chart:

idle	P_3	P_4	P_3	P_3	P_1	P_1	P_1	P_2

0 5 8 10 13 14 17 20 22 32

Ans

$\therefore$ Avg. TAT $= \frac{55}{4}$ Avg WT $= \frac{28}{4}$ Ans

Every process gets an equal chance to use the CPU: imp

fair & efficient

Round Robin (RR)

→ Round Robin is a preemptive CPU scheduling algorithm.

→ Each process gets a fixed time slice called a time quantum.

→ After a process uses up its time quantum, it is preempted (paused) & placed at the end of the ready queue.

→ The CPU is then given to the next process in the queue.

How Algorithm works:

(i) Read Queue
(ii) Dispatch (FCFS)
(iii) Time Quantum
(iv) Reduce BT
(v) Any process arrived at extended time?
(vi) Context Switching.

Best for multitasking & realtime systems.

CamScanner

Example Book:

Process	BT	CT	TAT	WT
P_1	~~24~~ 20	30	30	6
P_2	~~3~~ 0	7	7	4
P_3	~~3~~ 0	10	10	7

Time quantum = 4ms

Ready Queue:

~~P_1~~	~~P_2~~	~~P_3~~	~~P_1~~	~~P_1~~	~~P_1~~	~~P_1~~	~~P_1~~

Ganttchart:

P_1	P_2	P_3	P_1	P_1	P_1	P_1	P_1	

0 4 7 10 14 18 22 26 30

Ans

$\therefore$ Avg. WT $= \frac{17}{3}$ Ans

$\therefore$ Avg. TAT $= \frac{47}{3}$ Ans

CamScanner

Thread

Lightweight sub-task inside a process.

What is a Thread?

→ A thread is the smallest unit of CPU execution.

It contains:

- → Thread ID
- → Program Counter (PC)
- → Registers
- → Stack

Shares code, data and resources (like files) with other threads in the same process.

Single Vs Multithreaded Process

Traditional (Single-Threaded)

- → Has only one thread
- → Can only do one task at a time

Single-threaded process

Multi-threaded

- → Has multiple threads in one process.
- → Can perform multiple tasks at once (parallelism)

Multi-threaded process.

CamScanner

Why use Threads?

Example: Web Server

- → A web server serves pages, images, videos etc.
- → If it's single-threaded, it can handle only one client at a time.
- → Clients would wait a long time during heavy traffic.

Better Solution: Multithreading

- The server creates a new thread for each client request (instead of creating a new process).
- Faster, less memory usage, more efficient!

Thread Vs Process: (imp)

Feature	Process	Thread
• Creation cost	High	Low
• Memory use	More (separate)	Less (shared)
• Efficiency	Lower	Higher
• Resource sharing	Not shared easily	Shared with same process
• Speed	Low	High
• Part	It's full part.	Parts of a process

fig: Multithreaded server architecture.

CamScanner

• Weight	heavy weight	light weight
• Dependence	Independent	Dependent
• Context Switching	Slower	Faster
• System calls	Involved	Not involved
• Definition	A program in execution	A ligheweight unit of execution.
• Code & Data	Each has its own copy	Share the same code/data
• Parallel Tasks	One task at a time.	Multiple threads can run in parallel.
• Efficiency	More overhead	Less overhead.

Threads in OS Kernels:

- Modern OS kernels (like Linux) are multithreaded.
- During system boot, Linux creates multiple kernel threads to manage:
  - → Devices
  - → Memory
  - → Interrupts
  - → Other background tasks.

Threads in Applications:

- Many apps & algorithms benefit from multithreading:
  - → Sorting, Graphs, Trees
  - → Data mining, AI, Graphics
- On multicore CPUs, threads can run in true parallel, improving speed.

CamScanner

imp

Benefits of Multithreaded Programming:

Multithreading offers 4 major benefits:

1. **Responsiveness:**
   - → A multithreaded program can stay active & responsive even if part of it is busy.
   - → Useful for user interfaces:
     - Example: If a button triggers a long task, doing it in a separate thread keeps the UI smooth.
   - → Improves user experience by avoiding freezes.

2. **Resource Sharing:**
   - → Threads in the same process share memory, files & resources by default.
   - → Easier than processes, which need message passing or shared memory.
   - → This makes thread-based applications more efficient & simpler.

3. **Economy (Efficiency):**
   - → Creating threads is cheaper than creating full processes:
     - Less memory
     - Faster context switching.
   - → Threads save time, money & CPU overhead compared to processes.

4. **Scalability:**
   - → On multi-core CPUs, threads can run in parallel, increasing performance.
   - → A sigle-threaded app runs on only one core but multithreaded apps can ~~run~~ use all available cores.

CamScanner

Types of Threads in OS:

Threads are categorized into two types:

1. User-Level Threads (ULT)
2. Kernel-Level Threads (KLT)

fig: User & Kernel threads.

1. User-Level Threads (ULT):

→ Managed by user or application. (not by the OS)

→ The OS is unaware of these threads.

→ Implemented without system calls.

Advantages

- Easier to implement.
- Very fast context switching (no kernel involvement)
- Efficient & lightweight.
- Requires only program counter, registers & stack → simple structure

Disadvantages

- OS can't optimize or schedule them across CPUs.
- If one thread blocks, the whole process blocks.
- Thread management must be handled by the app → can get complex.

