Lecture: Introduction to Operating Systems

(Module: Definition, Types, and Kernel vs. User Mode)

---

1. What is an Operating System (OS)?

Definition

An Operating System (OS) is system software that acts as an intermediary between users and computer hardware. It manages hardware resources (CPU, memory, disk) and provides services to run applications.

Objectives of an OS

1. Resource Management: Allocate CPU, memory, and I/O devices efficiently.
2. Abstraction: Hide hardware complexity (e.g., files instead of disk blocks).
3. User Interface: Provide CLI (Command Line) or GUI (Graphical User Interface).
4. Security & Protection: Prevent unauthorized access (e.g., user permissions).

Key Functions of an OS

Function	**Description**
Process Management	Creates, schedules, and terminates processes (e.g., `fork()` in Linux).
Memory Management	Allocates RAM to processes (e.g., virtual memory, paging).
File Management	Manages files on disk (e.g., FAT32, NTFS).

Device Management	Controls printers, disks, etc., via drivers.
Security	Enforces login/auth, file permissions (e.g., `chmod` in Unix).

---

2. Types of Operating Systems

(1) Batch OS

- Era: 1950s (Mainframes).
- Workflow: Jobs collected in batches → executed sequentially.
- Example: Early payroll systems.
- Limitation: No user interaction during execution.

(2) Multiprogramming OS

- Goal: Maximize CPU usage by running multiple programs concurrently.
- Mechanism: When one program waits for I/O, another runs.
- Example: IBM OS/360.

(3) Time-Sharing OS

- Goal: Interactive computing (multiple users share CPU time).
- Feature: Rapid switching between tasks (*preemption*).
- Example: Unix, Multics.

(4) Real-Time OS (RTOS)

- Goal: Predictable timing (deadlines in milliseconds).
- Types:
  - Hard RTOS (Missed deadline = failure; e.g., pacemakers).
  - Soft RTOS (Tolerable delays; e.g., streaming systems).
- Example: VxWorks, FreeRTOS.

(5) Distributed OS

- Goal: Manage networked computers as a single system.
- Features: Fault tolerance, load balancing.
- Example: Google's Borg (predecessor to Kubernetes).

(6) Embedded OS

- Goal: Run on resource-constrained devices.
- Example: Android (for mobile), QNX (for cars).

---

3. Kernel vs. User Mode

Why Two Modes?

- Safety: Prevent user programs from crashing the OS.
- Control: Restrict direct hardware access.

Kernel Mode (Privileged Mode)

- Rings: Ring 0 (x86) or EL1 (ARM).
- Privileges: Full hardware access (e.g., modify page tables).
- Risks: Bugs cause system crashes (*kernel panic*).

User Mode

- Rings: Ring 3 (x86) or EL0 (ARM).
- Restrictions: Hardware access via system calls (e.g., `read()`).
- Safety: Process crashes don't affect the OS.

How They Interact?

1. User program calls `write()` (a system call).
2. CPU switches to kernel mode.
3. Kernel executes the request.
4. Returns to user mode.

```
c

Copy
```

Download

```c
// Example: System call in C
#include <unistd.h>
int main() {
  write(1, "Hello Kernel!", 13);  // Invokes kernel mode
}
```

Key Takeaways

1. OS Roles: Resource manager, abstraction layer, security enforcer.
2. OS Types: Batch → Distributed → Embedded (evolution driven by needs).
3. Kernel Mode: Trusted, powerful, risky.
4. User Mode: Restricted, safe, relies on syscalls.

Quiz (To Test Your Understanding)

1. Why does an OS use kernel/user mode separation?
2. Which OS type would you choose for a self-driving car? Why?
3. What happens if a user program tries to execute a privileged instruction?

Lecture on Operating Systems: Process Management

1. Introduction to Process Management

Process Management is a fundamental function of an operating system (OS) that deals with the creation, scheduling, and termination of processes. It ensures efficient CPU utilization, fairness, and responsiveness.

2. Process vs. Thread

Process

- A process is an instance of a running program.
- It has its own address space, file descriptors, and system resources.
- Processes are isolated from each other (one process crashing doesn't affect others).
- Communication between processes (Inter-Process Communication, IPC) is slower due to isolation.

