# DI03016061: Operating Systems

Slide Creator

# Course Overview I

# Course Overview II

# Course Overview III

# Course Overview IV

# Definition and Functions of an Operating System

Course: Operating Systems

## Unit 1

Introduction to Operating System

## Lecture 1

Definition and Functions of an Operating System

# Agenda

What is an Operating System?
Computer System Structure
Goals of an OS
OS as a Resource Manager
OS as a Control Program
Primary Functions of OS
OS Viewpoints

# What is an Operating System?

A program that acts as an intermediary between a user of a computer and the computer hardware.
Provides a user-friendly environment to execute programs.
Manages computer resources effectively.

**Examples**

Windows, Linux, macOS, Android, iOS.

# Computer System Structure

## Hardware

Provides basic computing resources (CPU, memory, I/O devices).

## Operating System

Controls and coordinates the use of hardware among application programs.

## Application Programs

Define the ways in which the system resources are used to solve computing problems of the users.

## Users

People, machines, other computers.

# Goals of an Operating System

## Convenience

Make the computer system convenient and easy to use for the user.

## Efficiency

Allow the computer system hardware to be used in an efficient manner.

## Ability to evolve

Construct the OS in a way as to permit effective development, testing, and introduction of new system functions.

# OS as a Resource Manager

**Manages all resources**

CPU time, memory space, file storage space, I/O devices.

Decides between conflicting requests for efficient and fair resource use.
Allocates resources to specific programs and users as necessary for their tasks.

# OS as a Control Program

Controls execution of programs to prevent errors and improper use of the computer.
Oversees the operation of I/O devices.
Provides security and access control.

# Primary Functions of OS - Part 1

## Process Management

Creating, scheduling, and terminating processes.

## Memory Management

Keeping track of which parts of memory are currently being used and allocating/deallocating memory space.

## File Management

Creating, deleting, and organizing files and directories.

# Primary Functions of OS - Part 2

## Device/I/O Management

Managing device drivers and controlling I/O operations.

## Security and Protection

Controlling access to resources and protecting data from unauthorized access.

## Networking

Facilitating communication over a network.

# OS Viewpoints

**User View**

Focuses on ease of use, performance, and ignoring resource utilization.

**System View**

Views the OS as a resource allocator and control program.

Different types of systems have different viewpoints (e.g., embedded systems vs mainframes).

# Summary

An OS is the interface between the user and hardware.
Main goals are convenience and efficiency.

**Key roles**

Resource Manager and Control Program.

Functions include Process, Memory, File, and I/O management.

# Evolution of Operating System

Course: Operating Systems

## Unit 1

Introduction to Operating System

## Lecture 2

Evolution of Operating System

# Agenda

Why study OS History?

## First Generation

No OS

## Second Generation

Batch Systems

## Third Generation

Multiprogramming

## Fourth Generation

Modern OS

Serial Processing vs Batch Processing
Concept of Buffering
Concept of Spooling
Summary of Evolution

# Why study the History of OS?

To understand how fundamental concepts of OS were developed.
To see how hardware evolution drove software evolution.
To appreciate the complexity of modern operating systems.
Many "old" concepts are still used today in different forms.

# First Generation (1945-1955): No OS

**Hardware**

Vacuum tubes and plugboards.

No operating system existed.
Programmers interacted directly with hardware.

**Serial Processing**

One job at a time, requiring manual setup and tear-down.

Highly inefficient use of expensive computer resources.

# Second Generation (1955-1965): Batch Systems

**Hardware**

Transistors and Mainframes (e.g., IBM 7094).

**Introduction of the first OS**

Batch System.

Jobs with similar needs were grouped and run together.
Reduced setup time.
Operator took over the machine setup tasks from the programmer.

# Serial Processing vs Batch Processing

## Serial Processing

User has complete access to machine, slow setup time, lots of idle CPU time.

## Batch Processing

Jobs grouped into batches, executed automatically one after another.

## Resident Monitor

The first primitive OS that automatically transferred control from one job to the next.

# Third Generation (1965-1980): Multiprogramming

**Hardware**

Integrated Circuits (ICs).

CPU was still often idle in batch systems during I/O operations.

**Multiprogramming was introduced**

Keep multiple jobs in memory.

When one job waits for I/O, OS switches CPU to another job.
Time-sharing systems also emerged in this era.

# Concept of Buffering

**Problem**

CPU is much faster than I/O devices.

**Solution**

Read data ahead of time into a buffer in main memory.

CPU computes data from the buffer while I/O device fills another part of the buffer.
Overlaps I/O of a single job with its own computation.

# Concept of Spooling

**SPOOL**

Simultaneous Peripheral Operations On-Line.

Uses disk as a very large buffer.
Read jobs from card reader directly to disk.
When a job executes, it reads its input from the disk.
When a job prints, it writes its output to the disk, which is later sent to the printer.

# Fourth Generation (1980-Present): Modern OS

**Hardware**

Large Scale Integration (LSI) and VLSI - Personal Computers.

Development of Network Operating Systems.
Rise of Graphical User Interfaces (GUIs).
Distributed Operating Systems and Mobile Operating Systems (Android, iOS).
Focus on user convenience, networking, and security.

# Summary

OS evolved from no OS to batch systems, to multiprogramming, to modern PC/Network OS.
Each evolution aimed to improve CPU utilization and user convenience.
Buffering and Spooling were key innovations to overlap CPU and I/O.
Hardware advancements directly enabled OS advancements.

# Types of OS: Batch, Time-Sharing, Multiprogramming

Course: Operating Systems

## Unit 1

Introduction to Operating System

## Lecture 3

Types of OS (Part 1)

# Agenda

Simple Batch Operating Systems
Pros and Cons of Batch OS
Multiprogramming Operating Systems
Requirements for Multiprogramming
Time-Sharing Operating Systems
Time-Sharing vs Multiprogramming
Pros and Cons of Time-Sharing OS

# Simple Batch Operating Systems

Users do not interact with the computer directly.
User prepares a job (program, data, control information) and submits it to a computer operator.
Operator groups similar jobs into batches and runs them.
Output is collected later by the user.
Relies on a Resident Monitor to sequence jobs.

# Advantages and Disadvantages of Batch OS

## Advantages

Moves much of the work of the operator to the computer. Good for large, repetitive tasks.

## Disadvantages

Lack of interaction between user and job.

## Disadvantages

CPU is often idle because I/O speeds are much slower than CPU speeds.

## Disadvantages

Difficult to debug programs.

# Multiprogrammed Operating Systems

Designed to maximize CPU utilization.
Keeps several jobs in memory simultaneously.
When one job needs to wait for I/O, the OS switches the CPU to another job.
Ensures that the CPU is never idle as long as there is a job ready to execute.

# How Multiprogramming Works

A subset of total jobs is kept in memory.
One job is selected and run via Job Scheduling.
If it waits (e.g., for I/O), the OS saves its state and switches to another job.
When the I/O finishes, the first job becomes ready to run again.

# Requirements for Multiprogramming

## Memory Management

Must allocate memory to several jobs at once.

## CPU Scheduling

Must choose which ready job to run next.

## Disk Storage

Requires backing store (disk) for jobs not currently in memory.

## Protection

Must protect jobs from interfering with one another.

# Time-Sharing Operating Systems

Also known as Multitasking systems.
A logical extension of multiprogramming.
CPU executes multiple jobs by switching among them very frequently.
Provides each user with the illusion that they have their own computer.
Response time should be very short (e.g., ¡ 1 second).

# Time-Sharing vs Multiprogramming

**Multiprogramming Objective**

Maximize CPU utilization.

**Time-Sharing Objective**

Minimize response time for users.

In Time-Sharing, switches happen due to time slices (quanta) expiring, not just I/O waits.
Time-Sharing requires an interactive computer system.

# Advantages and Disadvantages of Time-Sharing

## Advantages

Provides quick response time, avoids duplication of software, reduces CPU idle time.

## Disadvantages

Problem of reliability (if OS crashes, all users are affected).

## Disadvantages

Question of security and integrity of user programs and data.

## Disadvantages

Data communication problems.

# Summary

Batch OS processes grouped jobs without user interaction.
Multiprogramming keeps CPU busy by switching jobs during I/O waits.
Time-sharing switches jobs rapidly to provide interactive response times to multiple users.

# Types of OS: Multiprocessing, RTOS, Network

Course: Operating Systems

## Unit 1

Introduction to Operating System

## Lecture 4

Types of OS (Part 2)

# Agenda

Multiprocessing Operating Systems
Symmetric vs Asymmetric Multiprocessing
Multitasking Systems
Real-Time Operating Systems (RTOS)
Hard vs Soft Real-Time Systems
Network Operating Systems
Distributed Operating Systems

# Multiprocessing Operating Systems

Systems with more than one CPU in close communication.
Also known as parallel systems or tightly coupled systems.
Share the computer bus, clock, memory, and peripheral devices.

**Advantages**

Increased throughput, economy of scale, increased reliability (graceful degradation/fault tolerance).

# Symmetric vs Asymmetric Multiprocessing

## Asymmetric Multiprocessing (AMP)

Each processor is assigned a specific task. A master processor controls the system.

## Symmetric Multiprocessing (SMP)

Each processor performs all tasks within the OS. All processors are peers.

SMP is the most common multiprocessor system today.

# Multitasking Systems

Similar to time-sharing but generally applied to single-user personal computers.
Allows a user to run multiple applications simultaneously (e.g., browser, word processor, media player).

## Preemptive Multitasking

OS interrupts a process to give CPU to another.

## Cooperative Multitasking

Process must voluntarily yield the CPU.

# Real-Time Operating Systems (RTOS)

Used when rigid time requirements have been placed on the operation of a processor or the flow of data.
Often used as a control device in a dedicated application.
Must process data and respond without any delay.

**Examples**

Medical imaging systems, industrial control systems, weapon systems.

# Hard vs Soft Real-Time Systems

**Hard Real-Time System**

Guarantees that critical tasks complete on time. Missing a deadline causes total system failure (e.g., pacemaker).

Secondary storage is limited or absent, data is stored in ROM.
Soft Real-Time System: A critical real-time task gets priority over other tasks, but missing a deadline doesn't cause total failure (e.g., video streaming).
Useful in multimedia, virtual reality.

# Network Operating Systems

Runs on a server and provides the server the capability to manage data, users, groups, security, applications, and networking functions.
Users are aware of the existence of multiple computers.
Each computer has its own local operating system.

**Examples**

Linux, Windows Server.

# Distributed Operating Systems

Users are not aware of where their programs are running; it appears as a single system.
Computation, data, and resources are distributed across multiple physical machines.

**Advantages**

Resource sharing, computation speedup, reliability, communication.

Highly complex to design and implement.

# Summary

Multiprocessing increases throughput using multiple CPUs.
RTOS are designed for systems with strict time constraints.
Network OS provides shared resources over a network.
Distributed OS hides the network, presenting a unified system view.

# Services of Operating System

Course: Operating Systems

## Unit 1

Introduction to Operating System

## Lecture 5

Services of OS

# Agenda

Overview of OS Services
User Interface (UI)
Program Execution
I/O Operations
File-System Manipulation
Communications
Error Detection
Resource Allocation
Accounting
Protection and Security

# Overview of OS Services

An operating system provides an environment for the execution of programs.
It provides certain services to programs and to the users of those programs.

**Services can be categorized into two sets**

- Services helpful to the user.
- Services ensuring the efficient operation of the system itself.

# User Interface (UI)

Almost all operating systems have a user interface.

## Command-Line Interface (CLI)

Uses text commands and a method for entering them.

## Graphical User Interface (GUI)

A window system with a pointing device to direct I/O, choose from menus, etc.