2. Kernel-Level Threads (KLT):

→ Managed by the OS kernel.

→ OS keeps a thread table to track & schedule them.

→ Kernel knows about each thread & can schedule them independently.

CamScanner

Advantages

- Can run on multiple processor/cores (good for multicore systems)
- OS can optimize and load balance threads across CPUs.
- Better handling of blocking operations.
- Ideal for high-performance and real-time applications.

Disadvantages

- Slower context-switching (switches between user & kernel mode)
- Needs system calls frequency → more CPU overheads.
- Large number of threads may overload the kernel.
- More complex to implement than ULTs.

Feature	User-Level Tread (ULT)	Kernel-Level Thread (KLT)
Managed By	Application/User	OS kernel
OS Awareness	No	Yes
Context Switch	Faster	Slower
Blocking Systemcall	Blocks entire process	Only the calling thread is blocked.
Multicore Usage	Not well utilized	Good utilization across cores.
Efficiency	Lightweight & efficient	More overhead
Complexity	Simple	Complex.

CamScanner

Multithreading Models:

When using threads, we need to define how user-level threads maps to kernel-level threads. There are 3 models:

1. Many-to-One Model:

- Many user threads → One kernel thread.
- Thread management is done in user space (by a thread library)
- Efficient, but has major limits.

Advantages:

→ Fast context switching (done in user space)

→ Lightweight (fewer kernel interactions)

Disadvantages:

→ If one thread blocks, the entire process blocks.

→ No parallelism on multicore CPUs (only one thread can access the ~~thread~~ kernel.

→ Not suitable for ~~multi~~ modern multi-core systems.

Example: Used in Green Threads (early) Java, Solaris

fig: Many-to-one model

CamScanner

2. One-to-One Model:

- Each user thread → One kernel thread
- More powerful, supports real parallelism on multicore systems.

Advantages

→ Multiple threads can run in parallel.

→ If one thread blocks, others can continue.

Disadvantages

→ More overhead: every user thread requires a kernel thread.

→ Too many threads = resource limits or system shutdown.

Example:
Used in Linux, Windows.

fig: One-to-one model

3. Many-to-Many Model:

- Many user threads → Some (or equal number of) kernel threads
- Offers a balance: allows many user threads, with efficient kernel mapping.

Advantages:

- Supports parallelism
- Efficient resource use - user threads don't always need kernel threads.
- If one thread blocks, others can still run.

Disadvantages:

- Complex to implement.
- Requires a good scheduling strategy.

CamScanner

Comparison Table:

Feature	Many-to-one	One-to-one	Many-to-Many
Mapping	Many user → 1 Kernel	1 user → 1 Kernel	Many user → Many / Some Kernel.
Parallelism	No	Yes	Yes
Blocking impact	Entire process blocks	Others can continue	Others can continue
Efficiency	Very efficient	Moderate	Balanced
Example Use	Green Threads (Java)	Linux, Windows	Solaris, modern thread pools.

Thread Libraries:

A Thread Library gives programmers a way to create & manage threads through a set of functions (API).

- Two Implemention Methods:

1. User-Level Library:
   - Entirely in user space.
   - No kernel involvement.
   - Fast calls, but OS doesn't know about the threads.

fig: two-level-model

2. Kernel-Level Library:
   - Part of the OS kernel.
   - API calls go through system calls.
   - More control, better performance on multicore.

CamScanner

Important

Process Lifecycle:

When we run a program, it becomes a process and goes through different states before it finishes.

There are five states:

Process Synchronization Deadlock in Operating System

- **What is Process Synchronization?**

→ Process Synchronization of multiple processes or threads to ensure they can run together (concurrently) without interfering with each other.

Main Objective:

To ensure:

- Safe access to shared resources
- Consistent and reliable data
- Avoidance of race conditions and deadlocks.

- **Why is it important?**

→ In a multi-process system, many processes share memory, files or devices.

→ Synchronization prevents data inconsistency and ensures system stability.

CamScanner

Synchronization Techniques:

- Semaphores
- Monitors
- Critical Sections

• Types of Processes Based on Synchronization:

Serially ($P_1 | P_2 | P_3$)

- **Independent Process:** The execution of one process doesn't affect the execution of other processes.

Parallely ($P_1 | P_2 | P_3$)

- **Cooperative Process:** A process that can affect or be affected by other processes executing in the system.

• Problem Case:

Synchronization problems arise mostly in Cooperative processes because they share resources.

• Problems Due to Lack of Synchronization:

1. **Inconsistency:**
   - When two or more processes access shared data at the same time, one process's changes can be overwritten by another.
   - Result: Unreliable & incorrect data.

CamScanner

2. Loss of Data:

- When multiple processes write to the same resource without coordination.
- Result: Important data may be lost or corrupted.

3. Deadlock:

- Two or more processes wait forever for each other to release a resource
- Result: System freezes, no task completes

- Conditions That Require Process Synchronization:

1. Critical Section:

- The part of a program where shared resources are accessed.
- Only one process should enter at a time to prevent conflict.

2. Race Condition:

- When the result of a program depends on the order in which processes execute.
- Needs synchronization to ensure correct order of execution.