Thread

- A thread is a lightweight unit of execution within a process.
- Multiple threads share the same address space and resources of the parent process.
- Threads are faster to create and switch compared to processes.
- Communication between threads is faster (shared memory), but requires synchronization (locks, semaphores).

Feature	**Process**	**Thread**
Isolation	High (separate memory space)	Low (shared memory)
Creation Cost	High (OS involvement)	Low (managed by process)

Communication	Slow (IPC required)	Fast (shared memory)
Crash Impact	Only the process crashes	Can crash entire process

---

3. Process States

A process transitions through different states during its lifecycle:

1. New: The process is being created.
2. Ready: The process is loaded into memory and waiting for CPU time.
3. Running: The process is executing on the CPU.
4. Waiting/Blocked: The process is waiting for an event (I/O, signal).
5. Terminated: The process has finished execution.

https://www.gatevidyalay.com/wp-content/uploads/2018/11/Process-State-Diagram.png

---

4. Process Scheduling Algorithms

The OS scheduler selects which process runs next based on scheduling policies.

(a) First-Come, First-Served (FCFS)

- Non-preemptive (once a process starts, it runs to completion).
- Simple but suffers from convoy effect (short jobs wait behind long ones).

(b) Shortest Job First (SJF)

- Picks the process with the shortest burst time.

- Can be preemptive (Shortest Remaining Time First, SRTF) or non-preemptive.
- Optimal for minimizing average waiting time but requires future knowledge.

(c) Priority Scheduling

- Each process has a priority; the highest priority runs first.
- Can lead to starvation (low-priority processes never run).
- Solution: Aging (gradually increasing priority of waiting processes).

(d) Round Robin (RR)

- Each process gets a fixed time quantum (e.g., 10ms).
- Preemptive—if a process doesn't finish in its quantum, it goes back to the ready queue.
- Fair but may have high overhead due to frequent context switches.

(e) Multilevel Queue Scheduling

- Processes are grouped into multiple priority queues (e.g., foreground (interactive) vs. background (batch)).
- Each queue may use a different scheduling algorithm (e.g., RR for foreground, FCFS for background).
- Can lead to starvation if lower-priority queues are neglected.

---

5. Context Switching

- When the CPU switches from one process to another, the OS must:
    1. Save the state (PCB - Process Control Block) of the current process.
    2. Load the state of the next process.
- Overhead: Context switching is costly (CPU cycles wasted on saving/loading registers, cache flushes).
- Minimizing Context Switches: Using efficient scheduling (larger time quanta in RR) reduces overhead.

---

6. Inter-Process Communication (IPC)

Processes need to communicate for coordination, data sharing, and synchronization.

(a) Shared Memory

- Processes access a common memory region.
- Faster than message passing (no kernel involvement after setup).
- Requires synchronization (mutexes, semaphores) to avoid race conditions.

(b) Message Passing

- Processes communicate via kernel-managed messages (e.g., pipes, sockets, signals).
- Slower (kernel intervention needed) but avoids synchronization issues.
- Two models:
    - Direct Communication (sender/receiver explicitly named).
    - Indirect Communication (uses mailboxes/ports).

Feature	**Shared Memory**	**Message Passing**
Speed	Fast (direct access)	Slow (kernel calls)
Synchronization	Required (race conditions)	Not needed (kernel handles)
Complexity	Harder to implement	Easier to implement

---

7. Summary

- Process vs. Thread: Processes are isolated, threads share resources.
- Process States: New, Ready, Running, Waiting, Terminated.
- Scheduling Algorithms: FCFS, SJF, Priority, RR, Multilevel Queue.
- Context Switching: Saves and restores process state (overhead involved).
- IPC: Shared Memory (fast but needs sync) vs. Message Passing (slow but safer).

---

8. Questions for Discussion

1. Why is Round Robin considered fair but inefficient for long processes?
2. How does Priority Scheduling lead to starvation, and how can it be prevented?
3. Compare the advantages of threads over processes in a multi-core system.
4. When would you prefer Shared Memory over Message Passing?