## Batch Interface

Commands and directives are entered into files, which are then executed.

# Program Execution

The OS must be able to load a program into memory and run that program.
The program must be able to end its execution, either normally or abnormally (indicating error).
OS handles all the background tasks required to start a process.

# I/O Operations

A running program may require I/O (input/output), which may involve a file or an I/O device.
For efficiency and protection, users usually cannot control I/O devices directly.
Therefore, the operating system must provide a means to do I/O.

# File-System Manipulation

Programs need to read and write files and directories.
They also need to create and delete them by name, search for a given file, and list file information.
OS provides permission management to allow or deny access to files or directories based on file ownership.

# Communications

There are many circumstances in which one process needs to exchange information with another process.
Such communication may occur between processes executing on the same computer or on different computer systems tied together by a network.

**Methods**

Shared memory or message passing.

# Error Detection

The OS needs to be constantly aware of possible errors.
Errors may occur in the CPU and memory hardware, in I/O devices, or in the user program.
For each type of error, the OS should take the appropriate action to ensure correct and consistent computing.

# Resource Allocation & Accounting

Resource Allocation: When multiple users or multiple jobs run concurrently, resources must be allocated to each of them (CPU cycles, main memory, file storage).

**Accounting**

We want to keep track of which users use how much and what kinds of computer resources.

Accounting data can be used for billing or for accumulating usage statistics.

# Protection and Security

Protection involves ensuring that all access to system resources is controlled.
Security of the system from outsiders requires user authentication, extending to defending external I/O devices from invalid access attempts.
If a system is to be protected and secure, precautions must be instituted throughout it.

# Summary

OS services provide a user-friendly programming and execution environment.

## Key user services

UI, Program Execution, I/O, File System, Communication, Error Detection.

## Key system services

Resource Allocation, Accounting, Protection, and Security.

# Components of OS: Kernel, Shell, System Calls

Course: Operating Systems

## Unit 1

Introduction to Operating System

## Lecture 6

OS Components

# Agenda

Core Structure of an OS
What is a Kernel?
Functions of a Kernel
Types of Kernels
What is a Shell?
Kernel vs Shell
What are System Calls?
System Call Interface (API)
How a System Call Works
Types of System Calls

# Core Structure of an OS

An operating system can be viewed logically in layers.
Hardware sits at the bottom layer.
The Kernel is the core layer interacting directly with hardware.
The Shell is the outer layer interacting with the user.
Applications sit on top of the OS, using System Calls to request services.

# What is a Kernel?

The kernel is the central component of a computer operating system.
It is the only part of the OS that is always resident in memory.
It has complete control over everything in the system.
It acts as a bridge between applications and the actual data processing done at the hardware level.

# Functions of a Kernel

## Memory Management

Keep track of how much memory is used and by whom.

## Process Management

Determine which processes can use the CPU, when, and for how long.

## Device Drivers

Act as an intermediary between the OS and hardware devices.

## System Calls and Security

Receive service requests from processes.

# Types of Kernels

## Monolithic Kernel

All OS services run along with the main kernel thread in the same memory space. Fast, but less secure (e.g., Linux, UNIX).

Microkernel: Only core essential services (memory/process management) are in the kernel. Other services run in user space. Slower, but highly stable and secure (e.g., QNX, MINIX).

## Hybrid Kernel

Combines aspects of both (e.g., Windows NT, macOS).

# What is a Shell?

A shell is a software program that provides an interface for users to access the services of a kernel.
It translates human-readable commands into kernel-understandable language.

## Command-Line Shells

bash, sh, csh, PowerShell.

## Graphical Shells

Windows Desktop, GNOME, KDE.

# Kernel vs Shell

Kernel is the inner layer; Shell is the outer layer.
Kernel interacts with hardware; Shell interacts with the user.
Kernel performs the actual work (CPU, memory management); Shell takes user commands and passes them to the kernel.
A system has only one running kernel but can have multiple shells running simultaneously.

# What are System Calls?

A system call is the programmatic way in which a computer program requests a service from the kernel.
It provides an interface to the services made available by an operating system.
Usually accessed via high-level Application Programming Interfaces (APIs) rather than direct system call use.

# How a System Call Works

## Dual-Mode Operation

User mode and Kernel mode.

When an app requests a service, it triggers a software interrupt (trap).
The CPU switches from User mode to Kernel mode.
The OS executes the requested service.
The CPU switches back to User mode and returns control to the app.

# Types of System Calls

## Process Control

end, abort, load, execute, create process, terminate process.

## File Management

create file, delete file, open, close, read, write.

## Device Management

request device, release device, read, write.

## Information Maintenance

get time/date, set time/date, get system data.

## Communications

create/delete communication connection, send/receive messages.

# Summary

The Kernel is the core OS component managing hardware resources.
The Shell is the interface allowing users to interact with the Kernel.
System Calls are the programming interface used by applications to request OS services.
System calls facilitate safe transitions between User Mode and Kernel Mode.

# Concept of Process, Process States and PCB

Course: Operating Systems

## Unit 2

Process Management and Inter Process Communication

## Lecture 7

Concept of Process, Process States and PCB

# What is a Process?

A process is a program in execution.
A program is a passive entity (executable file on disk).
A process is an active entity, with a program counter specifying the next instruction to execute and a set of associated resources.
A single program can be associated with multiple processes (e.g., multiple users running the same web browser).

# Process in Memory

## Text Section

The executable code.

## Data Section

Global variables.

## Heap Section

Memory dynamically allocated during run time.

## Stack Section

Temporary data (function parameters, return addresses, local variables).

# Process State Concept

As a process executes, it changes state.
The state of a process is defined in part by the current activity of that process.
At any given time, a process can be in exactly one state.
The OS uses these states to manage and schedule process execution efficiently.

# The 5-State Process Model

**New**

The process is being created.

**Running**

Instructions are being executed.

**Waiting (Blocked)**

The process is waiting for some event to occur (e.g., I/O completion).

**Ready**

The process is waiting to be assigned to a processor.

**Terminated**

The process has finished execution.

# State Transition Diagram

**New → Ready**

Process admitted to the system.

**Ready → Running**

Dispatcher selects the process for execution.

**Running → Ready**

Interrupt (e.g., time slice expired).

**Running → Waiting**

I/O or event wait request.

**Waiting → Ready**

I/O or event completion.

**Running → Terminated**

Process exits.

# The 7-State Process Model

Extends the 5-state model by adding suspend states.

### Ready Suspend

Process is ready to execute but is swapped out to secondary memory (disk).

### Waiting Suspend (Blocked Suspend)

Process is waiting for an event and has been swapped to disk.

Used when main memory is full to make room for other processes.

# Process Control Block (PCB)

Also known as Task Control Block (TCB).
A data structure used by the OS to store all information about a specific process.
Serves as the repository for any information that may vary from process to process.
Each process has its own unique PCB.

# Information Stored in PCB

**Process State**

Current state (ready, running, etc.).

**Program Counter**

Address of the next instruction to execute.

**CPU Registers**

Contents of all process-centric registers.

**CPU Scheduling Information**

Priorities, scheduling queue pointers.

**Memory-Management Information**

Page tables, base/limit registers.

**Accounting Information**

CPU used, clock time elapsed, process ID.

**I/O Status Information**

I/O devices allocated, list of open files.

# Context Switching

When CPU switches to another process, the system must save the state of the old process.
The state is saved in the PCB of the old process.
The system then loads the saved state for the new process via its PCB.
Context-switch time is pure overhead; the system does no useful work while switching.
Speed depends heavily on memory speed, number of registers, and hardware support.

# Process Creation and Termination

Parent process create children processes, which, in turn create other processes, forming a tree of processes.
Process identified and managed via a process identifier (pid).

**Termination**

Process executes last statement and asks the OS to delete it (e.g., ‘exit()‘ system call).

Parent may terminate execution of children processes (e.g., ‘abort()‘).

# Summary

A process is a program in execution, consisting of code, data, stack, and heap.
Processes transition through various states (New, Ready, Running, Waiting, Terminated).
The OS manages processes using a Process Control Block (PCB) for each process.
Context switching allows a CPU to rapidly switch between processes, creating the illusion of parallel execution.

# Concept of Threads and Multithreading

Course: Operating Systems

## Unit 2

Process Management and Inter Process Communication

## Lecture 8

Concept of Threads and Multithreading

# What is a Thread?

A thread is a basic unit of CPU utilization.
It is sometimes called a lightweight process (LWP).
A thread comprises a thread ID, a program counter, a register set, and a stack.
It shares with other threads belonging to the same process its code section, data section, and other operating-system resources (like open files).

# Process vs Thread

## Heavyweight vs Lightweight

Processes are heavyweight (heavy resource usage); threads are lightweight.

## Resource Sharing

Processes are independent and do not share memory by default. Threads share the memory space of their parent process.

## Context Switching

Switching between processes is expensive. Switching between threads of the same process is much faster.

## Creation Time

Process creation takes more time and OS overhead than thread creation.

# Motivation for Threads

Responsiveness: May allow continued execution if part of a process is blocked (e.g., in a web browser, one thread loads images while another handles user input).

## Resource Sharing

Threads share resources of process, easier than shared memory or message passing.

## Economy

Cheaper to create and context-switch threads.

## Scalability

Process can take advantage of multiprocessor architectures.

# Single and Multithreaded Processes

## Single-threaded process

Has one program counter executing a single sequence of instructions.

## Multithreaded process

Has multiple program counters, executing concurrently.

## Memory Layout

In a multithreaded process, there is one code/data/heap segment, but multiple stack segments (one for each thread).

# Thread Control Block (TCB)

Just as a process has a PCB, a thread has a Thread Control Block (TCB). TCB stores thread-specific information.

**Contains**

Thread ID, Thread state, CPU information (Program Counter, registers), Thread priority, Pointer to the process's PCB.

The PCB contains a list of pointers to the TCBs of its threads.

# User-Level Threads

Thread management is done by a user-level threads library.
The kernel is unaware of the existence of user-level threads.
Fast to create and manage (no kernel intervention required).

**Disadvantage**

If one user-level thread performs a blocking system call, the entire process (and all its threads) is blocked.

# Kernel-Level Threads

Supported and managed directly by the operating system.
The kernel performs thread creation, scheduling, and management in kernel space.
Slower to create and manage compared to user threads (requires system calls).

**Advantage**

If one thread blocks, the kernel can schedule another thread from the same process.

# Multithreading Models

Systems that support both user and kernel threads use multithreading models to map them.

## Many-to-One Model

Many user-level threads mapped to single kernel thread.

## One-to-One Model

Each user-level thread maps to a kernel thread.

## Many-to-Many Model

Allows many user level threads to be mapped to a smaller or equal number of kernel threads.

# Thread Libraries

Provide programmer with API for creating and managing threads.

## POSIX Pthreads

A standard API (not an implementation) commonly used in UNIX-like systems.

## Windows Threads

Thread API native to Windows systems.

## Java Threads

Managed by the JVM, typically implemented using the underlying OS threads.

# Summary

Threads are lightweight processes sharing code, data, and OS resources.
They improve responsiveness, resource sharing, and utilize multiprocessor architectures.
User-level threads are fast but can block the process; kernel-level threads are managed by the OS and don't block the whole process.
Multithreading models (Many-to-One, One-to-One, Many-to-Many) map user threads to kernel threads.

# Process Scheduling Concepts and Criteria

Course: Operating Systems

## Unit 2

Process Management and Inter Process Communication

## Lecture 9

Process Scheduling Concepts and Criteria

# Basic Concepts of Scheduling

**Multiprogramming**

Have some process running at all times to maximize CPU utilization.