3. Preemption:

- When OS stops one process to give CPU to another.
- If the first process didn't finish updating a shared resource, the next process may read wrong or partial data without synchronization.

CS CamScanner

2. Loss of Data:

- When multiple processes write to the same resource without coordination.
- Result: Important data may be lost or corrupted.

3. Deadlock:

- Two or more processes wait forever for each other to release a resource
- Result: System freezes, no task completes

- Conditions That Require Process Synchronization:

1. Critical Section:
   - The part of a program where shared resources are accessed.
   - Only one process should enter at a time to prevent conflict.

2. Race Condition:
   - When the result of a program depends on the order in which processes execute.
   - Needs synchronization to ensure correct order of execution.

3. Preemption:
   - When OS stops one process to give CPU to another.
   - If the first process didn't finish updating a shared resource, the next process may read wrong or partial data without synchronization.

CS CamScanner

Race - Condition:

- What is a Race Condition?

→ A Race Condition occures when multiple processes/threads access shared data at the same time, and the final result depends on the order of execution.

- Why is Race Condition a problem?

→ Because it can lead to incorrect or inconsistent results if processes interleave improperly while accessing shared data.

How can

Example of Race Condition?

- Shared variable, balance = 100
- Two operations: deposit(10) and withdraw(10)
- If withdraw runs first, updates balance to 90, and then ~~incorrectly~~ ~~b~~ deposit runs, update to 110.
- Final result should be 100, but incorrectly becomes 110 → race condition.

- How can Race Conditions be prevented?

→ By using:

- Critical sections (only one process allowed inside at a time)
- Locks
- Atomic operations.

12 May 2025, 07:41

CamScanner

Critical Section:

Definition: A critical section is a code segment that can be accessed by only one process at a time. It contains shared variables that need to be synchronized to maintain the consistency of data variables.

Critical Section Problem:

The Critical Section Problem is about designing a method to allow multiple cooperative processes to access shared resources (like variables or files) without causing data inconsistency.

- **why Critical Section Problem important?**

Because in a multiprocess system, uncontrolled access to shared resources may lead to:

- Data inconsistency
- Race conditions
- System crashes.

- **General structure of a typical process:**

```
do {
    [entry section]
        critical section
    [exit section]
        remainder section
} while (TRUE);
```

- What happens in the Entry Section of a process?

→ In the entry section, the process requests permission to enter the critical section. It waits if another process is already insid

imp

- What are the three requirements of a good solution to the Critical Section Problem?

→ Any solution to the critical section problem must satisfy three requirements:

- **Mutual Exclusion:**
  - → Only one process can enter the critical section at a time.
  - → This prevents two processes from changing shared data at the same time.

- **Progress:**
  - → If no process is inside the critical section, and some processes are waiting, then one of the waiting processes should be allowed to enter soon.
  - → The system should not make them wait forever when it's free.

- **Bounded Waiting:**
  - → When a process asks to enter there should be a limit on how many other processes can go ahead before it gets its turn.
  - → This avoids starvation, so every process gets a fair chance.

Semaphore:

is an integer variable that is used to solve the critical problem by using two atomic operations wait & signal that are used process synchronization.

P.T.O

CamScanner

- What problems can occur due to the use of Critical Sections?

→ The use of critical sections in a program can cause a number of issues, including:

I. Deadlock:

- Happens when two or more processes wait for each other forever to release the critical section.
- As a result, none of them can proceed.

II. Starvation:

- When a process is continuously blocked from entering the critical section while others keep entering.
- It never gets its chance to run.

III. Overhead: Time and resources are used to acquire and release locks or semaphores, which may slow down the program.

- What are the advantages of using Critical Sections in process synchronization?

I. Prevents Race Condition:

▷ Prevents / Ensures only one process changes shared data at a time, so results stay correct.

II. Provide Mutual Exclusion:

▷ Stops multiple processes from using the same resource at the same time.

III. Reduce CPU usage:

▷ Processes can wait peacefully without wasting CPU time

IV. Simplifies Synchronization:

▷ Easy to manage - only one process uses a resource, no need for complex code.

CamScanner

- What are the disadvantages of using Critical Sections?

I. Overhead: Implementing critical section using synchronization mechanisms like semaphores and mutexes adds extra processing & slowing down program execution.

II. Deadlocks: If not designed properly, critical sections can lead to deadlocks.

III. Limits Parallelism: If used too much, only one process runs at a time, reducing speed in parallel systems.

IV. Contention: If many processes try to enter the same critical section often it causes slowdowns due to competition.

Peterson's Algorithm in Process Synchronization

```
int turn;
boolean flag[2]
```

$\boxed{j = 1 - i}$

Algorithm for process Pi

```
While (true)
{
    flag[i] = TRUE;
    turn = j;
    while (flag[j] && turn == j);
    CRITICAL SECTION
    flag[i] = FALSE;
    REMAINDER SECTION
}
```

Algorithm for process Pj

```
While (true)
{
    flag[j] = TRUE;
    turn = i;
    while (flag[i] &&
           turn == i);
    CRITICAL SECTION
    flag[j] = FALSE;
    REMAINDER SECTION
}
```

P.T.O.

CamScanner

Deadlock in Operating System

- **What is a Deadlock?**

→ Deadlock is a situation where two or more processes are blocked because each is holding a resource and waiting for another, which is held by another process.

