Midterm I Notes

Chapter 1: OS Basics

4 Components of a computer system

- Hardware: PC parts and I/O.
- Operating System: Controls hardware among apps and users.
- Application Programs: Defines the ways in which the resources are used.
- Users: The controller (person, machine, computer).

OS goals

- execute programs
- make computer convenient to use
- use hardware efficiently

OS Definition

- Resource allocator
- Control Program

Interrupts

- The *interrupt vector* is a collection of pointers to the interrupt service routine.
- An interrupt MUST save the address of where it was before the interrupt began.
- New interrupts are *disabled* when another interrupt is being processed, otherwise it would be a *lost interrupt*.
- a **trap** Is a software generated interrupt caused by error or user request.
- An OS is interrupt driven.

Caching

- Frequently used info is copied to cache.
- Larger cache is slower but more useful so its important to find the balance.

Multi Programming

- Multi Programming organises jobs so the CPU can always be doing something because the user can't keep the CPU busy at all times.
- A subset of total jobs is kept in memory.
- 1 job is selected via job scheduling.

- When the job has to wait for something like I/O, OS switches to another job in the meantime.

Process Management Activities

The OS is responsible for these:

- Creating and deleting both user and system processes
- Suspending and resuming processes
- Providing mechanisms for process synchronisation
- Providing mechanisms for process communication
- Providing mechanisms for deadlock handling

Memory Management

The OS:

- Keeping track of which parts of memory are currently being used and by whom.
- Deciding which processes (or parts thereof) and data to move into and out of memory.
- Allocating and deallocating memory space as needed.

Storage Management

The OS:

- Creating and deleting files
- Creating and deleting directories to organise files
- Supporting primitives to manipulate files and directories
- Mapping files onto secondary storage
- Backup files onto stable (non-volatile) storage media

Mass Storage Management

The OS:

- manages free space\
- allocates storage
- schedules disk

Protection & Security

Protection: any mechanism for controlling access of processes or users to resources defined by the OS.
Security: defence of the system against internal and external attacks.

- Huge range, including denial-of-service, worms, viruses, identity theft, theft of service
  For distinguishing users:
- UserId and SecurityId are one per user and include name and number.

- UserId is associated with all files and processes that the user controls.
- GroupId allows a set of users to control files and processes. Used for privilege levels.
- **Privilege escalation** allows a user to change to an Id with more rights.

Chapter 2: Operating-System Structures

Operating System Services

- **User Interface (UI)**: OS provides interfaces for users to interact, which can be:
  - **Command-Line Interface (CLI)**: Text-based interface where users type commands.
  - **Graphical User Interface (GUI)**: User-friendly, often using mouse and icons.
  - **Touchscreen Interface**: Interaction via gestures and virtual keyboards.
  - GUI was invented at Xerox Palo Alto Research Center.
- **Program Execution**: OS loads programs into memory, runs them, and manages the program's normal or abnormal termination.
- **I/O Operations**: Allows programs to perform I/O, like reading from or writing to files and devices.
- **File System Manipulation**: Creating, deleting, reading, writing, and managing files and directories.
- **Communications**: Processes exchange information using:
  - **Shared memory** or **message passing**.
- **Error Detection**: Constant monitoring for hardware, I/O device, or user program errors, ensuring appropriate corrective action.

Resource Management & Security

- **Resource Allocation**: Distributes CPU cycles, memory, file storage, and I/O devices to multiple jobs and users.
- **Accounting**: Tracks user resource usage.
- **Protection & Security**:
  - **Protection**: Ensures controlled access to system resources.
  - **Security**: Requires user authentication and ensures system defense against external threats.

User Operating System Interfaces

- **CLI (Command-Line Interface)**: Executes commands directly input by the user, often implemented as shells (like Bourne Shell).
- **GUI (Graphical User Interface)**: Desktop metaphor interface using icons, typically accessed via mouse and keyboard.
- **Touchscreen Interfaces**: Gesture-based interaction for devices without a mouse or keyboard, often with voice commands.