In a system with a single CPU core, only one process can run at a time. Others must wait until the CPU is free and can be rescheduled.
The CPU Scheduler is responsible for deciding which ready process gets the CPU next.

# CPU-I/O Burst Cycle

Process execution consists of a cycle of CPU execution and I/O wait.

**CPU Burst**

Time spent actively executing instructions.

**I/O Burst**

Time spent waiting for I/O operations to complete.

Processes alternate between these two states.

**I/O-bound program**

Many short CPU bursts.

**CPU-bound program**

Few very long CPU bursts.

# Types of Schedulers

Long-term Scheduler (Job Scheduler): Selects which processes should be brought into the ready queue from the disk (controls the degree of multiprogramming).

## Short-term Scheduler (CPU Scheduler)

Selects which process should be executed next and allocates CPU (executes very frequently).

## Medium-term Scheduler

Removes processes from memory (swaps to disk) to reduce degree of multi-programming and bring them back later.

# Preemptive vs. Non-preemptive Scheduling

Non-preemptive Scheduling: Once a CPU has been allocated to a process, it keeps the CPU until it releases it either by terminating or switching to the waiting state.
Preemptive Scheduling: The OS can interrupt a running process and force it to release the CPU (e.g., when a time quantum expires or a higher priority process arrives).
Preemptive requires special hardware (timers) and complex coordination to protect shared data.

# The Dispatcher

Dispatcher module gives control of the CPU to the process selected by the short-term scheduler.

**Involves**

- Switching context.
- Switching to user mode.
- Jumping to the proper location in the user program to restart that program.

**Dispatch Latency**

Time it takes for the dispatcher to stop one process and start another running.

# Scheduling Criteria (Part 1)

**To compare different scheduling algorithms, we use various criteria**

**CPU Utilization**

Keep the CPU as busy as possible (from 0% to 100%).

**Throughput**

Number of processes that complete their execution per time unit.

Both of these criteria should ideally be maximized.

# Scheduling Criteria (Part 2)

## Turnaround Time

Amount of time to execute a particular process (time of submission to time of completion).

## Waiting Time

Amount of time a process has been waiting in the ready queue.

## Response Time

Amount of time it takes from when a request was submitted until the first response is produced (important for interactive systems).

# Optimization Goals

Max CPU utilization.
Max throughput.
Min turnaround time.
Min waiting time.
Min response time.
In most cases, we optimize the average measure. However, sometimes optimizing the minimum or maximum values is preferred (e.g., guarantee a maximum response time).

# Context Switch Time Overhead

Context switching takes time.
If scheduling happens too frequently, CPU spends more time context switching than doing useful work.
This limits how often a preemptive scheduler should intervene.
Must balance responsiveness (frequent switching) with throughput (infrequent switching).

# Summary

Scheduling is essential for multiprogramming.
Processes have CPU and I/O bursts.
Schedulers exist at long, medium, and short-term levels.
Scheduling can be preemptive or non-preemptive.
Key criteria include CPU utilization, throughput, turnaround time, waiting time, and response time.

# Process Scheduling Algorithms

Course: Operating Systems

## Unit 2

Process Management and Inter Process Communication

## Lecture 10

Process Scheduling Algorithms

# First-Come, First-Served (FCFS)

The simplest CPU-scheduling algorithm.
The process that requests the CPU first is allocated the CPU first.

## Implementation

easily managed with a FIFO queue.

## Type

Non-preemptive. Once a process has the CPU, it keeps it until it releases it.

## Problem

Average waiting time is often quite long.

# FCFS Example & Convoy Effect

## Example

P1 (24ms burst), P2 (3ms burst), P3 (3ms burst).

## Order P1, P2, P3

Waiting time for P1=0, P2=24, P3=27. Average = 17ms.

## Order P2, P3, P1

Waiting time for P2=0, P3=3, P1=6. Average = 3ms.

## Convoy Effect

Short processes waiting behind one long process. Leads to lower CPU and device utilization.

# Shortest-Job-First (SJF)

Associates with each process the length of its next CPU burst.
Use these lengths to schedule the process with the shortest time.

**SJF is optimal**

Gives minimum average waiting time for a given set of processes.

**Difficulty**

Knowing the length of the next CPU request in advance is practically impossible.

# SJF: Preemptive vs Non-preemptive

**Non-preemptive SJF**

Once CPU given to the process it cannot be preempted until completes its CPU burst.

Preemptive SJF (Shortest-Remaining-Time-First or SRTF): If a new process arrives with CPU burst length less than remaining time of current executing process, preempt.
SRTF generally provides even lower average waiting time than non-preemptive SJF.

# Round Robin (RR) Scheduling

Designed especially for time-sharing systems.
Similar to FCFS, but preemption is added to switch between processes.

**Time Quantum (or Time Slice)**

A small unit of time (usually 10-100 ms).

Ready queue treated as a circular queue.
CPU scheduler goes around the ready queue, allocating the CPU to each process for a time interval of up to 1 time quantum.

# Round Robin Performance

Performance depends heavily on the size of the time quantum (q).

**If q is very large**

RR approaches FCFS behavior.

**If q is very small**

Frequent context switches; CPU spends too much time on overhead.

**Rule of thumb**

80% of CPU bursts should be shorter than the time quantum.

# Priority Scheduling

A priority number (integer) is associated with each process.
CPU allocated to the process with the highest priority (e.g., smallest integer = highest priority).
Can be Preemptive or Non-preemptive.
SJF is a special case of priority scheduling where priority is the inverse of predicted next CPU burst time.

# Starvation and Aging

**Problem with Priority Scheduling**

Starvation (or Indefinite Blocking).

Low priority processes may never execute if high priority processes keep arriving.

**Solution**

Aging.

Aging involves gradually increasing the priority of processes that wait in the system for a long time.

# Multilevel Queue Scheduling

Ready queue is partitioned into separate queues, e.g., foreground (interactive) and background (batch).
Each queue has its own scheduling algorithm (Foreground - RR, Background - FCFS).

**Scheduling must be done between the queues**

Fixed priority scheduling (e.g., serve all from foreground then background).

**Time slice**

Each queue gets a certain amount of CPU time which it can schedule amongst its processes (e.g., 80% to foreground, 20% to background).

# Multilevel Feedback Queue

A process can move between the various queues; aging can be implemented this way.

**Parameters defining a Multilevel Feedback Queue scheduler**

- Number of queues.
- Scheduling algorithms for each queue.
- Method used to determine when to upgrade a process.
- Method used to determine when to demote a process.

# Summary

FCFS is simple but suffers from convoy effect.
SJF gives minimum average waiting time but relies on predicting burst times.
Round Robin ensures fair CPU sharing and responsiveness using time quanta.
Priority scheduling needs aging to prevent starvation.
Multilevel feedback queues offer the most complex and adaptable scheduling mechanism.

# Inter Process Communication & Synchronization

Course: Operating Systems

## Unit 2

Process Management and IPC

## Lecture 11

IPC, Race Conditions, and Critical Section Problem

# Inter Process Communication (IPC)

Processes executing concurrently may be either independent processes or cooperating processes.
Cooperating process can affect or be affected by other processes, including sharing data.

**Reasons for cooperating processes**

Information sharing, Computation speedup, Modularity, Convenience.

IPC provides a mechanism to allow processes to communicate and synchronize their actions.

# Models of IPC

## Two fundamental models of IPC

- Shared Memory: A region of memory that is shared by cooperating processes is established.
- Fast, requires synchronization by the user processes.
- Message Passing: Communication takes place by means of messages exchanged between the cooperating processes.
- Better for smaller amounts of data, easier to implement in distributed systems.

# Process Synchronization Introduction

Concurrent access to shared data may result in data inconsistency. Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes.

**Classic example**

The Producer-Consumer problem.

If a Producer and Consumer share a counter variable, simultaneous updates can lead to incorrect counter values.

# The Race Condition

A race condition occurs when several processes access and manipulate the same data concurrently.
The outcome of the execution depends on the particular order in which the access takes place.
To prevent race conditions, concurrent processes must be synchronized.
Only one process at a time should be allowed to manipulate the shared variable.

# The Critical Section Problem

Consider a system of n processes.
Each process has a segment of code called a Critical Section (CS).
In the CS, the process may be changing common variables, updating tables, writing files, etc.

**The Problem**

Ensure that when one process is executing in its critical section, no other process is allowed to execute in its critical section.

# Requirements for a Solution

**A solution to the critical-section problem must satisfy three requirements**

- Mutual Exclusion: If process Pi is executing in its CS, then no other processes can be executing in their CS.
- Progress: If no process is in its CS and some processes wish to enter, the selection of the next process cannot be postponed indefinitely.
- Bounded Waiting: A bound must exist on the number of times that other processes are allowed to enter their CS after a process has made a request to enter its CS.

# Peterson's Solution

A classic software-based solution to the critical-section problem for 2 processes.

**Two processes share two variables**

- ‘int turn;‘ (indicates whose turn it is)
- ‘boolean flag[2];‘ (indicates if a process is ready to enter CS)

Process i sets ‘flag[i] = true‘ and ‘turn = j‘. It waits while ‘flag[j]‘ is true AND ‘turn == j‘.

# Hardware Support for Synchronization

Software solutions like Peterson's are complex and may fail on modern hardware.
Many systems provide hardware support for critical section code.

**Uniprocessors**

Could disable interrupts (too dangerous for user programs).

Modern machines provide special atomic hardware instructions.
Atomic means non-interruptible.

# Atomic Instructions

**TestAndSet Instruction**

Reads a boolean variable and sets it to true, all as one unbreakable step.

**Swap Instruction**

Swaps the contents of two variables atomically.

These instructions can be used to implement simple locks.
If a lock is 0 (free), TestAndSet returns 0 and sets it to 1. Process enters CS.
If lock is 1 (busy), TestAndSet returns 1. Process waits (spins).

# Summary

IPC allows processes to share data via shared memory or message passing.
Concurrent access to shared data leads to race conditions.
The Critical Section problem requires Mutual Exclusion, Progress, and Bounded Waiting.
Peterson's algorithm is a software solution for two processes.
Hardware atomic instructions (TestAndSet) provide robust mechanisms for building locks.

# Mutex Locks, Semaphores and Classic Problems

Course: Operating Systems

## Unit 2

Process Management and IPC

## Lecture 12

Mutual Exclusion, Semaphores, and Synchronization Problems

# Mutex Locks

Mutex stands for Mutual Exclusion.
It is the simplest synchronization tool provided by the OS API.
A process acquires a lock before entering a critical section; it releases the lock when it exits.
‘acquire()‘ and ‘release()‘ functions.
Typically implemented using hardware instructions (TestAndSet).

# Spinlocks

If a mutex requires a process to loop continuously while waiting for the lock, it is called a spinlock.

**Advantage**

No context switch is required when a process waits.

**Disadvantage**

Wastes CPU cycles if the lock is held for a long time.

Best used in multiprocessor systems where locks are held for very short durations.

# Semaphores

A synchronization tool that provides more sophisticated ways (than Mutex) for processes to synchronize their activities.
Introduced by Edsger Dijkstra.
A Semaphore S is an integer variable.

**Can only be accessed via two indivisible (atomic) operations**

‘wait()‘ (originally ‘P‘, for proberen - to test)
‘signal()‘ (originally ‘V‘, for verhogen - to increment)

# Binary vs Counting Semaphores

**Counting Semaphore**

Integer value can range over an unrestricted domain.

Used to control access to a given resource consisting of a finite number of instances.
Initialize semaphore to the number of available resources.

**Binary Semaphore**

Integer value can range only between 0 and 1.

Functions identically to a Mutex lock.