No process can continue – they're stuck waiting on each other.

- **Give a real-life example of Deadlock.**

→ Two trains are on one track coming from opposite directions. Neither can move forward unless the other moves back – but both are waiting → deadlock.

- **How does a process use resources in an operating system?**

→ A process in an operating system uses resources in the following way:

- Requests a resource
- Use the resource
- Release the resource

- If two or more processes:

→ Hold one resource &

→ Wait for another resource held by another process,

Then none of them can proceed leading to deadlock.

CamScanner

• What are the necessary conditions for deadlock to occur in an Operating System?

→ Deadlock can happen only if all four of the following conditions are true at the same time:

I. Mutual Exclusion:

- A resource can be used by only one process at a time (non-sharable resource).

II. Hold and Wait:

- A process is holding one resource and waiting for another resource help by another process.

III. No Preemption:

- Resources can't be forcibly taken from a process. They must be released voluntarily.

IV. Circular Wait:

- A set of processes are waiting for each other in a circular chain.

- Example:

  P0 → waiting for P1,

  P1 → waiting for P2,

  ..., Pn → waiting for P0.

If any one these four conditions is not true, deadlock can't occur.

CS CamScanner

- What are the methods of handling deadlocks in Operating Systems?

→ There are 3 main methods to handle deadlocks:

I. Deadlock Prevention or Avoidance.

II. Deadlock Detection and Recovery.

III. Deadlock Ignorance.

- Deadlock Prevention or Avoidance:

Definition: Deadlock Prevention avoids deadlock by ensuring that at least one of the four necessary conditions never occurs.

I. Mutual Exclusion:

- Allow multiple processes to access shared (read-only) resources at the same time.
- But not possible for non-sharable resources (like printer).

II. Hold and wait:

- Ensure a process doesn't hold any resource when it requests a new one.
- Either allocate all resources at once before starting or force process to release and re-request.
  - Problem: Causes starvation – Some processes may wait too long for all resources.

III. No Preemption:

- If a process holds some resource and requests another, it must release what it already holds if the new resources isn't available.
- Later, the process re-requests all resources.
  - Problem: May lead to live lock (processes continuously change state without making progress).

CamScanner

IV. Circular Wait:

- Use a fixed resource order.
- A process can only request resources in a specific order preventing circular chain.

- Only the Circular Wait prevention is practical; other methods can harm performance.

Deadlock Avoidance:

Uses RAG & Banker's algo

Deadlock Avoidance is futuristic – the system checks in advance wheather giving a resource will lead to a deadlock.

- Require prior knowledge of how many resources a process will need.
- Uses the Banker's Algorithm to decide if granting a request is safe or unsafe.

- Compare Prevention vs Avoidance:

Aspect	Prevention	Avoidance
• How it works	Blocks a condition	checks system state before allocation.
• Data requirement	No future knowledge needed.	Needs future resource demand.
• Performance	May reduce performance.	Also impacts performance.
• Practicality	Circular wait prevention is commonly used.	Used with Banker's Algorithm.

CamScanner

Deadlock Detection and Recovery:

If deadlock prevention or avoidance is not used, the system uses deadlock detection and recovery, which includes two steps:

1. Detection: Checks if a deadlock exists.
2. Recovery: If found, take action to resolve it.

Data remains correct, but system performance decreases.

Deadlock Detection:

Deadlock Detection is the process where the system checks if any processes are stuck waiting for each other.

Detection algorithms:

- Resource Allocation Graph.
- Banker's Algorithm.

Deadlock Recovery:

Deadlock Recovery is the process of removing deadlock after it is detected.

There are 4 main techniques:

1. Mutual Intervention: The system informs the user or admin to manually fix the deadlock.

   Not suitable for large systems.

2. Automatic Recovery: The system fixes the deadlock on its own by:

   → Aborting processes or

   → Preempting resources.

12 May 2025, 07:41

CamScanner

3. Process Termination:

- Abort all deadlocked processes:
  - → Solves deadlock fast.
  - → Loss of work done so far.
- Abort one process at a time:
  - → Reduces data loss.
  - → Requires running detection again after each abort.

→ Factors to choose which process to abort:

- Process priority
- Completion progress
- Resources used
- Remaining resource need
- Total number of processes to kill.
- Type of process (interactive or batch)

4. Resource Preemption:

- Selecting a Victim:
  Choose a process holding resources - try to minimize recovery cost.
- Rollback:
  If resources are taken from a process, rollback to a previous safe state or restart the process.
- Starvation Prevention:
  Avoids picking the same process repeatedly as a victim.
  Limits how often a process can be preempted or rolled back.

CS CamScanner

- What Is Deadlock Ignorance ?

→ In this method, the system does nothing and ignores deadlock. If it occurs, the system might crash or reboot.

- Known as the Ostrich Algorithm – "bury your head in the sand."

Used in UNIX and Windows when deadlocks are rare.

Resource Allocation Graph (RAG)

Definition: A Resource Allocation Graph is a visual tool used to show how resources are assigned and requested by processes in an operating system. It helps in detecting deadlocks by showing the relationships between processes and resources.

- Why is RAG used?

→ To visualize potential deadlocks clearly

→ To understand how resources are allocated and requested.

→ To track which resources are available, in use, or being waited on.