System Calls

- **System Calls**: Programming interfaces provided by OS for program interaction, often accessed via APIs:
  - **Win32 API** (Windows)
  - **POSIX API** (UNIX, Linux, MacOS)
  - **Java API** (Java Virtual Machine)
- Examples of system calls:
  - **Process Control**: Create, terminate, load, execute processes, memory management, synchronization.
  - **File Management**: Create, delete, open, close, read, write files.
  - **Device Management**: Request, release, read/write devices.
  - **Information Maintenance**: Get/set system data, time, attributes.
  - **Communications**: Establish communication, message passing, shared memory.
  - **Protection**: Control resource access, permissions

Chapter 3: Process Management

Process Concepts

- **Process**: A program in execution; considered the fundamental unit of work in an operating system.
- **Program vs Process**: A program is passive (static code), whereas a process is active (execution state)
- **Process State**: Includes new, running, waiting, ready, and terminated states.
- **Process Control Block (PCB)**: Stores information about a process, including process state, program counter, CPU registers, memory limits, and I/O status.

Process Scheduling

- **Scheduler**: Determines which processes run when there are multiple processes.
  - **Long-Term Scheduler (Job Scheduler)**: Selects processes from the job pool and loads them into memory for execution.
  - **Short-Term Scheduler (CPU Scheduler)**: Selects from among the processes that are ready to execute and allocates the CPU to one of them.
  - **Medium-Term Scheduler**: Involves swapping processes in and out of memory to optimize processing.
- **Context Switch**: Switching the CPU from one process to another, saving the state of the old process and loading the saved state for the new process. This incurs overhead as no useful work is done during the switch.

Operations on Processes

- **Process Creation**: Parent processes create child processes, which, in turn, create other processes, forming a process tree.
  - **Unix Example**: The `fork()` system call creates a new process.

- **Process Termination**: A process ends either voluntarily by calling an exit system call or involuntarily by being terminated by the OS (e.g., due to an error).
- **Cooperating Processes**: Processes that can affect or be affected by other processes, needing mechanisms for communication and synchronization.

Inter-Process Communication (IPC)

- **Shared Memory**: A region of memory is shared between cooperating processes. Faster, but requires synchronization.
- **Message Passing**: Involves sending messages between processes. Easier to implement but can be slower than shared memory.
- **Synchronization**: Ensures that processes do not interfere with each other while sharing resources.

Communication in Client-Server Systems

- **Sockets**: Used for communication between client and server systems. Acts as an endpoint for communication, typically over a network.
- **Remote Procedure Calls (RPCs)**: Allows a program to cause a procedure to execute in another address space, providing a high-level form of IPC.
- **Pipes**: A conduit allowing two processes to communicate. Can be unidirectional or bidirectional.

Chapter 4: Operating System Design and Structure

System Programs

- **System Programs**: Provide an environment for program development and execution.
  - Examples include:
    - **File Management**: Create, delete, copy, rename, and list files.
    - **Status Information**: Provides information like date, time, memory usage, etc.
    - **File Modification**: Text editors for creating and modifying files.
    - **Programming Language Support**: Compilers, assemblers, debuggers, and interpreters.
    - **Program Loading and Execution**: Utilities to load and execute programs.
    - **Communications**: Virtual connections among processes and users.
- **System Calls vs. System Programs**: System calls handle low-level OS requests, whereas system programs provide a user-friendly environment for utilizing these system services.

Operating System Design and Implementation

- **Design Goals**:
  - **User Goals**: OS should be convenient, reliable, and fast.
  - **System Goals**: OS should be easy to design, implement, and maintain.
- **Policy vs. Mechanism**
  - **Policy**: Defines *what* will be done.

    - **Mechanism**: Defines *how* to do it.
    - Separation of these allows flexibility in OS design.
- **Programming Languages**: OS implementation often uses a mix of programming languages, such as assembly for low-level code and C/C++ for the main body of the system.

Operating System Structure

- **Simple Structure (MS-DOS)**:
    - Designed for functionality with minimal space; lacks proper modular separation.
- **Monolithic Structure (UNIX)**:
    - Consists of a kernel that manages everything below the system-call interface and above hardware.
    - Provides core functions like file systems, CPU scheduling, memory management, etc.
- **Layered Approach**:
    - OS divided into layers, with each layer building on top of the lower ones.
    - Bottom layer is hardware, and the top layer is the user interface.
- **Microkernel System Structure**:
    - Moves as much functionality as possible from the kernel into user space.
    - **Benefits**: Easier to extend, more reliable, and more secure.
    - **Detriments**: Performance overhead due to user-kernel space communication.
- **Modular Approach**:
    - Uses loadable kernel modules (LKMs) for flexibility.
    - Similar to the layered approach but allows more dynamic addition of functionalities.
- **Hybrid Systems**:
    - Most modern OSs are hybrids, combining monolithic, microkernel, and modular approaches.
    - Examples: Linux, Solaris, Windows, macOS, and Android.