Lecture on Operating Systems: Threads & Multithreading

1. Introduction to Threads

A thread is the smallest unit of execution within a process. Unlike processes, threads share the same memory space and resources, making them lightweight and efficient for concurrent execution.

Why Use Threads?

- Responsiveness: Allows a program to remain responsive even if part of it is blocked (e.g., GUI applications).
- Resource Sharing: Threads share memory, avoiding costly IPC mechanisms.
- Scalability: Better utilization of multi-core CPUs.

- Economy: Creating threads is faster and consumes fewer resources than processes.

---

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

Feature	**User-Level Threads (ULT)**	**Kernel-Level Threads (KLT)**
Managed by	User-space (Thread library, e.g., POSIX Pthreads)	OS Kernel
Context Switching	Fast (No kernel mode switch)	Slow (Requires kernel intervention)
Blocking Issue	If one thread blocks, entire process blocks	One thread can block while others run
Scheduling	Managed by application (custom scheduling)	Managed by OS scheduler
Multi-core Support	Limited (All ULTs run on a single kernel thread)	Yes (Threads can run on different CPU cores)

Examples	GNU Portable Threads (GPT), Java Green Threads	Windows OS Threads, Linux Kernel Threads

Advantages of ULTs

✔ Faster thread creation & switching (no kernel mode switch).

✔ Custom scheduling policies possible.

✔ Portable (runs on any OS with thread library support).

Disadvantages of ULTs

❌ No true parallelism (all ULTs map to one kernel thread).

❌ If one thread blocks (e.g., I/O), the whole process blocks.

Advantages of KLTs

✔ True parallelism (threads can run on multiple CPU cores).

✔ One blocked thread doesn't block others.

Disadvantages of KLTs

❌ Slower thread operations (kernel mode switch required).

❌ Less flexible scheduling (OS decides, not application).

---

3. Multithreading Models

The relationship between user threads and kernel threads is defined by three models:

(1) Many-to-One Model

- Multiple user threads → One kernel thread.
- Pros: Fast, lightweight.
- Cons: No parallelism (entire process blocks if one thread blocks).
- Example: Older threading implementations (Green Threads in Solaris).

(2) One-to-One Model

- Each user thread → One kernel thread.
- Pros: True parallelism, no blocking issues.
- Cons: Overhead due to many kernel threads.
- Example: Windows, Linux (NPTL threads).

(3) Many-to-Many Model (Hybrid Model)

- Multiple user threads → Multiple kernel threads.
- Pros: Balances efficiency and parallelism.
- Cons: Complex implementation.
- Example: Solaris, modern POSIX threads (with thread pools).

Model	**Parallelism**	**Blocking Issue**	**Scalability**	**Example**
Many-to-One	❌ No	❌ Entire process blocks	❌ Poor	Green Threads
One-to-One	✔ Yes	✔ No blocking	❌ High overhead	Windows, Linux

Many-to-Many	✔ Yes	✔ No blocking	✔ Balanced	Solaris, POSIX

---

4. Thread Synchronization Issues

When multiple threads access shared resources, race conditions occur, leading to data corruption.

Common Problems

1. Race Condition
   - Two threads modify shared data simultaneously, leading to inconsistent results.
   - Example:
   - c
   - Copy
   - Download

```
// Thread 1: x++ (reads x=5, writes x=6)

// Thread 2: x++ (reads x=5, writes x=6)
```

   - *// Final x = 6 (should be 7)*
2. Deadlock
   - Two or more threads wait indefinitely for each other's resources.
   - Example:
     - Thread 1 holds Lock A and waits for Lock B.
     - Thread 2 holds Lock B and waits for Lock A.
3. Starvation
   - A thread is perpetually denied access to resources due to priority scheduling.
4. Priority Inversion

    ◦ A low-priority thread holds a lock needed by a high-priority thread, causing delays.