- How is RAG different from Banker's Algorithm?

RAG uses nodes and arrows (graph-based), while Banker's Algorithm uses tables. RAG is mainly used for deadlock detection; Banker's is for deadlock avoidance.

What are the types of vertices (nodes) in RAG?

1. Process Vertex:

→ Represented by a circle.

→ Each process is shown as one circle.

CS CamScanner

2. **Resource Vertex**: Represented by a square or rectangle.

- **Single Instance Resource:**
  - → Only one copy of the resource is available.
  - → Can be assigned to only one process at a time.

- **Multi-Instance Resource:**
  - → Multiple copies available.
  - → Can be shared by multiple processes (like printers, files).

What are the types of edges in RAG?

There are two types of edges in RAG:

- **Request Edge:**
  - → Drawn from a process to a resource
  - → Means the process is requesting the resource.

- **Assignment Edge:**
  - → Drawn from a resource to a process.
  - → Means the resource is currently assigned to that process.

CamScanner

Note: Single Instance → cycle (Must be) deadlock

How do you identify a deadlock in RAG?

→ If the RAG shows a circular chain of processes and resources (i.e., a cycle), and all resources involved are single-instance, then deadlock exists.

Question 01:

→

Process No	Allocated		Requested	
	R_1	R_2	R_1	R_2
P_1	1	0	0	1
P_2	0	1	1	0

∴ Availability → (0,0) R_1, R_2

∴ Deadlock. Ans

Question 02:

→

Process No	Allocated		Request	
	R_1	R_2	R_1	R_2
P_1	1	0	0	0
P_2	0	1	0	0
P_3	0	0	1	1

∴ Availability → R_1, R_2 (0,0)

Acyclic ∴ No deadlock.

$P_1 \xrightarrow{req} (0,0)$ ∴ can be terminated. ∴ Availability = (1,0)

$P_2 \xrightarrow{req} (0,0)$ ∴ can be terminated ∴ Availability (1,1)

∴ No Deadlock.

CamScanner

Starvation vs Deadlock

Starvation	Deadlock
Waiting time finite	Waiting time infinite.

Question 03:

Process No	Allocation		Request	
	R_1	R_2	R_1	R_2
P_1	1	0	0	1
P_2	0	1	1	0
P_3	0	0	0	0

→ Available = 0 0 (R_1 R_2)

P_3 = 0 0

New Available = 0 1

P_1 = 1 0

New Available = 1 1

P_2 = 0 1

∴ New Available = 1 2 ∴ No Deadlock

Ans

- **What is an Allocation Matrix?**

→ An Allocation Matrix shows which resources are currently assigned to which processes.

To fill it:

→ Look at each resource and note which process it is assigned to.

→ Mark 1 if assign, 0 if not.

- **What is a Request Matrix?**

→ A request matrix shows which resources are currently being requested by each process.

To fill it: → look at each process and see which resources it is requesting (wanting)

→ Mark 1 if requested, 0 otherwise.

CamScanner

What is the Allocation Resource Vector?

→ The Available Resource Vector tells how many instances of each resource are currently free.

NB: Every cycle in a multi-instance resource type graph is not a deadlock.

Example:

→

Process No	Allocate			Request		
	R1	R2	R3	R1	R2	R3
P_0	1	0	1	0	1	1
P_1	1	1	0	1	0	0
P_2	0	1	0	0	0	1
P_3	0	1	0	1	2	0

$R_1, R_2, R_3,$

∴ Availability ⇒ (0, 0, 1)

P_2 → 0, 1, 0

New → (0, 1, 1)

P_0 → (1, 0, 1)

New → (1, 1, 2)

P_1 → (1, 1, 0)

New → (2, 2, 2)

P_3 → (0, 1, 0)

∴ New → (2, 3, 2)

Ans

∴ $P_2 \to P_0 \to P_1 \to P_3$

Ans

∴ No Deadlock

Ans

CamScanner

Banker's Algorithm
(Deadlock Avoidance)

Definition:
Banker's Algorithm is a deadlock avoidance algorithm used by OS to allocate resources safely. It ensures the system stays in a safe state by checking wheather a request can be fulfilled without leading to deadlock.

Why is it called Banker's Algorithm?
→ Like a banker checking if it can safely lend money to customers while still keeping enough to meet future needs, the OS only allocates resources if it ensures all processes can still finish later.

Components of Banker's Algorithm:

Data Structures used (for n processes, m resource types):

Structure	Purpose.
• Available [m]	→ Number of free instances of each resource.
• Max [n] [m]	→ Maximum demand of each process.
• Allocation [n] [m]	→ Number of resources currently given to each process.
• Need [n] [m]	→ Remaining need = Max - Allocation.

Formula:

$$\boxed{Need[i][j] = Max[i][j] - Allocation[i][j]}$$

CamScanner

What does Banker's Algorithm do?

It works in two parts:

1. **Safety Algorithm:** checks if the system is in a safe state. (i.e., all processes can complete in some order).

2. **Resource Request Algorithm:**
When a process requests resources, the system pretends to allocate and checks with the safety Algorithm:

✓ If safe → Grant the request.

✗ If unsafe → Deny the request (to avoid deadlock).