System Boot

- **Bootstrapping**: The process of starting a computer by loading the kernel.
    - **Bootstrap Loader**: A small piece of code in ROM that locates and loads the OS kernel into memory to start execution.

Chapter 6: Process Concepts and Management

Process Concepts

- **Process**: A program in execution; forms the basis of all computation.
- **Process State**: As a process executes, it changes state:
    - **New**: The process is being created.
    - **Running**: Instructions are being executed.
    - **Waiting**: The process is waiting for some event to occur.

    - **Ready**: The process is waiting to be assigned to a processor.
    - **Terminated**: The process has finished execution.

Process Control Block (PCB)

- **PCB**: Represents each process in the operating system; also known as a task control block.
- **Information in PCB**:
    - **Process State**: Current state (e.g., running, waiting).
    - **Process ID**: Unique identifier for each process.
    - **Program Counter**: Location of next instruction to execute.
    - **CPU Registers**: Contents of process-specific registers.
    - **CPU Scheduling Information**: Priorities, scheduling pointers.
    - **Memory Management Information**: Memory allocated to the process.
    - **Accounting Information**: CPU usage, clock time used, time limits.
    - **I/O Status Information**: Allocated I/O devices and open files.

Process Scheduling

- **Objective**: Maximize CPU use by quickly switching between processes for time-sharing.
- **Scheduling Queues**:
    - **Job Queue**: Set of all processes in the system.
    - **Ready Queue**: Set of processes residing in main memory, ready and waiting to execute.
    - **Device Queue**: Set of processes waiting for an I/O device.
    - Processes may migrate among the various queues.
- **Schedulers**:
    - **Short-Term Scheduler (CPU Scheduler)**: Selects which process should be executed next; must be fast and is invoked frequently (milliseconds).
    - **Long-Term Scheduler (Job Scheduler)**: Selects which processes should be brought into the ready queue; invoked less frequently (seconds or minutes) and controls the degree of multiprogramming.

Context Switch

- **Context Switching**: When the CPU switches from one process to another, the system saves the state of the old process and loads the saved state for the new process.
- **Context-Switch Time**: Represents pure overhead; the system performs no useful work during this time.
- **Example**: Similar to suspending an activity (e.g., reading a book) and resuming it later when interrupted (e.g., mother calling)

Operations on Processes

- **Process Creation**:

- **Parent Process** creates **Child Processes**, forming a tree of processes.
- Each process is identified via a **Process Identifier (PID)**.
- **Resource Sharing Options**:
    - Parent and children share all resources.
    - Children share a subset of the parent's resources.
    - Parent and child share no resources.
- **Execution Options**:
    - Parent and children execute concurrently.
    - Parent waits until children terminate.
- **Address Space**:
    - Child is a duplicate of the parent.
    - Child has a new program loaded into its address space.
- **UNIX Example**:
    - `fork()` system call creates a new process.
    - `exec()` system call replaces the process's memory space with a new program.

Slide Set 7: Process Termination and Inter process Communication

Process Termination

- **Process Termination**: Process executes its last statement and requests OS to delete it using `exit()` system call.
    - Returns status data from child to parent using `wait()`.
    - Resources are deallocated by the OS.
    - **Parent Terminating Children**: Parent may terminate the execution of children processes using `abort()`.
        - Reasons include exceeding allocated resources, task no longer needed, or parent termination (cascading termination).
    - **Cascading Termination**: When a parent terminates, all its children are terminated.
    - **Wait System Call**: Parent process may wait for child termination by invoking `wait()`. If no parent is waiting, the terminated process becomes a **zombie**. If a parent terminates without invoking `wait()`, the child becomes an **orphan**.

Interprocess Communication (IPC)

- **Independent vs. Cooperating Processes**:
    - **Independent**: Processes that cannot affect or be affected by other processes.
    - **Cooperating**: Processes that can affect or be affected by others, requiring communication.
    - **Reasons for Cooperation**: Information sharing, computation speedup, modularity, and convenience.
- **IPC Models**:
    - **Shared Memory**: Processes share a specific memory space.