---

5. Synchronization Techniques

To prevent race conditions, we use:

(1) Mutex Locks

- Ensures only one thread can access a resource at a time.
- Example (C/Pthreads):

- c
- Copy
- Download

```
pthread_mutex_t lock;

pthread_mutex_lock(&lock); // Critical section
```

- `pthread_mutex_unlock(&lock);`

(2) Semaphores

- A counter that controls access to a resource pool.
- Two types:
    - Binary Semaphore (Mutex-like)
    - Counting Semaphore (N resources)
- Example:

- c
- Copy
- Download

```
sem_t sem;

sem_wait(&sem); // Decrements semaphore (if zero, waits)
```

- `sem_post(&sem); // Increments semaphore`

(3) Condition Variables

- Allows threads to wait for a condition before proceeding.
- Used with mutexes.
- Example:

- c
- Copy
- Download

```
pthread_cond_t cond;

pthread_mutex_t mutex;

pthread_cond_wait(&cond, &mutex); // Waits for signal
```

- `pthread_cond_signal(&cond);       // Wakes one waiting thread`

(4) Spinlocks

- Thread busy-waits instead of sleeping (useful for very short critical sections).
- Example (Linux Kernel):

- c
- Copy
- Download

```
spinlock_t lock;

spin_lock(&lock);   // Critical section
```

- `spin_unlock(&lock);`

---

6. Best Practices for Multithreading

✔ Minimize shared data (use thread-local storage where possible).

✔ Use higher-level constructs (e.g., thread pools, futures).

✔ Avoid deadlocks by acquiring locks in a fixed order.

✔ Prefer immutable data (no locks needed if data doesn't change).

---

7. Summary

- User-Level Threads (ULT): Fast, but no parallelism.
- Kernel-Level Threads (KLT): Slower, but enables true concurrency.
- Multithreading Models: Many-to-One (no parallelism), One-to-One (best parallelism), Many-to-Many (balanced).
- Synchronization Issues: Race conditions, deadlocks, starvation.
- Synchronization Techniques: Mutexes, semaphores, condition variables, spinlocks.

---

8. Discussion Questions

1. Why do many modern OSes (Linux, Windows) use the One-to-One model?
2. How does the Many-to-Many model improve upon the other two?
3. What is the difference between a mutex and a binary semaphore?
4. When would you prefer a spinlock over a mutex?

Lecture on CPU Scheduling in Operating Systems

1. Introduction to CPU Scheduling

CPU Scheduling is the process of selecting which process or thread gets to use the CPU at any given time. The scheduler (a component of the OS) decides the order of execution to optimize:

- CPU Utilization (Keep CPU as busy as possible)
- Throughput (Maximize processes completed per unit time)
- Response Time (Minimize time for interactive requests)
- Fairness (Ensure all processes get a fair share)

---

2. Preemptive vs. Non-Preemptive Scheduling

(a) Non-Preemptive Scheduling

- Once a process starts, it runs until completion (or until it blocks for I/O).
- Pros: Simple, low overhead.
- Cons: Poor responsiveness (long-running process can block others).
- Used in: Batch processing systems.
- Example Algorithms: FCFS, Non-preemptive SJF.

(b) Preemptive Scheduling

- The OS can interrupt a running process and switch to another.
- Pros: Better responsiveness, avoids CPU monopolization.
- Cons: Higher overhead (frequent context switches).
- Used in: Time-sharing, real-time systems.
- Example Algorithms: Round Robin, Preemptive SJF (SRTF), Priority Scheduling.

Feature	**Non-Preemptive**	**Preemptive**
Interruption	No (runs to completion)	Yes (can be interrupted)
Overhead	Low	High (context switches)

Responsiveness	Poor (long jobs block others)	Good (fair CPU sharing)
Use Case	Batch processing	Interactive systems

---

3. Scheduling Criteria (Performance Metrics)

When evaluating scheduling algorithms, we consider:

1. CPU Utilization (%):
   - Percentage of time CPU is busy (ideally close to 100%).
2. Throughput (jobs/sec):
   - Number of processes completed per unit time.
3. Turnaround Time (TAT):
   - Total time from submission to completion (`TAT = Completion Time - Arrival Time`).
4. Waiting Time (WT):
   - Total time a process spends waiting in the ready queue (`WT = TAT - Burst Time`).
5. Response Time (RT):
   - Time from submission until the first response (important for interactive systems).