# Semaphore Implementation (No Busy Waiting)

To avoid spinlocks, semaphores are implemented with a waiting queue.
If a process executes ‘wait()‘ and semaphore is not positive, it blocks itself and is placed in the wait queue.
When another process executes ‘signal()‘, one process is removed from the wait queue and awakened (state changed to ready).
OS handles the blocking and waking mechanisms.

# Deadlock and Starvation

**Deadlock**

Two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes.

(e.g., P0 waits for a semaphore held by P1, while P1 waits for a semaphore held by P0).

**Starvation**

Indefinite blocking. A process may never be removed from the semaphore queue in which it is suspended.

# Classic Problem: Bounded-Buffer

Also called Producer-Consumer problem.
A pool of n buffers, each capable of holding one item.
Producer produces items, consumer consumes them.

## Synchronization needs

- Mutex to protect buffer access.
- Semaphore 'empty' (counts empty slots, blocks producer if 0).
- Semaphore 'full' (counts full slots, blocks consumer if 0).

# Classic Problem: Readers-Writers

A dataset is shared among a number of concurrent processes.

**Readers**

only read the data set; they do not perform any updates.

**Writers**

can both read and write.

Rules: Allow multiple readers to read simultaneously. Only one single writer can access the shared data at a time. No readers can be reading while a writer is writing.

# Classic Problem: Dining Philosophers

5 philosophers sitting at a table with 5 chopsticks.
A philosopher needs 2 chopsticks to eat.
If all philosophers pick up their left chopstick simultaneously, they will wait forever for the right one (Deadlock).
Illustrates the difficulty of allocating multiple resources to multiple processes without deadlock or starvation.

# Summary

Mutexes are binary locks for critical sections.
Semaphores (counting and binary) provide more flexible synchronization.
Implementation using wait queues avoids wasteful busy-waiting.
Classic problems (Bounded-Buffer, Readers-Writers, Dining Philosophers) illustrate common synchronization scenarios and pitfalls.

# Deadlocks: Characteristics, Prevention and Avoidance

Course: Operating Systems

## Unit 2

Process Management and IPC

## Lecture 13

Deadlocks: Characteristics, Prevention and Avoidance

# The Deadlock Problem

A set of blocked processes each holding a resource and waiting to acquire a resource held by another process in the set.

**Example**

System has 2 disk drives. P1 and P2 each hold one drive and each needs another one.

**Result**

System comes to a halt. Manual intervention (killing a process) is often required.

# Deadlock Characterization

**Deadlock can arise if four conditions hold simultaneously (Coffman conditions)**

- Mutual exclusion: At least one resource must be held in a non-sharable mode.
- Hold and wait: A process holding at least one resource is waiting to acquire additional resources held by other processes.
- No preemption: A resource can be released only voluntarily by the process holding it.
- Circular wait: There exists a set $\{P0, P1, ..., P0\}$ of waiting processes where P0 is waiting for a resource held by P1, ..., and Pn is waiting for P0.

# Resource-Allocation Graph (RAG)

A directed graph used to precisely describe deadlocks.

## Vertices V

- P = {P1, P2, ..., Pn}, the set consisting of all processes.
- R = {R1, R2, ..., Rm}, the set consisting of all resource types.

## Edges E

- Request edge: directed edge P1 $\rightarrow$ Rj
- Assignment edge: directed edge Rj $\rightarrow$ Pi

# Cycles in RAG

If graph contains no cycles $\rightarrow$ no deadlock.

**If graph contains a cycle**

- If only one instance per resource type, then deadlock.
- If several instances per resource type, possibility of deadlock (cycle is a necessary but not sufficient condition).

# Methods for Handling Deadlocks

- Ensure the system will never enter a deadlock state (Prevention or Avoidance).
- Allow the system to enter a deadlock state and then recover (Detection and Recovery).
- Ignore the problem and pretend deadlocks never occur in the system (used by most operating systems, including UNIX and Windows).

# Deadlock Prevention

## Goal

Restrain the ways request can be made to invalidate one of the four necessary conditions.

## Mutual Exclusion

Not required for sharable resources (e.g., read-only files); must hold for non-sharable resources.

Hold and Wait: Require process to request and be allocated all its resources before it begins execution, or allow process to request resources only when it has none. (Low resource utilization, starvation possible).

# Deadlock Prevention (Cont.)

## No Preemption

- If a process holding some resources requests another resource that cannot be immediately allocated, all resources currently being held are preempted (released).

## Circular Wait

- Impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration.

# Deadlock Avoidance

Requires that the system has some additional a priori information available.
Simplest and most useful model requires that each process declare the maximum number of resources of each type that it may need.
The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure there can never be a circular-wait condition.

# Safe State

When a process requests an available resource, system must decide if immediate allocation leaves the system in a safe state.
System is in safe state if there exists a sequence ¡P1, P2, ..., Pn¿ of ALL the processes such that for each Pi, the resources Pi can still request can be satisfied by currently available resources + resources held by all the previous Pj.
If a system is in safe state $\rightarrow$ no deadlocks.
If a system is in unsafe state $\rightarrow$ possibility of deadlock.

# Summary

Deadlock requires Mutual Exclusion, Hold & Wait, No Preemption, and Circular Wait.
Resource Allocation Graphs (RAG) visualize resource states; cycles indicate potential deadlocks.
Prevention algorithms structurally prevent one of the four conditions.
Avoidance algorithms dynamically assess requests to keep the system in a 'Safe State'.

# Deadlocks: Banker's Algorithm, Detection and Recovery

Course: Operating Systems

## Unit 2

Process Management and IPC

## Lecture 14

Banker's Algorithm, Deadlock Detection, and Recovery

# Banker's Algorithm

A deadlock avoidance algorithm applicable to systems with multiple instances of each resource type.
Less efficient than the resource-allocation graph algorithm (which is for single instances).
Each process must a priori claim maximum use.
When a process requests a resource it may have to wait if allocation makes system unsafe.
When a process gets all its resources it must return them in a finite amount of time.

# Banker's Algorithm: Data Structures

Let n = number of processes, and m = number of resources types.

## Available

Vector of length m. If Available[j] = k, there are k instances of resource type Rj available.

## Max

n x m matrix. If Max[i,j] = k, then process Pi may request at most k instances of Rj.

## Allocation

n x m matrix. If Allocation[i,j] = k, Pi is currently allocated k instances of Rj.

## Need

n x m matrix. If Need[i,j] = k, Pi may need k more instances of Rj to complete. (Need[i,j] = Max[i,j] - Allocation[i,j])

# Safety Algorithm

- Let Work and Finish be vectors of length m and n. Initialize: Work = Available; Finish[i] = false for i = 0, 1, ..., n-1.
- Find an i such that both:

a) Finish[i] == false
b) Need_i ¡= Work
If no such i exists, go to step 4.

- Work = Work + Allocation_i; Finish[i] = true; go to step 2.
- If Finish[i] == true for all i, then the system is in a safe state.

# Resource-Request Algorithm for Process Pi

Request_i = request vector for process Pi.

- If Request_i ¡= Need_i, go to step 2. Otherwise, raise error condition (exceeded max claim).
- If Request_i ¡= Available, go to step 3. Otherwise Pi must wait (resources not available).
- Pretend to allocate requested resources to Pi by modifying the state as follows:

Available = Available - Request_i;
Allocation_i = Allocation_i + Request_i;
Need_i = Need_i - Request_i;
If safe $\rightarrow$ resources allocated. If unsafe $\rightarrow$ Pi waits, old state is restored.

# Deadlock Detection

If a system does not employ a deadlock-prevention or a deadlock-avoidance algorithm, then a deadlock situation may occur.

**In this environment, the system must provide**

- An algorithm that examines the state of the system to determine whether a deadlock has occurred.
- An algorithm to recover from the deadlock.

# Detection Algorithm Usage

**When, and how often, to invoke depends on**

- How often is a deadlock likely to occur?
- How many processes will need to be rolled back?

Invoking detection frequently adds significant computational overhead.
If invoked arbitrarily, there may be many cycles in the resource graph and we would not be able to tell which of the many deadlocked processes 'caused' the deadlock.

# Recovery from Deadlock: Process Termination

**Two main approaches to break a deadlock by terminating processes**

- Abort all deadlocked processes: Simple but very costly (lots of work thrown away).
- Abort one process at a time until the deadlock cycle is eliminated: Slower, requires running the detection algorithm after each termination.

Which process to abort? Choose the 'victim' based on priority, computation time, resources used, etc.

# Recovery from Deadlock: Resource Preemption

Take a resource away from one process and give it to another until the deadlock is broken.

**Issues**

- Selecting a victim: Which process and which resource? Cost minimization.
- Rollback: Cannot simply continue. Must return process to a safe state (often just restart it).
- Starvation: Same process might always be picked as victim. Include number of rollbacks in cost factor.

# Summary of Unit 2

Processes and Threads are the fundamental units of execution.
Scheduling algorithms (FCFS, SJF, RR) determine CPU allocation.
Synchronization (Mutex, Semaphores) is vital for coordinating concurrent access to shared resources.
Deadlocks can occur in concurrent systems; they can be prevented, avoided (Banker's Algorithm), or detected and recovered from.

# Process Address Space

Course: Operating Systems

## Unit 3

Memory Management

## Lecture 15

Process Address Space

# Introduction to Memory Management

Memory is a large array of words or bytes, each with its own address.
CPU fetches instructions from memory according to the value of the program counter.
To execute a program, it must be mapped to absolute addresses and loaded into memory.
Memory management is the process of controlling and coordinating computer memory.

# Basic Hardware

CPU can access main memory and registers directly.
Register access is very fast (usually one CPU clock).
Main memory access may take many cycles, causing a stall.
Cache memory is placed between main memory and CPU registers to improve speed.

# Base and Limit Registers

Provide memory protection for each process.

**Base Register**

Holds the smallest legal physical memory address.

**Limit Register**

Specifies the size of the range.

Any memory access by the process is checked against these registers.

# Address Binding

Programs on disk ready to be brought into memory form an input queue.
Addresses in the source program are generally symbolic.

**Binding of instructions and data to memory addresses can happen at**

**Compile time**

If memory location is known a priori.

**Load time**

If memory location is not known at compile time.

**Execution time**

If the process can be moved during execution.

# Logical vs Physical Address Space

**Logical address**

Generated by the CPU (also referred to as virtual address).

**Physical address**

Address seen by the memory unit.

Logical and physical addresses are identical in compile-time and load-time binding.
They differ in execution-time binding.

# Memory-Management Unit (MMU)

Hardware device that at run time maps virtual to physical addresses.
The value in the relocation register (base register) is added to every address generated by a user process.
The user program deals with logical addresses; it never sees the real physical addresses.

# Process Address Space Overview

A process in memory is organized into several segments.
This organization allows the OS to manage memory efficiently.
Provides isolation between different processes.
Facilitates sharing of common code (like libraries).

# Components of Process Address Space

## Text Segment

Contains the executable code (machine instructions).

## Data Segment

Global and static variables (initialized and uninitialized).

## Heap

Dynamic memory allocated during run time (e.g., via malloc).

## Stack

Local variables, function parameters, return addresses.

# Dynamic Loading

Routine is not loaded until it is called.
Better memory-space utilization; unused routines are never loaded.
Useful when large amounts of code are needed to handle infrequently occurring cases.
Does not require special support from the operating system, implemented through program design.

# Dynamic Linking and Shared Libraries

**Static linking**

System libraries are treated like any other object module.

**Dynamic linking**

Linking postponed until execution time.

A stub is used to locate the appropriate memory-resident library routine.
Saves memory and disk space by sharing libraries across multiple processes (Shared Libraries / DLLs).

# Summary

Hardware provides base and limit registers for protection.
Address binding can occur at compile, load, or execution time.
The MMU translates logical addresses to physical addresses at runtime.
A process address space consists of Text, Data, Heap, and Stack segments.

