Operating System: By Anurag Mahapatra

Module I

1. Introduction to Operating Systems

1.1 What is an Operating System (OS)?

An **Operating System (OS)** is a system software that acts as an intermediary or interface between a user of a computer and the computer hardware. It is a program that manages the computer's hardware, provides a basis for application programs, and acts as an environment in which a user can execute programs in a convenient and efficient manner.

Fundamental Views of an OS:

- **Resource Manager:** The OS manages and allocates all system resources, including the CPU, memory, I/O devices, and files. It handles conflicting requests for resources, ensuring efficient and fair use. This is a bottom-up view.
- **Control Program:** The OS controls the execution of user programs and the operations of I/O devices to prevent errors and improper use of the computer. It looks over and protects the computer system. This is a black-box view.
- **Virtual Machine:** The OS hides the complexities of the hardware from the user and application programs. It provides a high-level, abstract interface that is easier to use than the raw hardware, creating an "extended machine."

Primary Goals of an Operating System:

1. **Convenience:** To make the computer system easier and more convenient for the user to interact with.
2. **Efficiency:** To manage computer system resources in an efficient manner, maximizing throughput, minimizing response time, and ensuring fairness.
3. **Ability to Evolve:** To permit the development, testing, and introduction of new system functions without interfering with existing services.

Computer System Structure: A computer system can be divided into four main components:

1. **Hardware:** Provides basic computing resources (CPU, memory, I/O devices).
2. **Operating System:** Controls and coordinates the use of hardware among various applications and users.
3. **Application Programs:** Define the ways in which system resources are used to solve the computing problems of the users (e.g., word processors, compilers, web browsers).
4. **Users:** People, machines, or other computers.

1.2 Evolution and Types of Operating Systems

1. The First Generation (1940s - early 1950s): No Operating System

- Computers were large, expensive machines run from a console. Programming was done in absolute machine language, often by wiring plug-boards. There were no operating systems.

www.iitkirba.xyz

2. The Second Generation (1955-1965): Simple Batch Systems

- **Concept:** To improve efficiency, jobs with similar needs were batched together and run as a group. Users submitted jobs (on punch cards) to a computer operator, who would batch and run them.
- **Function:** The OS's primary function was to automatically transfer control from one job to the next.
- **Characteristics:** No direct user interaction; the CPU was often idle due to slow mechanical I/O.

3. The Third Generation (1965-1980): Multiprogramming and Time-Sharing

- **Multiprogramming Systems:**
  - **Concept:** Multiple jobs are kept in main memory simultaneously. When one job has to wait for an I/O operation, the OS switches the CPU to another job, increasing CPU utilization.
  - **Mechanism:** Requires memory management to hold multiple jobs and CPU scheduling to choose the next job to run.
- **Time-Sharing (Multitasking) Systems:**
  - **Concept:** A logical extension of multiprogramming where the CPU switches between jobs so frequently that users can interact with each program while it is running.
  - **Mechanism:** Each user is given a small time slice or **quantum**. If a process is not finished within its quantum, it is preempted.
  - **Objective:** To provide a responsive, interactive experience for multiple users.

4. The Fourth Generation (1980-Present Day): Modern Systems

- **Personal Computer (PC) Systems:**
  - Systems dedicated to a single user, emphasizing convenience, responsiveness, and a graphical user interface (GUI).
  - Examples: MS-DOS, Windows, macOS, Linux (desktop versions).
- **Parallel Systems (Multiprocessor Systems):**
  - **Concept:** Systems with more than one CPU in close communication, sharing system resources. Also known as "tightly coupled systems."
  - **Advantages:** Increased throughput, economy of scale, and enhanced reliability (graceful degradation).
  - **Types:**
    - **Symmetric Multiprocessing (SMP):** Each processor runs an identical copy of the OS. All processors are peers.
    - **Asymmetric Multiprocessing (ASMP):** A master processor controls the system, and slave processors perform assigned tasks.
- **Distributed Systems:**
  - **Concept:** A collection of autonomous, physically separate computer systems networked to provide users with access to shared resources. Also known as "loosely coupled systems."
  - **Characteristics:** Each processor has its own local memory; communication occurs over a network.
  - **Advantages:** Resource sharing, computation speedup, reliability.