Goal: Minimize WT, TAT, RT while maximizing Throughput & CPU Utilization.

---

4. CPU Scheduling Algorithms

(1) First-Come, First-Served (FCFS)

- Non-preemptive (runs until completion).
- Pros: Simple, easy to implement.
- Cons: Convoy effect (short jobs stuck behind long ones).
- Example:

- text
- Copy
- Download

```
Processes: P1 (Burst=10), P2 (Burst=5), P3 (Burst=8)

Order: P1 → P2 → P3
```

- `Avg WT = (0 + 10 + 15)/3 = 8.33`

(2) Shortest Job First (SJF)

- Runs the process with the shortest burst time first.
  - Non-preemptive: Runs to completion.
  - Preemptive (SRTF): Can interrupt if a shorter job arrives.
- Pros: Minimizes average waiting time (optimal for non-preemptive).
- Cons: Requires future knowledge (burst time estimation).
- Example (Non-preemptive SJF):

- text
- Copy
- Download

```
Processes: P1 (Burst=6), P2 (Burst=8), P3 (Burst=7), P4 (Burst=3)

Order: P4 → P1 → P3 → P2
```

- `Avg WT = (0 + 3 + 9 + 16)/4 = 7.0`

(3) Priority Scheduling

- Each process has a priority (higher priority runs first).
  - Preemptive: Higher-priority process can interrupt.
  - Non-preemptive: Runs until completion.
- Pros: Useful for real-time systems.
- Cons: Starvation (low-priority jobs may never run).
- Solution: Aging (gradually increase priority of waiting jobs).

(4) Round Robin (RR)

- Each process gets a fixed time quantum (q) (e.g., 10ms).
- Preemptive: If not done, goes back to the ready queue.
- Pros: Fair, good for time-sharing systems.
- Cons: High context switch overhead if `q` is too small.
- Example (q=4):

- text
- Copy
- Download

```
Processes: P1 (Burst=6), P2 (Burst=3), P3 (Burst=1)

Order: P1 (4) → P2 (3) → P3 (1) → P1 (2)
```

- `Avg WT = (6 + 4 + 6)/3 = 5.33`

(5) Multilevel Feedback Queue (MLFQ)

- Multiple queues with different priorities and scheduling policies.
  - Higher-priority queues may use RR (short interactive jobs).
  - Lower-priority queues may use FCFS (long CPU-bound jobs).
- Aging: Jobs move up if they wait too long.
- Pros: Balances responsiveness and throughput.
- Used in: Most modern OS schedulers (Linux, Windows).

---

5. Comparison of Scheduling Algorithms

Algorithm	Preemptive?	Avg WT	Starvation?	Use Case
FCFS	❌ No	High	❌ No	Batch processing

SJF	✔ (SRTF) / ❌	Lowest	❌ No	Batch (non-preemptive)
Priority	✔ / ❌	Medium	✔ Yes	Real-time systems
Round Robin	✔ Yes	Medium	❌ No	Time-sharing
MLFQ	✔ Yes	Balanced	✔ (Aging fixes)	General-purpose OS

---

6. Advanced Topics

(a) Real-Time Scheduling

- Hard Real-Time: Deadlines must be met (e.g., flight control).
- Soft Real-Time: Deadlines are preferred but not critical (e.g., video streaming).

(b) Linux CFS (Completely Fair Scheduler)

- Uses virtual runtime (vruntime) to ensure fairness.
- Dynamically adjusts priorities to balance CPU allocation.

---

7. Summary

- Preemptive vs. Non-Preemptive: Preemptive allows interruption; non-preemptive runs to completion.
- Scheduling Metrics: CPU Utilization, Throughput, Turnaround Time, Waiting Time, Response Time.
- Algorithms:
    - FCFS: Simple but suffers from convoy effect.
    - SJF: Optimal for waiting time but needs burst estimation.
    - Priority: Good for real-time but risks starvation.
    - Round Robin: Fair but high overhead.
    - MLFQ: Best for general-purpose OS (balancing interactivity & throughput).