- **Message Passing**: Processes communicate via messages (e.g., `send(message)` and `receive(message)`).
    - Message passing can be **Direct** (explicitly named processes) or **Indirect** (via **mailboxes**/ports).

Communication Models

- **Shared Memory**:
    - Memory area shared among processes that wish to communicate.
    - The user processes control the synchronisation of access to this memory.
    - **Synchronisation**: A key issue for ensuring correct access by multiple processes.
- **Message Passing**:
    - Mechanism for communication without shared variables.
    - **Direct Communication**: Processes must explicitly name each other.
        - Links are automatically established and usually bi-directional.
    - **Indirect Communication**: Uses mailboxes (or ports).
        - Mailboxes have unique IDs and can be shared among multiple processes.
        - Operations include creating a new mailbox, sending and receiving messages, and destroying a mailbox.
    - **Synchronisation**: Message passing can be blocking (synchronous) or non-blocking (asynchronous).

Slide Set 8: Threads and Concurrency

Thread Concepts

- **Thread**: The fundamental unit of CPU utilization within a process.
    - A thread consists of a **Thread ID**, **Program Counter**, **Register Set**, and a **Stack**.
    - Threads belonging to the same process share the **Code Section**, **Data Section**, and **OS Resources** such as files and signals.
    - A single process can have multiple threads, each capable of handling separate tasks.

Benefits of Threads

- **Responsiveness**: Allows continued execution even if part of the process is blocked (e.g., user interfaces).
- **Resource Sharing**: Threads share resources of a process, making it easier compared to shared memory or message passing.
- **Economy**: Cheaper than process creation, with lower overhead for switching.
- **Scalability**: Can take advantage of multiprocessor architectures, improving performance.

User Threads and Kernel Threads

- **User Threads**: Managed at the user level by thread libraries, without kernel intervention.
  - Examples of Thread Libraries: **POSIX Pthreads**, **Windows threads**, **Java threads**.
- **Kernel Threads**: Managed and supported directly by the OS kernel.
  - Supported by most general-purpose operating systems, including **Windows**, **Linux**, **macOS**, and **Solaris**.

Multithreading Models

- The relationship between user threads and kernel threads is established through three primary models:
  - **Many-to-One**: Many user-level threads are mapped to a single kernel thread.
  - **One-to-One**: Each user-level thread maps to its own kernel thread.
  - **Many-to-Many**: Many user-level threads are mapped to an equal or smaller number of kernel threads.

Threading Issues

- **Semantics of fork() and exec()**:
  - **fork()**: Duplicates the calling process. In a multithreaded program, two types of behaviour are possible:
    - Duplicate all threads.
    - Duplicate only the calling thread.
  - **exec()**: Replaces the process's memory space with a new program. Works as expected, replacing all threads.
- **Thread Cancellation**:
  - **Target Thread**: The thread that is to be cancelled.
  - **Asynchronous Cancellation**: Terminates the target thread immediately.
  - **Deferred Cancellation**: The target thread periodically checks whether it should terminate, offering a controlled way of cancelling.

Slide Set 10: CPU Scheduling

Basic Concepts

- **CPU Scheduling** is fundamental for multiprogramming systems.
- **CPU Burst**: the time a process spends executing instructions on the CPU.
- **I/O Burst**: the time a process waits for i/o operations, such as reading the disk or waiting for user input.
- **CPU-I/O Burst Cycle**: Process execution alternates between **CPU bursts** (computation) and **I/O bursts** (waiting). The cycle alternates until the process is complete.
- **CPU Scheduler**: also known as Short-term scheduler selects from among the processes in the ready queue to allocate CPU time.
  - Scheduling occurs under the following circumstances:

    1. **Non-preemptive**: Process switches from running to waiting state or when it terminates.
    2. **Preemptive**: Process switches from running to ready state or from waiting to ready state.
  - Its main functions include Decision making, Efficiency, and context switching.