# Swapping and Contiguous Memory Allocation

Course: Operating Systems

## Unit 3

Memory Management

## Lecture 16

Swapping and Contiguous Memory Allocation

# What is Swapping?

A process must be in memory to be executed.
A process can be swapped temporarily out of memory to a backing store.
It can later be brought back into memory for continued execution.
Total physical memory space of processes can exceed physical memory.

# Standard Swapping Process

**Backing store**

Fast disk large enough to accommodate copies of all memory images.

**Roll out, roll in**

A swapping variant used for priority-based scheduling algorithms.

Lower-priority process is swapped out so higher-priority process can be loaded.
Major part of swap time is transfer time.

# Contiguous Memory Allocation

Main memory must support both OS and user processes.

**Memory is divided into two partitions**

one for the resident OS and one for user processes.

In contiguous memory allocation, each process is contained in a single contiguous section of memory.

# Memory Protection in Contiguous Allocation

Relocation registers used to protect user processes from each other.
Base register contains value of smallest physical address.
Limit register contains range of logical addresses.
MMU maps logical address dynamically by adding base register.

# Memory Allocation (Holes and Partitions)

**Multiple-partition allocation**

Memory divided into fixed or variable-sized partitions.

**Hole**

Block of available memory; holes of various sizes are scattered throughout memory.

When a process arrives, it is allocated memory from a hole large enough to accommodate it.
OS maintains information about allocated partitions and free holes.

# First-Fit Allocation Method

Allocate the first hole that is big enough.
Searching can start either at the beginning of the set of holes or where the previous first-fit search ended.
Stops searching as soon as it finds a free hole that is large enough.

# Best-Fit Allocation Method

Allocate the smallest hole that is big enough.
Must search entire list, unless ordered by size.
Produces the smallest leftover hole.

# Worst-Fit Allocation Method

Allocate the largest hole.
Must also search entire list.
Produces the largest leftover hole, which may be more useful than the small holes from best-fit.

# Fragmentation

## External Fragmentation

Total memory space exists to satisfy a request, but it is not contiguous.

## Internal Fragmentation

Allocated memory may be slightly larger than requested memory.

The size difference is memory internal to a partition, but not being used.

# Compaction

A solution to external fragmentation.
Shuffle memory contents to place all free memory together in one large block.
Only possible if relocation is dynamic and done at execution time.
Can be very expensive in terms of I/O and CPU time.

# Summary

Swapping moves processes between memory and disk to free up RAM.
Contiguous allocation places each process in a single block of memory.
First-fit and Best-fit are common allocation algorithms.
External fragmentation is a severe limitation of contiguous allocation, solved by compaction.

# Segmentation

Course: Operating Systems

## Unit 3

Memory Management

## Lecture 17

Segmentation

# Introduction to Segmentation

Memory management scheme that supports user view of memory.
A program is a collection of segments.

**A segment is a logical unit such as**

main program, procedure, function, method, object, local variables, global variables, etc.

# What is a Segment?

Segments are variable-sized blocks of memory.
Each segment has a specific purpose and name (or number).
Addresses specify both the segment name and the offset within the segment.
Different segments can grow independently.

# Segmentation Architecture

**Logical address consists of a two tuple**

¡segment-number, offset¿.

**Segment table**

Maps two-dimensional physical addresses.

**Each table entry has**

- Base: Contains the starting physical address.
- Limit: Specifies the length of the segment.

# Segmentation Hardware

Segment-table base register (STBR) points to the segment table's location in memory.
Segment-table length register (STLR) indicates number of segments used by a program.
Segment number 's' is legal if s ¡ STLR.
Offset 'd' must be between 0 and the limit of the segment.

# Address Translation in Segmentation

- CPU generates logical address (s, d).
- Check if s ¡ STLR (if not, trap).
- Look up segment 's' in segment table to find Base and Limit.
- Check if d ¡ Limit (if not, trap).
- Physical address = Base + d.

# Example of Segmentation

Assume a program with 5 segments (0 to 4).
Segment 2 is 400 bytes long, starts at base 4300.

**Logical address (2, 53) translates to**

Base 4300 + Offset 53 = Physical Address 4353.
Logical address (2, 500) would cause a trap (limit exceeded).

# Protection in Segmentation

Protection bits are associated with each segment.
Read, write, execute privileges can be set independently for each segment.
Code segment can be marked as read-only and execute-only.
Data segment can be marked read-write.

# Sharing in Segmentation

Segments allow sharing of code or data between processes.
Multiple segment tables can point to the same physical base address.
Common for shared libraries (e.g., standard C library).
Shared code must be reentrant (pure code, cannot modify itself).

# Fragmentation in Segmentation

Segmentation eliminates internal fragmentation.
Segments are exactly the size they need to be.
However, segmentation suffers from external fragmentation.
Memory becomes divided into variable-sized holes, requiring algorithms like best-fit or compaction.

# Advantages and Disadvantages

## Advantages

- Aligns with programmer's view of memory.
- Excellent support for sharing and protection.
- No internal fragmentation.

## Disadvantages

- External fragmentation.
- Segment tables can be large.

# Summary

Segmentation maps physical memory to logical program structures.
Logical addresses consist of a segment number and an offset.
Provides robust protection and sharing capabilities.
Suffers from external fragmentation, necessitating compaction.

# Paging Concepts and Hardware

Course: Operating Systems

## Unit 3

Memory Management

## Lecture 18

Paging Concepts and Hardware

# Introduction to Paging

A non-contiguous memory management scheme.
Solves the severe problem of external fragmentation.
Eliminates the need for memory compaction.
Divides physical memory into fixed-sized blocks called Frames.
Divides logical memory into blocks of the same size called Pages.

# Basic Method of Paging

To run a program of size N pages, we need to find N free frames.
The OS keeps track of all free frames.
A page table is used to translate logical to physical addresses.
Any page can be placed into any available frame.

# Address Translation Architecture

**Logical address generated by CPU is divided into two parts**

- Page number (p): Used as an index into a page table.
- Page offset (d): Combined with base address to define the physical address.

Page size is typically a power of 2 (e.g., 4 KB to 8 KB).

# Logical to Physical Translation

Let 'm' be the bits for logical address, and 'n' for page size.
Page offset (d) occupies the lower 'n' bits.
Page number (p) occupies the upper 'm - n' bits.
Physical address is formed by appending offset 'd' to the Frame number 'f'.

# Paging Example

Suppose page size = 4 bytes, Physical memory = 32 bytes (8 frames).
Logical address space = 16 bytes (4 pages).

**Logical Address 5 (binary 01 01)**

- Page number = 1
- Offset = 1

If Page 1 is in Frame 6 (binary 110), Physical address = 110 01 (binary 25).

# Fragmentation in Paging

Paging has NO external fragmentation.
Any free frame can be allocated to any process.
Paging suffers from internal fragmentation.
A process may not require the exact size of a whole page, leaving the last frame partially empty.

# Free Frames Management

OS maintains a free-frame list.
When a process arrives, OS allocates required frames.
When process terminates, frames are returned to the free list.
Frames allocated to a process do not need to be contiguous.

# Paging Hardware Mechanism

Page table is kept in main memory.
Page-table base register (PTBR) points to the page table.
Page-table length register (PTLR) indicates size of the page table.
Every data/instruction access requires TWO memory accesses.

# Paging vs Segmentation

## Paging

- Fixed size blocks.
- Invisible to the user/programmer.
- Suffers internal fragmentation.

## Segmentation

- Variable size blocks.
- Visible to the programmer (logical).
- Suffers external fragmentation.

# Summary

Paging maps logical pages to physical frames.
It eliminates external fragmentation entirely.
Translation is hardware-assisted using a page table.
The main drawback of basic paging is the double memory access overhead.

# Advanced Paging and TLBs

Course: Operating Systems

## Unit 3

Memory Management

## Lecture 19

Advanced Paging and TLBs

# The Problem with Basic Paging

Accessing the page table in main memory requires a memory access.
Accessing the actual data requires a second memory access.
Execution time is doubled, degrading system performance.

**Solution**

Hardware cache called Translation Look-aside Buffer (TLB).

# Translation Look-aside Buffer (TLB)

A special, small, fast-lookup hardware cache.
Contains a few of the most recently accessed page-table entries.
Operates as associative memory (parallel search).
Each entry contains a Page Number and its Frame Number.

# TLB Hit and Miss

**TLB Hit**

Page number is found in TLB.

- Frame number is immediately available. (Fast)

**TLB Miss**

Page number is not in TLB.

- Must access page table in main memory.
- Frame number is fetched and added to TLB for future use. (Slow)

# Effective Access Time (EAT)

**Hit Ratio ($\alpha$)**

Percentage of times a page number is found in TLB.

TLB search time $= \epsilon$.
Memory access time $=$ m.
EAT $=$ (m $+ \epsilon$) * $\alpha$ + (2m $+ \epsilon$) * (1 - $\alpha$)
If $\alpha$ is high (e.g., 99%), EAT is very close to single memory access time.

# Memory Protection in Paging

Implemented by associating protection bits with each frame.
Valid-invalid bit attached to each entry in the page table.

- 'Valid' indicates that the page is in the process's logical address space.
- 'Invalid' indicates that the page is not in the process's address space.

Violations cause hardware traps.

# Shared Pages

Paging makes sharing code very easy.
Shared code must be Reentrant (read-only, non-self-modifying).
Multiple processes point their page tables to the same physical frames.
Each process keeps its own private data pages.

# Structure of the Page Table

Modern systems have large logical address spaces (32-bit or 64-bit).
A 32-bit system with 4KB pages has 1 million page table entries.
Each entry takes ˜4 bytes $\rightarrow$ 4MB continuous memory just for one page table!

**Solutions**

Hierarchical Paging, Hashed Page Tables, Inverted Page Tables.

# Hierarchical Paging

Break up the logical address space into multiple page tables.
A simple technique is a two-level page table.
The page table itself is paged.

**Logical address is divided into**

Outer page number, Inner page number, Offset.

# Hashed Page Tables

Common in address spaces ¿ 32 bits.
The virtual page number is hashed into a page table.
Each entry contains a linked list of elements hashing to the same location.
Search the linked list for a match to get the physical frame.

# Inverted Page Tables

One entry for each physical frame of memory, not logical page.
Entry consists of the virtual address of the page stored in that real memory location.
Decreases memory needed to store each page table.
Increases time needed to search the table when a page reference occurs.

# Unit 3 Summary

Memory Management is vital for system performance.
We moved from Contiguous Allocation to Segmentation.
Paging solves external fragmentation but needs TLBs for speed.
Advanced page table structures support modern 64-bit architectures.

# File Concept and Attributes

Course: Operating Systems

## Unit 4

File Management

## Lecture 20

File Concept and Attributes

# Introduction to File Management

Storage management is a core function of the OS.

**The OS abstracts physical storage properties to define a logical storage unit**

the file.

Files are mapped by the OS onto physical devices (disks, tapes).
Provides a uniform logical view of information storage.

# What is a File?

A file is a contiguous logical address space.
It is a named collection of related information recorded on secondary storage.
Files represent both programs (source and object forms) and data.
Data files may be numeric, alphabetic, alphanumeric, or binary.

# File Structure Concepts

A file has a certain defined structure, which depends on its type.

## Text file

Sequence of characters organized into lines.

## Source file

Sequence of subroutines and functions.

## Object file

Sequence of bytes organized into blocks understandable by the system's linker.

## Executable file

Series of code sections that the loader can bring into memory.

# Logical vs. Physical View

## Logical View

How users and applications see the file (e.g., a continuous stream of bytes or records).