Question:

A = 10, B = 5, C = 7

Process No	Allocation			Max Need			Availability			Remaining need			
	A	B	C	A	B	C	A	B	C	A	B	C	
P_1	0	1	0	7	5	3	7	3	2	7	4	3	→ P_1
P_2	2	0	0	3	2	2	5	3	2	1	2	2	→ P_2
P_3	3	0	2	9	0	2	7	4	3	6	0	0	→ P_3
P_4	2	1	1	4	2	2	7	4	5	2	1	1	→ P_4
P_5	0	0	2	5	3	3	7	5	5	5	3	1	→ P_5
	7	2	5				10	6	7				

Safe Sequence:

$P_2 \rightarrow P_4 \rightarrow P_5 \rightarrow P_1 \rightarrow P_3$ Ans

No deadlock

safe.

$\boxed{\text{Remaining Need} = \text{Max need} - \text{Allocate}}$

$\boxed{\text{Current Availability} >= \text{Remaining Need}}$

Question: Check for deadlock, if deadlock is not present then find the safe sequence.

Process No	Allocation			Max Need			Availability			Remaining need		
	E	F	G	E	F	G	E	F	G	E	F	G
P_1	1	0	1	4	3	1	3	3	0	3	3	0
P_2	1	1	2	2	1	4	4	3	1	1	0	2
P_3	1	0	3	1	3	3	5	4	3	0	3	0
P_4	2	0	0	5	4	1	6	4	6	3	4	1
							8	8	7			

Safe sequence: $P_1 \rightarrow P_2 \rightarrow P_3 \rightarrow P_4$

$\therefore$ No deadlock Ans

Ans check:

E = 5, F = 1, G = 6

3, 3, 0

8, 4, 6

12 May 2025, 07:41

CamScanner

Question: A system is having 3 processes, each requires two units of resources R.

(i) The minimum num. of units of R requires such that no deadlock will occur?

(ii) The maximum num. of units of R requires such that deadlock will occur?

→

A	B	C
2	2	2

→ 1 + 1 + 1 = 3+1 = 4 → (i) Ans

(ii) 4-1 = 3 Ans

যা লাগবে তার থেকে এক কম নিয়ে নিয়ে সবযোগকরে আরও এক যোগ 1 পারেরটা একযোগ লাগবে না।

CamScanner

What is memory? Virtual memory, Physical memory.

Figure 9.1 A base and a limit register define a logical address space.

Figure 9.4 Memory management unit (MMU).

CamScanner

Figure 9.6 Hardware support for relocation and limit registers.

9.2 Contiguous Memory Allocation

The main memory must accommodate both the operating system and the various user processes. We therefore need to allocate main memory in the most efficient way possible. This section explains one early method, contiguous memory allocation. The memory is usually divided into two partitions: one for the operating system and one for the user processes.

In contiguous memory allocation, **each process is contained in a single section of memory** that is contiguous to the section containing the next process.

Variable partition scheme

In this scheme, the operating system keeps a table indicating which parts of memory are available and which are occupied. Initially, all memory is available for user processes and is considered one large block of available memory, a hole. Eventually, as you will see, memory contains a set of holes of various sizes. Figure 9.7 depicts this scheme. Initially, the memory is fully utilized, containing processes 5, 8, and 2. After process 8 leaves, there is one contiguous **hole**. Later on, process 9 arrives and is allocated memory. Then process 5 departs, resulting in two noncontiguous holes.

What happens when there isn't sufficient memory to satisfy the demands of an arriving process? One **option is to simply reject** the process and provide an appropriate error message. Alternatively, we can place su**ch processes into a wait queue**. When memory is later released,

CamScanner

the **operating system checks the wait queue to determine** if it will satisfy the memory demands of a waiting process.

In general, as mentioned, the memory blocks available comprise a set of holes of various sizes scattered throughout memory. When a process arrives and needs memory, the system searches the set for a hole that is large enough for this process. If the hole is too large, it is split into two parts. One part is allocated to the arriving process; the other is returned to the set of holes. When a process terminates, it releases its block of memory, which is then placed back in the set of holes. If the new hole is adjacent to other holes, these adjacent holes are merged to form one larger hole.

This procedure is a particular instance of the general dynamic storage allocation problem, which concerns how to satisfy a request of size n from a list of free holes. There are many solutions to this problem. The first-fit, best-fi, and worst-fi strategies are the ones most commonly used to select a free hole from the set of available holes.

Figure 9.7 Variable partition.

- **First fit**. Allocate the first hole that is big enough. Searching can start either at the beginning of the set of holes or at the location where the previous first-fit search ended. We can stop searching as soon as we find a free hole that is large enough.
- **Best fi** . Allocate the smallest hole that is big enough. We must search the entire list, unless the list is ordered by size. This strategy produces the smallest leftover hole.
- **Worst fit**. Allocate the largest hole. Again, we must search the entire list, unless it is sorted by size. This strategy produces the largest leftover hole, which may be more useful than the smaller leftover hole from a best-fit approach.

CamScanner

Example: Hole sizes are

30k 50k 65k 40k 15k 45k 70k 60K

A new process is to be loaded of size 38k.

First fit: ?

Best fit: ?

Worse fit: ?