Preemptive Vs. Non-Preemptive

- **Preemptive**: Lets OS interrupt and suspend current process **before** if completes its CPU burst. This ensures higher priority tasks don't get neglected.
  - **Key Characteristics**
    - **Interruptions Allowed:** Processes can be interrupted at any time, especially when a higher-priority process arrives.
    - **Time Sharing:** Facilitates time-sharing systems where multiple processes share CPU time effectively.
    - **Responsive:** Enhances system responsiveness, particularly important for interactive and real-time applications.
  - Advantages are responsiveness, fair allocation of cpu, and better handling of real time tasks.
  - Disadvantages are increased overhead, complex implementation, and potential of starvation.
- **Non-preemptive**: Ensures once a process starts its burst, it runs to completion or voluntarily relinquishes control.
  - **Key Characteristics**: No interruptions, simpler to implement, predictable behaviour.
  - Advantages are lower overhead, simplicity, and avoiding starvation.
  - Disadvantages are that its less responsive, potential for poor CPU utilisation, and is less suitable for real time systems.

Dispatcher

- **Dispatcher**: Gives control of the CPU to the process selected by the short-term scheduler.
  - Involves **switching context**, **switching to user mode**, and **jumping to the proper location** in the user program to resume execution.
  - **Dispatch Latency**: The time taken by the dispatcher to stop one process and start another.
  - **Execution Transfer**: initiating the execution of the chosen process by the CPU.

Scheduling Criteria

- **CPU Utilisation**: Keep the CPU as busy as possible.
- **Throughput**: Number of processes that complete their execution per time unit.
- **Turnaround Time**: Time taken to execute a particular process.
- **Waiting Time**: Time a process has been waiting in the ready queue.
- **Response Time**: Time it takes from submission of a request until the first response is produced.

Scheduling Algorithms

- **First-Come, First-Served (FCFS)**:

    - Processes are assigned to the CPU in the order they arrive.
    - disadvantage is **Convoy Effect**: Short processes waiting behind long processes.
- **Round Robin (RR)**:
    - Each process gets a small unit of CPU time (time quantum).
    - If the process does not finish in its allotted time, it is added to the end of the ready queue.
    - Balances response time for time-sharing systems.
    - **Time Quantum**: Should be large compared to context-switch time to reduce overhead.

Slide Set 11: Advanced CPU Scheduling Techniques

Priority Scheduling

- **Priority Scheduling**: Each process is assigned a priority number.
    - **CPU Allocation**: Process with the highest priority (smallest integer) is allocated the CPU.
    - Can be **Preemptive** or **Nonpreemptive**.
    - **SJF as Priority Scheduling**: Shortest-Job-First (SJF) can be considered priority scheduling where priority is the inverse of the predicted next CPU burst time.
    - **Starvation Problem**: Low-priority processes may never execute.
    - **Solution - Aging**: Gradually increase the priority of waiting processes over time.
- **Example**:
    - Processes with different priorities are scheduled, and the average waiting time is calculated to assess efficiency.

Shortest-Job-First (SJF) Scheduling

- **SJF Scheduling**: Associates each process with the length of its next CPU burst.
    - **Optimal**: Produces the minimum average waiting time for a given set of processes.
    - **Difficulty**: Knowing the exact length of the next CPU burst is challenging.
- **Preemptive Version**: Known as **Shortest Remaining Time First (SRTF)**, where the scheduler may preempt a running process if a new process arrives with a shorter burst time.

Example Scheduling Scenarios

- **Example of Preemptive SJF**:
    - Involves processes arriving at different times, leading to frequent context switches for optimal scheduling.
    - **Average Waiting Time**: Calculated to demonstrate efficiency.
- **Priority Scheduling Example**:
    - Multiple processes are scheduled based on priority, and **Gantt Chart** is used to visualise scheduling.
    - **Average Waiting Time**: Summarised for comparison with other scheduling approaches.

Slide Set 12: Scheduling Algorithm Evaluation and Examples

Algorithm Evaluation

- **Evaluating CPU Scheduling Algorithms**: Criteria to consider when selecting an appropriate CPU scheduling algorithm for a system.
- **Deterministic Modelling**: A type of analytic evaluation that uses a predefined workload to evaluate the performance of scheduling algorithms.
    - Simple and fast, but requires exact numbers and only applies to predetermined inputs.
    - Example: Applying **FCFS**, **SJF**, and **Round Robin** (RR) to compare their respective average waiting times.