## Physical View

How the file is actually stored on the storage medium (e.g., scattered blocks on a disk).

The OS is responsible for translating between logical and physical views.

# File Attributes Overview

A file is more than just its data; it has properties known as attributes.
Attributes vary from one operating system to another.
They are stored in the directory structure (or file control block).
Common attributes include Name, Identifier, Type, Location, Size, Protection, and Timestamps.

# File Attributes: Name and Identifier

**Name**

The symbolic file name is the only information kept in human-readable form.

**Identifier**

A unique tag, usually a number, identifying the file within the file system.

The identifier is the non-human-readable name for the file.

# File Attributes: Type and Location

**Type**

Needed for systems that support different types of files.

**Location**

A pointer to a device and to the location of the file on that device.

The OS uses location attributes to find the actual data blocks on the disk.

# File Attributes: Size

**Size**

The current size of the file (in bytes, words, or blocks).

Maximum allowed size may also be included.
Size attributes are updated dynamically as the file is written to or truncated.

# File Attributes: Protection and Time

**Protection**

Access-control information determines who can do reading, writing, executing, etc.

**Time, date, and user identification**

Data for protection, security, and usage monitoring.

Timestamps usually include creation time, last modification time, and last access time.

# Storing File Attributes

Information about all files is kept in the directory structure.
The directory structure resides on secondary storage.
A directory entry consists of the file's name and its unique identifier or its attributes directly.
Typically takes more than a kilobyte to store attributes per file in modern systems.

# Summary

A file is a logical abstraction for physical storage.
Files consist of data and structural information.
File attributes (Name, Identifier, Type, Location, Size, Protection, Timestamps) provide essential metadata.
The OS uses these attributes to manage access, security, and physical storage operations.

# File Types and Operations

Course: Operating Systems

## Unit 4

File Management

## Lecture 21

File Types and Operations

# Introduction to File Types

When an OS designs a file system, it must consider whether to recognize and support different file types.
If an OS recognizes the type of a file, it can operate on the file in reasonable ways.

**Example**

If a user tries to print an executable file, the OS can prevent it if it knows the file type.

# Common File Types and Extensions

**Executable**

.exe, .com, .bin (ready to run machine-language program)

**Object**

.obj, .o (compiled, machine language, not linked)

**Source code**

.c, .cpp, .java (source code in various languages)

**Batch**

.bat, .sh (commands to the command interpreter)

**Text**

.txt, .doc (textual data, documents)

**Archive**

.zip, .tar, .rar (various files grouped together, sometimes compressed)

# Magic Numbers

Many operating systems (like UNIX/Linux) use 'magic numbers' to identify file types.
A magic number is a specific sequence of bytes at the beginning of a file.
The OS looks at these bytes to determine the file's actual format, regardless of its extension.

**Example**

A PDF file typically starts with '%PDF'.

# Basic File Operations

A file is an abstract data type. The OS provides system calls to perform operations on it.

**The six basic file operations are**

- Creating a file
- Writing a file
- Reading a file
- Repositioning within a file
- Deleting a file
- Truncating a file

# Creating and Writing a File

## Creating a file

- Find space in the file system.
- Add an entry for the new file in the directory.

## Writing a file

- Specify the file name and the information to be written.
- The OS searches the directory, keeps a write pointer to the location in the file where the next write is to take place.

# Reading and Repositioning

## Reading a file

- Specify the file name and where (in memory) the next block of the file should be put.
- OS maintains a read pointer.

## Repositioning within a file (Seek)

- The directory is searched for the appropriate entry, and the current-file-position pointer is repositioned to a given value.
- Does not involve actual I/O.

# Deleting and Truncating a File

## Deleting a file

- Search the directory for the named file.
- Release all file space and invalidate the directory entry.

## Truncating a file

- The user wants the file to remain but wants to erase its contents.
- File attributes remain unchanged (except for file length, which is reset to 0), but space is released.

# The Open File Table

Most OS require that a file be explicitly 'opened' before operations (except create/delete) can occur.
The OS keeps a small table, the 'open-file table', containing information about all open files.
When a file operation is requested, the file is specified via an index into this table.
Avoids constant directory searching.

# Open File Information

## Information associated with an open file includes

## File pointer

Tracks the last read/write location (per process).

## File-open count

Number of times a file is open (to allow safe removal from the open-file table).

## Disk location of the file

Cached to avoid disk reads for directory lookups.

## Access rights

Checked on open and cached for subsequent operations.

# Summary

File types allow the OS and applications to handle data correctly.
Basic file operations include create, write, read, reposition, delete, and truncate.
The OS uses an open-file table to cache file information and optimize repetitive operations.
Understanding these operations is key to understanding how applications interact with storage.

# File Access Methods

Course: Operating Systems

## Unit 4

File Management

## Lecture 22

File Access Methods

# Introduction to Access Methods

Files store information. When it is used, this information must be accessed and read into computer memory.
Different applications access files in different ways.
The file system provides different access methods to accommodate these varying needs.
The two primary access methods are Sequential Access and Direct Access.

# Sequential Access

The simplest and most common access method.
Information in the file is processed in order, one record after the other.

**Example**

Compilers usually read source code files sequentially.

**Operations**

read_next(), write_next(), reset(), skip_forward(n).

# Sequential Access Mechanism

The OS maintains a read/write pointer.
A read_next() operation reads the next portion of the file and automatically advances the pointer.
A write_next() operation appends to the end of the file (or overwrites at the current pointer) and advances.
Best suited for magnetic tapes or batch processing.

# Direct Access (Relative Access)

A file is made up of fixed-length logical records that allow programs to read and write records rapidly in no particular order.
The direct access method is based on a disk model of a file, since disks allow random access to any block.

**Operations**

read(n), write(n) where n is the relative block number.

# Direct Access Mechanism

The file is viewed as a numbered sequence of blocks or records.
We can read block 14, then read block 53, then write block 7.
Relative block numbers are an index relative to the beginning of the file.
The OS handles mapping these logical relative block numbers to absolute physical disk addresses.

# Simulating Access Methods

## Direct access can simulate sequential access

- Keep a variable 'cp' (current position).
- To read_next(): read(cp); cp = cp + 1;

Sequential access cannot easily simulate direct access (very slow on tape media).
Modern OSes often merge both into a unified I/O interface using seek() followed by read().

# Other Access Methods

Other access methods can be built on top of a direct-access method.
These generally involve the creation of an index for the file.
Keep the index in memory. Fast determination of exact record location.

**Types**

Indexed Sequential Access Method (ISAM), Hashed Files.

# Indexed Access Method

Instead of specifying a block number, the application specifies a key. The file system maintains an index, similar to an index in the back of a book.

**To find a record**

- Search the index to find the pointer.
- Use the pointer to directly access the record.

# Multi-Level Indexing

If the index itself becomes too large to fit in memory, a multi-level index is used.
Create an index for the index file.
Example: IBM's ISAM (Indexed Sequential Access Method) uses a master index pointing to disk tracks, which contain secondary indices pointing to records.

# Summary

Access methods define how applications read and write data in files.
Sequential access reads data in strict order (good for text, streams).
Direct access reads data at any specified location (good for databases).
Indexed methods use keys and pointers to quickly locate specific records.
Modern OSes typically provide a direct access model that can support all these methods.

# File Allocation Methods (Part 1)

Course: Operating Systems

## Unit 4

File Management

## Lecture 23

File Allocation Methods (Part 1)

# Introduction to File Allocation

A direct-access disk can be viewed as a large array of disk blocks.

**The OS must allocate these blocks to files in a way that**

- Disk space is utilized effectively.
- Files can be accessed quickly.

**There are three major methods of allocating disk space**

Contiguous, Linked, and Indexed.

# Contiguous Allocation

Requires that each file occupy a set of contiguous blocks on the disk.
Disk addresses define a linear ordering on the disk.
The directory entry for a file indicates the address of the starting block and the length of the area allocated for this file.

# Contiguous Allocation: Advantages

**Excellent Read Performance**

Successive blocks are physically adjacent, meaning minimal disk head movement (seek time).

Supports both Sequential and Direct Access efficiently.

- To access block i of a file starting at b, simply read disk block b + i.

# Contiguous Allocation: Disadvantages

## External Fragmentation

Free disk space is broken into little pieces.

## Finding Space for a New File

OS must use First Fit, Best Fit, etc., which is complex and slow.

## File Growth

It's hard to expand a file if the adjacent space is already in use. May require copying the entire file to a new, larger space.

# Extent-Based Systems

A modification of contiguous allocation.
File is allocated in a set of contiguous chunks called 'extents'.
A file consists of one or more extents.
Directory entry contains the starting address and length of the first extent, plus a pointer to the next extent.

# Linked Allocation

Solves all problems of contiguous allocation.
Each file is a linked list of disk blocks.
The disk blocks may be scattered anywhere on the disk.
The directory contains a pointer to the first and last blocks of the file.

# Linked Allocation: Mechanism

**To create a file**

Just create a directory entry.

**To write**

Find any free block, write to it, and link it to the end of the file.

**To read**

Start at the first block and follow the pointers.

# Linked Allocation: Advantages

No external fragmentation.
File size does not need to be declared in advance.
Files can grow dynamically without needing to copy existing data.
Any free block can be used.

# Linked Allocation: Disadvantages

Supports Sequential Access only effectively. Direct access is extremely slow (must follow pointers from the beginning).

## Space Overhead

Pointers take up space in every block.

## Reliability

If a pointer is lost or damaged due to a disk error, the rest of the file is lost.

# File-Allocation Table (FAT)

An important variation of linked allocation, used in MS-DOS and OS/2.
A section of disk at the beginning of each partition is set aside to contain the table (FAT).
The FAT has one entry for each disk block and is indexed by block number.
The directory entry contains the block number of the first block; the FAT entry contains the block number of the next block.

# Summary

## Contiguous Allocation

Fast access, suffers from fragmentation and hard-to-predict file sizes.

## Linked Allocation

No fragmentation, easy growth, but terrible direct access and pointer overhead.

## FAT

Improves linked allocation by centralizing pointers.

Next time, we will discuss Indexed Allocation, which solves many of linked allocation's problems.

# Indexed Allocation & File Sharing

Course: Operating Systems

## Unit 4

File Management

## Lecture 24

Indexed Allocation and File Sharing

# Indexed Allocation

Linked allocation solves external fragmentation but hurts direct access.

**Indexed allocation solves this by bringing all the pointers together into one location**

the index block.

Each file has its own index block, which is an array of disk block addresses. The ith entry in the index block points to the ith block of the file.

# Indexed Allocation: Characteristics

## Supports Direct Access

Just look up the pointer in the index block.

## No External Fragmentation

Any free block on the disk can be allocated to a file.

## Pointer Overhead

The index block takes up a whole disk block, even for very small files.

If a file is too large, a single index block may not be able to hold all the pointers.

# Combined Scheme (UNIX Inodes)

UNIX uses a combined indexed allocation scheme.
The index block (inode) contains say, 15 pointers.

- 12 Direct pointers: Point directly to data blocks. Great for small files.
- 1 Single Indirect pointer: Points to a block of pointers.
- 1 Double Indirect pointer: Points to a block of single indirect pointers.
- 1 Triple Indirect pointer: Points to a block of double indirect pointers.

# Introduction to File Sharing

In multi-user systems, sharing files is highly desirable.
File sharing must be carefully controlled to ensure data integrity and security.
The OS must arbitrate access based on permissions and access rights.
Issues include concurrent access, locking, and consistency.

# Multiple Users and Permissions

Systems define concepts of Owner (creator), Group (set of users), and Universe (everyone else).
Permissions are typically defined for each of these categories.
The OS checks the user ID of the requesting process against the file's access control lists or permission bits.
Prevents unauthorized access.