www.iitkirba.xyz

- **Real-Time Operating Systems (RTOS):**
  - **Concept:** An OS used for applications with rigid time constraints. The correctness of a computation depends not only on the logical result but also on the time at which the results are produced.
  - **Types:**
    - **Hard Real-Time:** Guarantees that critical tasks complete on time. (e.g., industrial control, airbag systems).
    - **Soft Real-Time:** A critical task gets priority, but a late result is acceptable, though its value may be degraded. (e.g., multimedia streaming).

2. Operating System Structures

2.1 System Components & Services

An OS provides an environment for program execution through its services. Key services include:

1. **User Interface:**
   - **Command-Line Interface (CLI):** Users type commands directly (e.g., MS-DOS, Unix Shell).
   - **Graphical User Interface (GUI):** Users interact with visual elements like icons and menus (e.g., Windows, macOS).
2. **Program Execution:** Loading, executing, and managing the termination of programs.
3. **I/O Operations:** Managing communication with I/O devices on behalf of user programs.
4. **File-System Manipulation:** Creating, deleting, reading, writing, and managing permissions for files and directories.
5. **Communications:** Enabling inter-process communication (IPC) via shared memory or message passing.
6. **Error Detection:** Detecting and responding to errors in hardware, software, or user programs.
7. **Resource Allocation & Accounting:** Allocating resources to multiple users/jobs and keeping track of usage.
8. **Protection and Security:** Controlling access to resources and defending against internal/external threats.

2.2 System Calls

A **system call** is the programmatic interface to the services provided by the OS. It is how a user-level process requests a privileged action from the OS kernel.

- **Mechanism:** When a system call is executed, it causes a **trap**, switching the processor from **user mode** to **kernel mode**. The kernel examines the request, executes the necessary service, and then returns control to the user process, switching back to user mode. This dual-mode operation is essential for system protection.

- **Types of System Calls:**
  - **Process Control:** `fork()` (create process), `exit()` (terminate), `wait()` (wait for child process).
  - **File Management:** `open()`, `read()`, `write()`, `close()`.
  - **Device Management:** `ioctl()` (control device), `read()`, `write()`.
  - **Information Maintenance:** `getpid()` (get process ID), `alarm()`, `sleep()`.
  - **Communication:** `pipe()` (create communication channel), `socket()`.

www.iitkirba.xyz

2.3 OS Structure & Kernel Design

The **kernel** is the core of the operating system, the one program running at all times. OS design involves structuring the kernel to be efficient, reliable, and extensible.

- **Monolithic Structure:**
  - **Concept:** The entire operating system is a single large program running in kernel mode. All services (scheduling, file system, networking) are part of this one executable.
  - **Pros:** Excellent performance due to low overhead (function calls between components are fast).
  - **Cons:** Tightly coupled components make it complex, hard to debug, and difficult to maintain. A bug in one component can crash the entire system.
  - **Examples:** UNIX, Linux, MS-DOS.
- **Microkernel Structure:**
  - **Concept:** The kernel is stripped down to only the absolute essential functions, such as IPC, memory management, and CPU scheduling. Other services (file systems, device drivers) run as user-level server processes.
  - **Pros:** More reliable and secure (a failing server doesn't crash the kernel), easier to extend and port.
  - **Cons:** Poorer performance due to the overhead of message passing between user-level servers and the kernel.
  - **Examples:** Mach, QNX.
- **Hybrid Kernels:**
  - **Concept:** A compromise that combines the speed of monolithic kernels with the modularity of microkernels. Core services run in the kernel, but other components can be loaded dynamically.
  - **Examples:** Windows NT, macOS.
- **Exokernels:**
  - **Concept:** A research-oriented design that minimizes kernel abstractions. The kernel's only job is to securely multiplex the hardware, allocating physical resources to applications. Applications manage their own resources.
  - **Pros:** High performance and flexibility for application developers.
  - **Cons:** Greatly increases the complexity of application development.