9.2.3 Fragmentation

Both the first-fit and best-fit strategies for memory allocation suffer from **external fragmentation**. As processes are loaded and removed from memory, the free memory space is broken into little pieces. External fragmentation exists when there is **enough total memory space to satisfy a request** but the **available spaces are not contiguous**: storage is fragmented into a large number of small holes. This fragmentation problem can be severe. In the worst case, we could have a block of free (or wasted) memory between every two processes. If all these small pieces of memory were in one big free block instead, we might be able to run several more processes.

Memory fragmentation can be **internal as well as external**. Consider a multiple-partition allocation scheme with a hole of **18,464 bytes**. Suppose that the next process requests **18,462** bytes. If we allocate exactly the requested block, we are left with a hole **of 2 bytes**. The overhead to keep track of this hole will be substantially larger than the hole itself. The general approach to avoiding this problem is to break the physical memory into fixed-sized blocks and allocate memory in units based on block size. With this approach, the memory allocated to a process may be slightly larger than the requested memory. The difference between these two numbers is internal fragmentation —unused memory that is **internal to a partition**.

Solution:

- One solution to the problem of external fragmentation is compaction.
- Another possible solution to the external-fragmentation problem is to permit the logical address space of processes to be noncontiguous, thus allowing a process to be allocated physical memory wherever such memory is available. This is the strategy used in paging, the most common memory management technique for computer systems.

CamScanner

Paging:

Paging is a memory management scheme that permits a process's physical address space to be noncontiguous. Paging avoids external fragmentation and the associated need for compaction, two problems that plague contiguous memory allocation.

Every address generated by the CPU is divided into two parts: a **page number** (p) and a **page offset** (d):

Figure 9.8 Paging hardware.

The following outlines the steps taken by the MMU to translate a logical address generated by the CPU to a physical address:

1. Extract the page number p and use it as an index into the page table.
2. Extract the corresponding frame number f from the page table.
3. Replace the page number p in the logical address with the frame number f.

CamScanner

Figure 9.9 Paging model of logical and physical memory.

Figure 9.13 Valid (v) or invalid (i) bit in a page table.

CamScanner

Segmentation in Operating System

A process is divided into Segments. The chunks that a program is divided into which are not necessarily all of the exact sizes are called segments. Segmentation gives the user's view of the process which paging does not provide. Here the user's view is mapped to physical memory.

Swapping

Process instructions and the data they operate on must be in memory to be executed. However, a process, or a portion of a process, can be swapped temporarily out of memory to a backing store and then brought back into memory for continued execution (Figure 9.19). Swapping makes it possible for the total physical address space of all processes to exceed the real physical memory of the system, thus increasing the degree of multiprogramming in a system

CamScanner

Figure 9.19 Standard swapping of two processes using a disk as a backing store.

10.2 Demand Paging

CamScanner

Figure 10.4 Page table when some pages are not in main memory.

what happens if the process tries to access a page that was not brought into memory? Access to a page marked invalid causes a page fault. The paging hardware, in translating the address through the page table, will notice that the invalid bit is set, causing a trap to the operating system. This trap is the result of the operating system's failure to bring the desired page into memory. The procedure for handling this page fault is straightforward (Figure 10.5):

1. We check an internal table (usually kept with the process control block) for this process to determine whether the reference was a valid or an invalid memory access.
2. If the reference was invalid, we terminate the process. If it was valid but we have not yet brought in that page, we now page it in.
3. We find a free frame (by taking one from the free-frame list, for example).
4. We schedule a secondary storage operation to read the desired page into the newly allocated frame.
5. When the storage read is complete, we modify the internal table kept with the process and the page table to indicate that the page is now in memory.

CamScanner

6. We restart the instruction that was interrupted by the trap. The process can now access the page as though it had always been in memory.

Figure 10.5 Steps in handling a page fault.

Page Replacement Algorithms in Operating Systems

CamScanner

In an operating system that uses paging for memory management, a page replacement algorithm is needed to decide which page needs to be replaced when a new page comes in. Page replacement becomes necessary when a page fault occurs and no free page frames are in memory. in this article, we will discuss different types of page replacement algorithms.

Page Replacement Algorithms

Page replacement algorithms are techniques used in operating systems to manage memory efficiently when the physical memory is full. When a new page needs to be loaded into physical memory, and there is no free space, these algorithms determine which existing page to replace.

If no page frame is free, the virtual memory manager performs a page replacement operation to replace one of the pages existing in memory with the page whose reference caused the page fault. It is performed as follows: The virtual memory manager uses a page replacement algorithm to select one of the pages currently in memory for replacement, accesses the page table entry of the selected page to mark it as "not present" in memory, and initiates a page-out operation for it if the modified bit of its page table entry indicates that it is a dirty page.

Common Page Replacement Techniques

- **First In First Out (FIFO)**
- **Optimal Page replacement**
- **Least Recently Used (LRU)**
- **Most Recently Used (MRU)**

First In First Out (FIFO)

This is the simplest page replacement algorithm. In this algorithm, the operating system keeps track of all pages in the memory in a queue, the oldest page is in the front of the queue. When a page needs to be replaced page in the front of the queue is selected for removal.

Example 1: Consider page reference string 1, 3, 0, 3, 5, 6, 3 with 3-page frames. Find the number of page faults using FIFO Page Replacement Algorithm.