# Semantics of File Sharing

When multiple users share a file, what happens if they write to it simultaneously?

## UNIX Semantics

Writes are immediately visible to all other users. File has a single image.

## Session Semantics (Andrew File System)

Writes are visible only when the file is closed.

## Immutable Shared Files Semantics

Once created, a file cannot be modified. Sharing is implicitly safe.

# File Locking

To manage concurrent access, systems provide file locking.

## Shared lock (Read lock)

Multiple processes can hold it simultaneously.

## Exclusive lock (Write lock)

Only one process can hold it at a time.

## Advisory locking

Processes must explicitly check for locks (UNIX default).

## Mandatory locking

OS prevents access if a lock is held (Windows default).

# Remote File Systems

Networking allows sharing files across machines.

## Client-Server Model

A server machine stores files, client machines access them over a network.

## Examples

NFS (Network File System) for UNIX, CIFS/SMB for Windows.

The OS provides an interface making remote files appear local.

# Stateful vs. Stateless File Service

**Stateful (e.g., CIFS)**

Server keeps track of open files, current position pointers, etc.

- Better performance, handles locks well. If server crashes, state is lost.

**Stateless (e.g., NFS v3)**

Each request contains all information needed (file handle, offset, etc.). Server remembers nothing between requests.

- Very robust to server crashes, but requires more network overhead.

# Summary

Indexed allocation provides fast direct access without fragmentation by using index blocks.
UNIX inodes use a hybrid indexed approach.
File sharing requires managing permissions, locking, and consistency semantics.
Remote file systems extend sharing across networks using stateful or stateless models.

# File Protection and Unit Review

Course: Operating Systems

## Unit 4

File Management

## Lecture 25

File Protection and Unit Review

# Need for File Protection

Information kept in a computer system must be kept safe from physical damage (reliability) and improper access (protection).
Protection ensures that only authorized users can access files in authorized ways.
Especially critical in multi-user and networked environments.
Protection mechanisms are enforced by the Operating System.

# Types of Access

Protection mechanisms provide controlled access by limiting the types of operations that can be performed.

## Common operations to be controlled

- Read: Read from the file.
- Write: Write or append to the file.
- Execute: Load the file into memory and execute it.
- Append: Write only at the end of the file.
- Delete: Delete the file.
- List: List the name and attributes of the file (directory access).

# Access Control Lists (ACLs)

The most common approach to the protection problem is to make access dependent on the identity of the user.
An Access Control List (ACL) is associated with each file and directory.
It specifies user names and the types of access allowed for each user.
When a user requests access, the OS checks the ACL. If authorized, access is granted; otherwise, it is denied.

# UNIX Permission Bits

**To simplify ACLs, systems like UNIX condense users into three classifications**

- Owner: The user who created the file.
- Group: A set of users who are sharing the file.
- Universe/Others: All other users in the system.

**Only 3 fields are needed**

Owner, Group, Others.

# UNIX rwx Concept

**For each category (Owner, Group, Others), 3 bits define access**

r (Read) = 4
w (Write) = 2
x (Execute) = 1

**Example**

chmod 764 file.txt

- Owner: 7 (4+2+1 = rwx)
- Group: 6 (4+2 = rw-)
- Others: 4 (r–)

# Other Protection Approaches

## Passwords

A password can be attached to each file or to a directory.

- Pros: Simple to implement.
- Cons: Users must remember many passwords, easily forgotten or leaked.

## Capability Lists

The user holds a 'ticket' or capability that grants access to an object. (Reverse of ACLs).

## Encryption

Data is mathematically scrambled so even if access is gained, data is unreadable without the key.

# Unit 4 Review: File Concept & Types

## File

Logical abstraction of storage.

## Attributes

Metadata (name, size, location, protection) managed by the OS.

## File Types

Identified by extensions or magic numbers to guide OS/application behavior.

## Operations

Create, Write, Read, Reposition, Delete, Truncate.

# Unit 4 Review: Access & Allocation

## Access Methods

- Sequential: Read in order (tape-like).
- Direct: Access any block instantly (disk-like).

## Allocation Methods

- Contiguous: Fast but fragments.
- Linked: No fragmentation but slow direct access (FAT improves this).
- Indexed: Fast direct access, no external fragmentation (UNIX inodes).

# Unit 4 Review: Sharing & Protection

## Sharing

- Requires locking and concurrency control semantics.
- Remote file systems extend this over networks (Client/Server).

## Protection

- Defending against unauthorized access.
- Controlled via Access Control Lists (ACLs) and UNIX rwx permission bits.

# End of Unit 4

This concludes Unit 4 on File Management.

## You should now understand

- What files and directories are.
- How data is accessed and stored on disks.
- How operating systems implement sharing and protection.

## Next Unit

We will move to I/O Systems and Device Management.

# Linux Installation and Process Commands

Course: Operating Systems

## Unit 5

Linux Commands and Shell Programming

## Lecture 26

Installation and Process Commands

# Learning Objectives

Understand the prerequisites and methods for installing Linux.
Learn the steps involved in a typical Linux OS installation.
Understand what a process is in the context of Linux.
Learn commands to view, manage, and terminate Linux processes.

# Introduction to Linux OS

Linux is an open-source Unix-like operating system based on the Linux kernel.
First released by Linus Torvalds on September 17, 1991.
Typically packaged in a Linux distribution (e.g., Ubuntu, Fedora, Debian).
Known for its stability, security, and flexibility.

# Linux Installation Pre-requisites

## Hardware Requirements

Minimum RAM, storage, and CPU depend on the specific distribution (e.g., Ubuntu requires 4GB RAM, 25GB storage).

## Installation Media

A bootable USB drive or DVD containing the Linux ISO image.

## Backup Data

Always backup important data before installing a new OS, especially in a dual-boot setup.

## Secure Boot

May need to be disabled in BIOS/UEFI for some distributions.

# Linux Installation Methods

## Standalone Installation

Erasing the entire disk and installing Linux as the only OS.

## Dual Boot

Installing Linux alongside an existing OS (like Windows), allowing selection at boot time.

## Virtual Machine (VM)

Running Linux inside a host OS using software like VirtualBox or VMware.

## Windows Subsystem for Linux (WSL)

Running a Linux environment directly on Windows without a traditional VM.

# Step-by-step Installation (Ubuntu)

- Boot from USB: Insert bootable media and select it in BIOS/UEFI.
- Choose Language and Keyboard Layout.
- Network Connection: Connect to Wi-Fi for updates during installation.
- Installation Type: Normal or Minimal, and download updates/third-party software.
- Disk Partitioning: 'Erase disk' or 'Something else' (manual partitioning).
- Create User Account: Set username, computer name, and password.

# Introduction to Linux Processes

A process is an executing instance of a program.
Every process has a unique Process ID (PID).
The first process started by the kernel is init (or systemd) with PID 1.
Processes can spawn child processes.

## Foreground Processes

Run on the terminal and block other commands.

## Background Processes

Run silently, allowing the terminal to be used.

# Process States

## Running (R)

The process is either executing or ready to execute.

## Sleeping (S/D)

Waiting for an event or resource (Interruptible or Uninterruptible).

## Stopped (T)

The process has been suspended (e.g., via Ctrl+Z).

## Zombie (Z)

The process has terminated but its parent hasn't read its exit status.

# The 'ps' Command

## ps (Process Status)

Displays a snapshot of currently running processes.

## Basic usage

ps (shows processes for the current shell).

## Common options

- ps -e or ps -A: Select all processes.
- ps -f: Full-format listing (shows UID, PID, PPID, CMD, etc.).
- ps aux: Widely used BSD-style syntax to show all processes with detailed info.

# The 'top' Command

## top

Provides a dynamic, real-time view of running processes.

## Displays system summary

Uptime, load average, CPU usage, memory usage.

## Interactive commands in top

- k: Kill a process (prompts for PID).
- q: Quit top.
- P: Sort by CPU usage.
- M: Sort by Memory usage.

# The 'kill' Command

## kill

Sends a signal to a process, usually to terminate it.

## Syntax

kill [signal] PID

## Common signals

- SIGTERM (15): Default signal, requests graceful termination.
- SIGKILL (9): Forces immediate termination (cannot be ignored).

## Example

kill -9 1234 kills process with PID 1234.

## killall

Kills processes by name (e.g., killall firefox).

# Process Priorities: 'nice' and 'renice'

Linux uses a priority system based on 'niceness' values.
Niceness ranges from -20 (highest priority) to 19 (lowest priority). Default is 0.

**nice**

Starts a new process with a specific niceness.

- Example: nice -n 10 ./script.sh

**renice**

Alters the scheduling priority of an already running process.

- Example: renice +5 1234

Only the root user can decrease niceness (increase priority).

# Date, Time and Directory Commands

Course: Operating Systems

## Unit 5

Linux Commands and Shell Programming

## Lecture 27

Calendar, Date, Sleep, and Directory Commands

# Learning Objectives

Learn how to display calendars and system dates in Linux.
Understand the usage of the sleep command.
Familiarize with the Linux File System Hierarchy.

**Master basic directory commands**

pwd, cd, mkdir, rmdir, and ls.

# The 'cal' Command

## cal

Displays a simple, formatted calendar in the terminal.

## Basic usage

cal (displays the current month).

## Common options

- cal 2024: Displays the calendar for the entire year 2024.
- cal 8 2024: Displays the calendar for August 2024.
- cal -y: Displays the calendar for the current year.
- cal -3: Displays previous, current, and next month.

# The 'date' Command

## date

Prints or sets the system date and time.

## Basic usage

date (prints current date and time).

## Formatting output using '+'

- date +%Y-%m-%d: Outputs YYYY-MM-DD (e.g., 2024-08-15).
- date +%H:%M:%S: Outputs HH:MM:SS (e.g., 14:30:00).
- date '+%A %d %B, %Y': Outputs formatted string like 'Thursday 15 August, 2024'.

# The 'sleep' Command

**sleep**

Delays or pauses the execution of a script or command chain for a specified amount of time.

**Syntax**

sleep NUMBER[SUFFIX]

**Suffixes**

- s: seconds (default)
- m: minutes
- h: hours
- d: days

**Example**

sleep 5 (pauses for 5 seconds).

**Example**

sleep 2m (pauses for 2 minutes).

# Linux File System Hierarchy

Linux uses a single, hierarchical directory structure starting from the root directory ('/').

## Important directories

- /: Root directory.
- /home: User home directories (e.g., /home/user).
- /etc: System configuration files.
- /bin and /sbin: Essential system binaries (commands).
- /var: Variable data like logs and databases.

# Directory Command: 'pwd'

## pwd (Print Working Directory)

Outputs the absolute path of the current directory you are in.

## Syntax

```
pwd
```

## Absolute vs Relative Paths

- Absolute: Full path from the root (e.g., /home/user/Documents).
- Relative: Path relative to the current directory (e.g., ./Documents or ../Downloads).

# Directory Command: 'cd'

## cd (Change Directory)

Used to navigate between directories.

## Basic usage

cd /path/to/directory

## Special shortcuts

- cd ~ or cd : Go to your home directory.
- cd .. : Move up one directory level (parent directory).
- cd - : Return to the previous working directory.
- cd / : Go to the root directory.

# Directory Commands: 'mkdir' and 'rmdir'

## mkdir (Make Directory)

Creates new directories.

- Example: mkdir folder1
- Use -p to create parent directories as needed (e.g., mkdir -p parent/child/grandchild).

## rmdir (Remove Directory)

Removes completely empty directories.

- Example: rmdir folder1
- Note: rmdir will fail if the directory contains any files or subdirectories.

# Directory Command: 'ls'

## ls (List)

Lists directory contents (files and folders).