3. Processes

3.1 Process Concept

A **process** is a program in execution. It is an active entity, unlike a program which is a passive file on a disk. A process includes:

- **Text Section:** The program code.
- **Data Section:** Global variables.
- **Heap:** Dynamically allocated memory.
- **Stack:** Function parameters, return addresses, local variables.

www.iitkirba.xyz

- **Program Counter & CPU Registers:** The current state of the process's execution.

3.2 Process States & PCB

As a process executes, it transitions between states:

1. **New:** The process is being created.
2. **Ready:** The process has all necessary resources except the CPU and is waiting to be executed.
3. **Running:** The process is currently being executed by the CPU.
4. **Waiting (Blocked):** The process is waiting for an event (e.g., I/O completion).
5. **Terminated:** The process has finished execution.

The OS maintains a **Process Control Block (PCB)** for each process. This data structure stores all information related to the process, including its state, program counter, CPU registers, scheduling information, memory management details, and I/O status.

3.3 Process Management and Context Switch

- **Process Creation:** A process can create new processes, forming a tree of processes. The creating process is the **parent**, and the new processes are **children**. In UNIX, this is done via the `fork()` system call.
- **Process Termination:** A process terminates when it finishes executing its final statement (`exit()`). A parent may also terminate its children (`abort()`).
- **Context Switch:** When the OS switches the CPU from one process to another, it performs a **context switch**. This involves saving the context (state in the PCB) of the old process and loading the context of the new process. This is overhead, as no user-level work is done.

3.4 Inter-Process Communication (IPC)

IPC mechanisms allow cooperating processes to communicate and synchronize.

1. **Shared Memory:** A region of memory is shared by processes. It's fast but requires explicit synchronization (e.g., using semaphores) to avoid race conditions.
2. **Message Passing:** Processes exchange messages via the kernel. It's safer and easier to manage but involves more overhead due to kernel intervention.

4. Threads

4.1 Introduction to Threads

A **thread** is a single sequential flow of execution within a process. A process can have multiple threads, which share the process's code, data, and resources but have their own program counter, register set, and stack. They are often called **lightweight processes**.

Benefits of Multithreading:

- **Responsiveness:** An application can remain responsive to user input while other threads perform long-running tasks.
- **Resource Sharing:** Threads share the memory and resources of their parent process, making communication efficient.

www.iitkirba.xyz

- **Economy:** It's cheaper and faster to create and switch between threads than processes.
- **Scalability:** Can utilize multiprocessor architectures by running threads in parallel.

4.2 Threading Models

The relationship between user threads (managed by a user-level library) and kernel threads (managed by the OS) is defined by a threading model:

1. **Many-to-One:** Many user threads map to a single kernel thread. Efficient but a blocking call by one thread blocks the entire process.
2. **One-to-One:** Each user thread maps to a dedicated kernel thread. Provides true parallelism but has higher overhead. (Used by Linux, Windows).
3. **Many-to-Many:** Multiplexes many user threads onto a smaller or equal number of kernel threads. A flexible compromise.

5. Processor Scheduling

5.1 Scheduling Concepts

- **CPU-I/O Burst Cycle:** Process execution consists of a cycle of CPU execution and I/O wait.
- **CPU Scheduler (Short-Term Scheduler):** Selects a process from the ready queue to allocate the CPU to.
- **Dispatcher:** The module that gives control of the CPU to the process selected by the scheduler. The time it takes is called **dispatch latency**.
- **Scheduling Types:**
  - **Non-Preemptive:** A process keeps the CPU until it voluntarily releases it.
  - **Preemptive:** The OS can force a process to release the CPU.