Example of Preemptive Priority Scheduling

- **Preemptive Priority Scheduling**: CPU is allocated to the highest priority process.
    - **Example Scenario**: Different processes arrive at different times with specific priorities and burst times.
    - **Gantt Chart Representation**: Visual representation of how processes are scheduled based on priority.
    - **Average Waiting Time**: Calculated to assess the effectiveness of the scheduling.

Activity and Algorithm Selection

- **Activity Example**: Given three processes, analyze the completion order for different algorithms (**FCFS**, **SJF**, **Priority Queue**, **Round Robin**).
- **Algorithm Preference**: Factors to consider when choosing an algorithm include average waiting time, response time, and process priorities.

Slide Set 13: Real-Time and Multiple-Processor Scheduling

Multiple-Processor Scheduling

- **Multiple Processors**: Scheduling is more complex with multiple CPUs.
- **Asymmetric Multiprocessing**: Only one processor accesses the system data structures, reducing the need for data sharing.
- **Symmetric Multiprocessing (SMP)**: Each processor is self-scheduling, either using a common ready queue or having its own private queue.
- **Load Balancing**:
    - **Push Migration**: Periodic task checks load and pushes tasks from overloaded CPUs to others.
    - **Pull Migration**: Idle processors pull waiting tasks from busy processors to balance workload.

Real-Time Scheduling

- **Priority-Based Scheduling**: Essential for real-time systems, which can be **soft** or **hard** real-time.
    - **Soft Real-Time**: Provides priority scheduling but without strict deadline guarantees.
    - **Hard Real-Time**: Requires strict guarantees to meet deadlines.

- **Periodic Tasks**: Require CPU at constant intervals, defined by **processing time (t)**, **deadline (d)**, and **period (p)**.

Rate Monotonic Scheduling (RMS)

- **RMS**: Assigns priority based on the inverse of the period.
  - **Shorter Periods = Higher Priority**.
  - Example:
    - Process P1 (Capacity 3, Period 20), Process P2 (Capacity 2, Period 5), Process P3 (Capacity 2, Period 10).
    - Shorter periods receive higher priority, ensuring timely completion.

Earliest Deadline First Scheduling (EDF)

- **EDF**: Priorities assigned based on deadlines.
  - **Earlier Deadline = Higher Priority**.
  - Example:
    - Process P1 (Capacity 3, Period 20, Deadline 7), Process P2 (Capacity 2, Period 5, Deadline 4), Process P3 (Capacity 2, Period 10, Deadline 8).
  - Ensures processes with the most imminent deadlines are executed first.

Slide Set 14: Process Synchronization

Process Synchronization Concepts

- **Synchronization** is crucial to maintain consistency in shared data.
- **Critical-Section Problem**: Ensures that when one process is executing in its critical section, no other process can execute in its critical section.
- **Cooperating Processes**:
  - **Directly Share Logical Address Space**: Share both code and data.
  - **Indirectly Share**: Share data via files or messages.
  - **Concurrent Access Problem**: Access to shared data concurrently may lead to data inconsistency.
  - **Solution**: Implement synchronization mechanisms to ensure orderly access.

Producer-Consumer Problem

- **Producer-Consumer Problem**: Classic synchronization problem.
  - **Shared Buffer**: Acts as the region where both producer and consumer operate.
  - **Two Buffer Types**:
    - **Unbounded Buffer**: No limit on the size; producer can continue producing regardless of consumer actions.

    - **Bounded Buffer**: Fixed size; producer must wait if the buffer is full, and the consumer waits if it is empty.
  - **Synchronization**: Ensures that producers and consumers do not clash over accessing the buffer.

Race Condition

- **Race Condition**: Occurs when multiple processes access shared data and the final outcome depends on the order of execution.
  - **Example**:
    - **Counter Update**: Operations like `counter++` and `counter--` can lead to incorrect values if interleaved improperly.
    - Consider a scenario where multiple processes manipulate a shared counter, leading to inconsistent states if proper synchronization isn't implemented.

Solutions to Critical-Section Problem

- **Hardware-Based Solutions**: Hardware support to control critical sections, preventing simultaneous access.
- **Software-Based Solutions**: Algorithmic methods to solve the critical-section problem without hardware intervention (e.g., Peterson's solution).
- **Semaphores**: Widely used synchronization tool that uses two atomic operations, **wait** and **signal**, to manage concurrent access.
  - **Binary Semaphore (Mutex)**: Provides mutual exclusion.
  - **Counting Semaphore**: Useful for managing a resource pool of finite instances.