## Basic usage

ls (lists contents of current directory).

## Common options

- ls -l: Long format (shows permissions, owner, size, modification date).
- ls -a: Show hidden files (files starting with a dot, e.g., .bashrc).
- ls -h: Human-readable sizes (e.g., 1K, 234M, 2G).
- ls -t: Sort by modification time.

# Summary of Directory Navigation

Use pwd to check your current location.
Use ls to see what's in the current directory.
Use cd to move around.
Use mkdir to create new folders to organize your work.
These four commands form the essential workflow for command-line file system navigation.

# Linux File and User Commands

Course: Operating Systems

## Unit 5

Linux Commands and Shell Programming

## Lecture 28

Linux File Commands and User Commands

# Learning Objectives

Learn commands to create, copy, move, and remove files.
Understand how to view file contents in the terminal.
Explore tools for searching files and text patterns.
Learn how to manage file permissions and ownership.
Understand commands for user and group administration.

# File Manipulation Commands

### touch

Updates file timestamps. If the file doesn't exist, creates an empty file (e.g., touch file.txt).

### cp (Copy)

Copies files or directories.

- cp source.txt dest.txt
- cp -r dir1 dir2 (recursive copy for directories).

### mv (Move)

Moves or renames files or directories.

- mv oldname.txt newname.txt (rename).

### rm (Remove)

Deletes files or directories.

- rm file.txt
- rm -rf dir1 (forcefully and recursively remove a directory).

# Viewing File Content

## cat (Concatenate)

Displays the entire content of a file on the screen.

## more

Displays file content one screen at a time. Press Space to advance.

## less

Advanced version of 'more' allowing backward and forward navigation.

## head

Displays the first 10 lines of a file (e.g., head -n 5 file.txt).

## tail

Displays the last 10 lines of a file (e.g., tail -f log.txt to follow live updates).

# Searching and Counting

## grep (Global Regular Expression Print)

Searches for a specific text pattern inside files.

- grep 'error' syslog.txt
- grep -i 'warning' (case-insensitive search).

## find

Searches for files in a directory hierarchy based on criteria.

- find / -name 'file.txt' (search by name).
- find . -size +10M (search for files ¿ 10MB).

## wc (Word Count)

Counts lines, words, and characters in a file.

- wc -l file.txt (counts lines).

# File Permissions ('chmod')

Linux file permissions consist of Read (r=4), Write (w=2), and Execute (x=1).

### Permissions apply to

Owner (u), Group (g), and Others (o).

### chmod

Changes file permissions.

### Symbolic mode

- chmod u+x script.sh (adds execute permission to owner).

### Numeric (Octal) mode

- chmod 755 script.sh (Owner: rwx(7), Group: r-x(5), Others: r-x(5)).
- chmod 644 file.txt (Owner: rw-(6), Group: r–(4), Others: r–(4)).

# File Ownership ('chown')

Every file and directory in Linux has an owner user and a group owner.

### chown (Change Owner)

Changes the user and/or group ownership of a file.

### Syntax

chown [owner]:[group] file

### Examples

- chown alice file.txt (Changes owner to alice).
- chown alice:developers file.txt (Changes owner to alice, group to developers).

### Note

You typically need superuser (root) privileges to use chown.

# Introduction to User Management

Linux is a multi-user operating system.

### Root User

The superuser with absolute privileges over the system (UID 0).

### Regular Users

Standard users with restricted privileges.

### System Users

Accounts used by services and applications, not meant for login.

### su (Substitute User)

Switches the current user session to another user.

### sudo (Superuser Do)

Executes a single command with root privileges.

# Adding and Modifying Users

**useradd**

Creates a new user account.

- useradd -m bob (Creates user bob and his home directory).

**passwd**

Sets or changes a user's password.

- passwd bob (Requires root privileges to change another user's password).

**usermod**

Modifies user account properties.

- usermod -aG sudo bob (Adds bob to the 'sudo' group).

# Deleting Users and User Information

## userdel

Deletes a user account.

- userdel bob
- userdel -r bob (Removes user bob along with his home directory).

## Information Commands

- who: Shows who is currently logged into the system.
- whoami: Prints the current effective username.
- id: Prints user and group IDs (UID/GID) for the current user.

# Summary

Files can be managed using cp, mv, rm, and viewed with cat, less, etc.
Searching is done efficiently using grep and find.
File security is maintained through permissions (chmod) and ownership (chown).
User administration involves commands like useradd, usermod, and passwd.
Privilege escalation is handled via su and sudo.

# Networking Commands and Basic Shell Scripts

Course: Operating Systems

## Unit 5

Linux Commands and Shell Programming

## Lecture 29

Networking Commands and Introduction to Shell Scripts

# Learning Objectives

Learn commands to test network connectivity and configuration.
Understand how to connect to remote Linux machines.
Understand what a shell script is and how to execute one.

**Learn basic operators in shell programming**

arithmetic, relational, boolean, and string.

# Network Interface Configuration

**ifconfig**

Historical command to view or configure network interfaces (part of net-tools).

- Displays IP address, MAC address, and network statistics.

**ip**

The modern, more powerful replacement for ifconfig (part of iproute2).

- ip addr show: Displays IP addresses.
- ip link show: Displays network interfaces.
- ip route: Displays the routing table.

# Network Connectivity Testing

## ping

Tests reachability of a host on an IP network using ICMP Echo requests.

- ping google.com (Press Ctrl+C to stop).
- ping -c 4 8.8.8.8 (Sends exactly 4 packets).

## traceroute (or tracepath)

Prints the route (path) packets trace to network host.

- Useful for finding where a network connection is failing or lagging.

# Network Statistics and Remote Access

**netstat**

Prints network connections, routing tables, and interface statistics.

- netstat -tuln (Shows listening TCP/UDP ports).

**ssh (Secure Shell)**

Securely log into remote machines over a network.

- Syntax: ssh username@hostname_or_IP

**scp (Secure Copy)**

Securely transfer files between local and remote hosts.

- scp file.txt user@remote:/path/to/destination

# Introduction to Shell Scripts

A Shell Script is a text file containing a sequence of commands for a UNIX-based operating system.
Used for automating repetitive tasks, system administration, and application execution.
The Shebang (#!) on the first line specifies the interpreter (e.g., #!/bin/bash).

## Execution steps

- Create file: touch myscript.sh
- Write code and save.
- Make executable: chmod +x myscript.sh
- Run: ./myscript.sh

# Variables and Basic I/O

Variables store data. No data type declaration is needed.

**Assignment**

NAME="Alice" (No spaces around the '=' sign).

**Access**

Use '$' before the variable name (e.g., echo $NAME).

**Output**

echo command prints text to the terminal.

- echo "Hello World"

**Input**

read command takes input from the user.

- read -p 'Enter your name: ' USER_INPUT

# Arithmetic Operators

Bash requires tools like 'expr' or double parentheses '(( ))' for arithmetic.

**Operators**

\+ (Addition), - (Subtraction), * (Multiplication), / (Division), % (Modulus).

**Example using expr**

- sum=‘expr 5 + 3‘ (Spaces required around operators)

**Example using $(( ))**

- sum=$(( 5 + 3 ))

Bash arithmetic is strictly integer-based. For floating-point, use 'bc' (Basic Calculator).

# Relational Operators

Used to compare numeric values.

**-eq**

Equal to

**-ne**

Not equal to

**-gt**

Greater than

**-lt**

Less than

**-ge**

Greater than or equal to

**-le**

Less than or equal to

**Example syntax**

```
if [ $a -eq $b ]
```

# String and Boolean Operators

## String Operators

- = : Strings are equal
- != : Strings are not equal
- -z : String has zero length (empty)
- -n : String is not empty

## Boolean (Logical) Operators

- ! : Logical NOT
- -o : Logical OR
- -a : Logical AND
- Modern bash also supports && (AND) and —— (OR) with double brackets [[ ]].

# File Test Operators

Used to test various properties associated with a file.

**-d file**

Checks if file is a directory.

**-f file**

Checks if file is an ordinary file.

**-r file**

Checks if file has read permission.

**-w file**

Checks if file has write permission.

**-x file**

Checks if file has execute permission.

**-s file**

Checks if file size is greater than 0.

# Summary

Networking commands (ping, ip, netstat, ssh) are vital for system connectivity and administration.
Shell scripts automate commands and logic.
Scripts must be made executable (chmod +x) and start with a shebang.
Variables store data, while operators allow for arithmetic and conditional comparisons of numbers, strings, and files.

# Shell Scripts: Control and Loop Statements

Course: Operating Systems

## Unit 5

Linux Commands and Shell Programming

## Lecture 30

Control Structures and Loop Statements in Shell Scripting

# Learning Objectives

Understand the syntax and usage of 'if', 'if-else', and 'elif' statements. Learn how to use 'case' statements for multi-way branching.

**Master loops**

'for', 'while', and 'until'.

**Understand loop control mechanisms**

'break' and 'continue'.

Learn how to pass and handle command-line arguments in scripts.

# The 'if' Statement

Executes a block of code only if a specified condition evaluates to true.

## Syntax

```
if [ condition ]
then
command1
command2
fi
```

## Spaces are crucial

inside the brackets [ ], there must be spaces around the condition.

# if-else and if-elif-else

if-else provides an alternative execution path when the condition is false.
if [ condition ]; then
# true block
else
# false block
fi
if-elif-else allows checking multiple conditions sequentially.
if [ cond1 ]; then ... elif [ cond2 ]; then ... else ... fi

# The 'case' Statement

An elegant alternative to deeply nested if-elif-else statements.
Matches a variable against multiple patterns.

## Syntax

```
case $variable in
pattern1) command ;;
pattern2) command ;;
*) default_command ;; # Catch-all pattern
esac
```

# Introduction to Loops

Loops execute a block of code multiple times.

## Three main types of loops in Bash

- for loop: Iterates over a list of items.
- while loop: Repeats as long as a condition is TRUE.
- until loop: Repeats as long as a condition is FALSE (until it becomes true).

# The 'for' Loop

Iterates over a specific list of values.

## Syntax

```
for var in item1 item2 item3
do
echo $var
done
```

## Example (iterate over files)

```
for file in *.txt; do cat $file; done
```

## C-style for loop

for (( i=0; i¡5; i++ )); do ... done

# The 'while' Loop

Executes a block of commands repeatedly as long as the condition evaluates to true.

## Syntax

```
while [ condition ]
do
commands
done
```

## Example

```
count=1
while [ $count -le 5 ]; do echo $count; ((count++)); done
```

# The 'until' Loop

Executes commands repeatedly until the condition evaluates to true (i.e., while it is false).

## Syntax

```
until [ condition ]
do
commands
done
```

## Example

```
count=1
until [ $count -gt 5 ]; do echo $count; ((count++)); done
```

# Loop Control: 'break' and 'continue'

**break**

Terminates the loop entirely and passes execution to the command following the loop.

- Useful for exiting a loop early when a specific condition is met.

**continue**

Skips the rest of the current iteration and jumps to the next iteration of the loop.

- Useful for skipping specific items in a list without stopping the whole loop.

# Command Line Arguments

Scripts can accept arguments passed from the command line (e.g., ./script.sh arg1 arg2).

## Special variables

- $0: The name of the script itself.
- $1, $2... $9: The first, second, etc. arguments.
- $#: The total number of arguments passed.
- $* or $@: All arguments passed as a single string or array.
- $?: The exit status of the last executed command (0 means success).

# Summary of Shell Scripting

Control structures like if-else and case add logic and decision-making.
Loops (for, while, until) allow for automation of repetitive tasks.
Loop execution can be modified using break and continue.
Command line arguments ($1, $#, etc.) make scripts dynamic and adaptable.