5.2 Scheduling Criteria

- **CPU Utilization:** Percentage of time the CPU is busy.
- **Throughput:** Number of processes completed per unit of time.
- **Turnaround Time:** Total time from submission to completion.
- **Waiting Time:** Total time a process spends in the ready queue.
- **Response Time:** Time from request submission to the first response.

5.3 Scheduling Algorithms

1. **First-Come, First-Served (FCFS):** Non-preemptive. Processes are served in the order they arrive. Simple but can have high average waiting time (convoy effect).
2. **Shortest-Job-First (SJF):** Can be preemptive or non-preemptive. Selects the process with the shortest next CPU burst. Optimal for average waiting time but suffers from potential starvation of long jobs. The preemptive version is **Shortest-Remaining-Time-First (SRTF)**.
3. **Priority Scheduling:** Can be preemptive or non-preemptive. Each process has a priority; the highest-priority process is selected. Can cause starvation, which can be mitigated with **aging**.
4. **Round Robin (RR):** Preemptive. Each process gets a fixed time quantum. Fair and suitable for time-sharing systems, but performance is sensitive to the quantum size.

www.iitkirba.xyz

5. **Multilevel Queue Scheduling (MLQ):** The ready queue is partitioned into several separate queues (e.g., foreground, background). Each queue has its own scheduling algorithm. Scheduling between queues is typically a fixed-priority preemptive scheme. Processes are permanently assigned to a queue.
6. **Multilevel Feedback Queue Scheduling (MLFQ):** Similar to MLQ, but allows processes to move between queues. If a process uses too much CPU time, it is moved to a lower-priority queue. If it waits too long in a lower-priority queue, it can be moved to a higher-priority queue. This is the most complex but also most general algorithm.

Module II

1. Process Synchronization

1.1 The Critical-Section Problem

A **critical section** is a code segment that accesses shared resources. The **critical-section problem** is to design a protocol to 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:

1. **Mutual Exclusion:** Only one process can be in its critical section at a time.
2. **Progress:** If no process is in a critical section, a process that wants to enter should not be blocked indefinitely.
3. **Bounded Waiting:** There's a limit on how many times other processes can enter their critical sections while a process is waiting.

1.2 Solutions to Mutual Exclusion

- **Hardware Solutions:**
  - **Test-and-Set:** An atomic instruction that tests a memory word and sets it.
  - **Compare-and-Swap:** An atomic instruction that compares a memory location to a value and, if they are equal, modifies it. These are used to build synchronization primitives but often lead to busy waiting.
- **Semaphores:** A semaphore is a synchronization variable `S` accessed only through two atomic operations: `wait()` and `signal()`.
  - `wait(S)` (or `P(S)`): Decrements the semaphore value. If the value becomes negative, the process blocks.
  - `signal(S)` (or `V(S)`): Increments the semaphore value. If the value is not positive, a waiting process is unblocked.

1.3 Classic Synchronization Case Studies

- **The Dining Philosophers Problem:**
  - **Scenario:** Five philosophers sit at a table with five chopsticks. Each needs two chopsticks to eat. This models processes competing for a limited set of resources.

www.iitkirba.xyz

- **Problem:** A naive solution where each philosopher picks up their left chopstick first can lead to **deadlock**—all philosophers holding one chopstick, waiting for another that will never be released.
  - **Semaphore-based Solution:** Represent each chopstick as a semaphore. A philosopher performs `wait()` on both chopstick semaphores to eat and `signal()` on both when done. To prevent deadlock, solutions include: 1. Allowing at most four philosophers at the table. 2. Allowing a philosopher to pick up chopsticks only if both are available (done in a critical section). 3. Using an asymmetric rule: odd-numbered philosophers pick up their left chopstick first, while even-numbered philosophers pick up their right first.