Slide Set 15: Synchronization Tools and Critical Section Problem

Background on Process Synchronization

- **Process Synchronization** is necessary to ensure that processes sharing data do not end up causing inconsistencies in the shared data due to concurrent access.
- **Cooperating Processes**: Processes that affect or are affected by other processes in the system. These processes may either share data directly or communicate via files or messages.
- **Problem**: When processes concurrently access shared data, it may lead to **data inconsistency**
- **Solution**: Use **process synchronization** techniques to ensure the orderly execution of cooperating processes.d

The Critical-Section Problem

- The **Critical Section** is the part of the code where shared resources are accessed and modified.
- The **Critical-Section Problem** is about designing a protocol to ensure that no two processes are in their critical section simultaneously.
- General Structure:

  - Each process must request permission to enter its **critical section** (entry section).
  - After leaving the critical section, it must execute an **exit section** and then proceed to the **remainder section**.
- **Requirements for a Solution**:
  1. **Mutual Exclusion**: Only one process should be in the critical section at a time.
  2. **Progress**: If no process is in the critical section and some want to enter, one must be allowed to proceed without indefinite postponement.
  3. **Bounded Waiting**: There should be a bound on how long a process waits before entering its critical section.

Peterson's Solution

- **Peterson's Solution** is a classic software-based solution to the **critical-section problem** and works for two processes
- The solution relies on two variables:
  - **turn**: Indicates whose turn it is to enter the critical section.
  - **flag[]**: Indicates if a process is ready to enter its critical section.
- **Algorithm** for Process Pi:
  - Set `flag[i] = true` to indicate readiness.
  - Set `turn = j` to give the other process a chance.
  - Wait while the other process is ready and it's their turn.
  - Enter **critical section**.
  - After the critical section, set `flag[i] = false` to release.
- Peterson's solution meets the requirements of **mutual exclusion**, **progress**, and **bounded waiting**.

Synchronization Hardware

- Some systems provide **hardware support** for process synchronization.
- **Locks**: Protect critical regions by acquiring and releasing locks.
  - On **uniprocessor systems**, synchronization can be achieved by **disabling interrupts** during the execution of critical sections.
- **Test and Set Instruction**:
  - A hardware instruction that helps achieve **atomic operations** to avoid race conditions.
  - Shared Boolean variable `lock` is used to protect the critical section.
  - **Algorithm**:
    - Use `test_and_set(&lock)` to acquire the lock.
    - Execute the **critical section**.
    - Set `lock = false` to release.

Slide Set 16: Synchronization Tools (Mutexes and Semaphores)

Mutex Locks

- A **mutex lock** is a simple synchronization tool used to ensure that only one thread can access the critical section at a time.
- **Acquire and Release**:
  - Before entering the critical section, a process must **acquire** the lock.
  - After completing the critical section, it must **release** the lock.
- **Drawback**: Simple mutex locks can lead to **busy waiting**, also known as **spinlock**, where the process repeatedly checks the lock until it is available.

Semaphores

- **Semaphores** are more advanced synchronization tools that can coordinate access to resources among multiple processes or threads.
- A **Semaphore S** is an integer variable that is accessed via two atomic operations:
  1. **wait(S)** (also called `P()`): Decrements `S` if it is greater than 0; if `S` is 0, the process must wait.
  2. **signal(S)** (also called `V()`): Increments `S` and potentially allows a waiting process to proceed.
- **Binary Semaphores** (also called **mutex locks**): Semaphore values are restricted between 0 and 1, effectively serving as a simple lock.
- **Counting Semaphores**: These can take on any non-negative integer value, allowing multiple units of a resource to be controlled.

Solutions Using Semaphores

- **Busy Waiting**: The basic implementation of semaphores can lead to **busy waiting**, where a process waits in a loop until the semaphore value changes.
- **Blocking and Wakeup**: Instead of busy waiting, semaphores are often implemented with a mechanism to **block** a process if it cannot proceed, and **wake it up** once the resource is available.
- Semaphores can solve various classic synchronization problems, such as the **Producer-Consumer Problem** and the **Readers-Writers Problem**.