CamScanner

Page reference 1, 3, 0, 3, 5, 6, 3

1	3	0	3	5	6	3
		0	0	0	0	3
	3	3	3	3	6	6
1	1	1	1	5	5	5
Miss	Miss	Miss	Hit	Miss	Miss	Miss

Total Page Fault = 6

FIFO - Page Replacement

Initially, all slots are empty, so when 1, 3, 0 came they are allocated to the empty slots ---> 3 Page Faults.

when 3 comes, it is already in memory so ---> 0 Page Faults. Then 5 comes, it is not available in memory, so it replaces the oldest page slot i.e 1. ---> 1 Page Fault. 6 comes, it is also not available in memory, so it replaces the oldest page slot i.e 3 ---> 1 Page Fault. Finally, when 3 come it is not available, so it replaces 0 1-page fault.

Implementation of FIFO Page Replacement Algorithm

Optimal Page Replacement

In this algorithm, pages are replaced which would not be used for the longest duration of time in the future.

Example: Consider the page references 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 3 with 4-page frame. Find number of page fault using Optimal Page Replacement Algorithm.

CamScanner

Page reference 7,0,1,2,0,3,0,4,2,3,0,3,2,3 No. of Page frame - 4

7	0	1	2	0	3	0	4	2	3	0	3	2	3
			2	2	2	2	2	2	2	2	2	2	2
		1	1	1	1	1	4	4	4	4	4	4	4
	0	0	0	0	0	0	0	0	0	0	0	0	0
7	7	7	7	7	3	3	3	3	3	3	3	3	3
Miss	Miss	Miss	Miss	Hit	Miss	Hit	Miss	Hit	Hit	Hit	Hit	Hit	Hit

Total Page Fault = 6

Optimal Page Replacement

Initially, all slots are empty, so when 7 0 1 2 are allocated to the empty slots ---> 4 Page faults

0 is already there so ---> 0 Page fault. when 3 came it will take the place of 7 because it is not used for the longest duration of time in the future---> 1 Page fault. 0 is already there so ---> 0 Page fault. 4 will takes place of 1 ---> 1 Page Fault.

Now for the further page reference string ---> 0 Page fault because they are already available in the memory. Optimal page replacement is perfect, but not possible in practice as the operating system cannot know future requests. The use of Optimal Page replacement is to set up a benchmark so that other replacement algorithms can be analyzed against it.

Least Recently Used

In this algorithm, page will be replaced which is least recently used.

Example Consider the page reference string 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 3 with 4-page frames. Find number of page faults using LRU Page Replacement Algorithm.

CamScanner

Page reference 7,0,1,2,0,3,0,4,2,3,0,3,2,3 No. of Page frame - 4

7	0	1	2	0	3	0	4	2	3	0	3	2	3
			2	2	2	2	2	2	2	2	2	2	2
		1	1	1	1	1	4	4	4	4	4	4	4
	0	0	0	0	0	0	0	0	0	0	0	0	0
7	7	7	7	7	3	3	3	3	3	3	3	3	3
Miss	Miss	Miss	Miss	Hit	Miss	Hit	Miss	Hit	Hit	Hit	Hit	Hit	Hit

Total Page Fault = 6

Here LRU has same number of page fault as optimal but it may differ according to question.

Least Recently Used - Page Replacement

Initially, all slots are empty, so when 7 0 1 2 are allocated to the empty slots ---> 4 Page faults

0 is already there so ---> 0 Page fault. when 3 came it will take the place of 7 because it is least recently used ---> 1 Page fault

0 is already in memory so ---> 0 Page fault.

4 will takes place of 1 ---> 1 Page Fault

Now for the further page reference string ---> 0 Page fault because they are already available in the memory.

Most Recently Used (MRU)

In this algorithm, page will be replaced which has been used recently. Belady's anomaly can occur in this algorithm.

Example 4: Consider the page reference string 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 3 with 4-page frames. Find number of page faults using MRU Page Replacement Algorithm.

CamScanner

Page reference 7,0,1,2,0,3,0,4,2,3,0,3,2,3 No. of Page frame - 4

7	0	1	2	0	3	0	4	2	3	0	3	2	3
			2	2	2	2	2	2	3	0	3	2	3
		1	1	1	1	1	1	1	1	1	1	1	1
	0	0	0	0	3	0	4	4	4	4	4	4	4
7	7	7	7	7	7	7	7	7	7	7	7	7	7
Miss	Miss	Miss	Miss	Hit	Miss	Miss	Miss	Hit	Miss	Miss	Miss	Miss	Miss

Total Page Fault = 12

Most Recently Used - Page Replacement

Initially, all slots are empty, so when 7 0 1 2 are allocated to the empty slots ---> 4 Page faults

0 is already their so--> 0 page fault

when 3 comes it will take place of 0 because it is most recently used ---> 1 Page fault

when 0 comes it will take place of 3 ---> 1 Page fault

when 4 comes it will take place of 0 ---> 1 Page fault

2 is already in memory so ---> 0 Page fault

when 3 comes it will take place of 2 ---> 1 Page fault

when 0 comes it will take place of 3 ---> 1 Page fault

when 3 comes it will take place of 0 ---> 1 Page fault

when 2 comes it will take place of 3 ---> 1 Page fault

when 3 comes it will take place of 2 ---> 1 Page fault

CamScanner