- **The Barbershop Problem:**

  - **Scenario:** A barbershop with one barber, one barber chair, and N waiting chairs. If no customers, the barber sleeps. If a customer arrives and the barber is busy, they wait if a chair is free, or leave if not. If the barber is asleep, the customer wakes him.
  - **Problem:** Synchronize the actions of the barber and customers to avoid race conditions (e.g., a customer trying to wake a barber who is already cutting hair).
  - **Semaphore-based Solution:** 1. `customers`: A semaphore to count waiting customers (used by the barber to sleep if 0). 2. `barber`: A semaphore to indicate if the barber is ready (used by customers to wait for the barber). 3. `mutex`: A semaphore for mutual exclusion when accessing the number of free waiting chairs. 4. An integer `waiting` to count customers in the waiting chairs.

2. Deadlocks

2.1 Deadlock Concepts

A **deadlock** is a state where two or more processes are indefinitely blocked, each waiting for a resource held by another process in the same set.

Necessary Conditions for Deadlock (Coffman Conditions):

1. **Mutual Exclusion:** Resources are non-sharable.
2. **Hold and Wait:** A process holds at least one resource while waiting for another.
3. **No Preemption:** A resource cannot be forcibly taken from a process.
4. **Circular Wait:** A circular chain of processes exists, where each process waits for a resource held by the next in the chain.

Resource-Allocation Graph (RAG): A directed graph used to visualize deadlocks.

- **Vertices:** Processes (circles) and Resource Types (rectangles).
- **Edges:**
  - **Request Edge:** `P -> R` (Process P requests resource R).
  - **Assignment Edge:** `R -> P` (Resource R is allocated to process P).
- If the graph contains a cycle, a deadlock *may* exist. If each resource type has only one instance, a cycle *guarantees* a deadlock.

2.2 Deadlock Prevention

www.iitkirba.xyz

Prevent deadlocks by ensuring at least one of the four necessary conditions never holds.

- **Eliminate Mutual Exclusion:** Not feasible for non-sharable resources.
- **Eliminate Hold and Wait:** Require processes to request all resources at once.
- **Allow Preemption:** Forcibly take resources from a waiting process.
- **Eliminate Circular Wait:** Impose a total ordering on resource types and require requests to follow that order.

2.3 Deadlock Avoidance

Use algorithms to ensure the system never enters an **unsafe state** (a state that could lead to deadlock).

- **Safe State:** A state is safe if there is at least one sequence of process executions that allows all processes to complete.
- **Banker's Algorithm:**
    - **Concept:** A deadlock avoidance algorithm. When a new process enters, it must declare the maximum number of instances of each resource type it may need.
    - **Mechanism:** When a process requests resources, the system simulates the allocation. It checks if the resulting state is safe. If it is, the resources are allocated; otherwise, the process must wait.
    - **Data Structures:**
        - `Available`: Vector of available resources.
        - `Max`: Matrix of maximum demand for each process.
        - `Allocation`: Matrix of currently allocated resources.
        - `Need`: Matrix of remaining resources needed (`Need = Max - Allocation`).
    - The **Safety Algorithm** checks if the system is in a safe state by finding an execution sequence.

2.4 Deadlock Detection and Recovery

Allow deadlocks to occur, then detect and resolve them.

- **Detection:**
    - If resources have single instances, check for cycles in the wait-for graph.
    - If resources have multiple instances, use a detection algorithm similar in structure to the Banker's Algorithm.
- **Recovery:**
    - **Process Termination:**
        - Abort all deadlocked processes.
        - Abort one process at a time until the deadlock cycle is broken.
    - **Resource Preemption:**
        - Select a victim process to preempt resources from.
        - Roll back the victim process to a safe state.
        - Ensure starvation doesn't occur (a process is always chosen as the victim).

www.iitkirba.xyz