---

8. Discussion Questions

1. Why is SJF optimal for minimizing average waiting time?
2. How does Round Robin prevent starvation compared to Priority Scheduling?
3. What happens if the time quantum in RR is too large or too small?
4. How does Multilevel Feedback Queue improve upon simple priority scheduling?

Lecture on Process Synchronization & Deadlocks

1. Introduction to Process Synchronization

In a multi-process system, concurrent access to shared resources can lead to race conditions, where the final outcome depends on the order of execution.
Goal: Ensure that only one process accesses a shared resource at a time (mutual exclusion).

---

2. The Critical Section Problem

A critical section is a code segment that accesses shared resources (variables, files, devices).

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 the critical section, a requesting process must be allowed in.
3. Bounded Waiting: No process should wait indefinitely to enter its critical section.

Peterson’s Solution (Software-Based)

- Uses two shared variables (`flag[2]`, `turn`).
- Works for two processes only.

c

Copy

Download

```c
do {
    flag[i] = true;        // Process i wants to enter
    turn = j;              // Let the other process go first
    while (flag[j] && turn == j);  // Wait if the other process is in CS
    // Critical Section
    flag[i] = false;       // Process i exits
    // Remainder Section
} while (true);
```

Limitations:

❌ Not scalable for `N` processes.

❌ Busy waiting (wastes CPU cycles).

---

3. Synchronization Mechanisms

(1) Mutex Locks (Binary Semaphores)

- A lock that ensures mutual exclusion.
- Operations:
  - `acquire()` – Waits until lock is free, then takes it.
  - `release()` – Releases the lock.
- Example (Pthreads):

- c
- Copy
- Download

```
pthread_mutex_t lock;

pthread_mutex_lock(&lock);    // Enter critical section

// Critical Section
```

- `pthread_mutex_unlock(&lock); // Exit critical section`

Limitations:

❌ Busy waiting in simple implementations (spinlock).

❌ Deadlock risk if not used carefully.

(2) Semaphores

- A counter that controls access to a shared resource.
- Types:
  - Binary Semaphore (`0` or `1`) → Acts like a mutex.
  - Counting Semaphore (`N` resources available).
- Operations:
  - `wait()` (or `P`) – Decrements semaphore (blocks if `≤ 0`).
  - `signal()` (or `V`) – Increments semaphore (wakes a waiting process).
- Example (Producer-Consumer Problem):

- c

- Copy
- Download

```
sem_t empty, full, mutex;

sem_init(&empty, 0, BUFFER_SIZE); // Initially all slots empty

sem_init(&full, 0, 0);            // Initially no items

sem_init(&mutex, 0, 1);           // Mutex for buffer access


// Producer:

sem_wait(&empty);  // Wait for empty slot

sem_wait(&mutex);  // Enter critical section

// Add item to buffer

sem_post(&mutex);  // Exit critical section

sem_post(&full);   // Increment full count


// Consumer:

sem_wait(&full);   // Wait for an item

sem_wait(&mutex);  // Enter critical section

// Remove item from buffer

sem_post(&mutex);  // Exit critical section
```

- `sem_post(&empty);  // Increment empty count`

(3) Monitors

- A high-level abstraction that encapsulates shared data and synchronization.
- Features:
  - Mutual exclusion is implicit (only one thread can execute a monitor method at a time).

    - Uses condition variables (`wait`, `signal`, `broadcast`) for synchronization.
- Example (Java `synchronized`):

- java
- Copy
- Download

```java
class Buffer {

    private int item;

    private boolean empty = true;



    public synchronized void produce(int value) {

        while (!empty) wait();  // Wait if buffer is full

        item = value;

        empty = false;

        notify();              // Wake up consumer

    }



    public synchronized int consume() {

        while (empty) wait();  // Wait if buffer is empty

        empty = true;

        notify();              // Wake up producer

        return item;

    }
```

- }

---

4. Deadlocks

A deadlock occurs when two or more processes are blocked forever, each waiting for a resource held by the other.

Four Necessary Conditions (Coffman Conditions)

1. Mutual Exclusion: Only one process can use a resource at a time.
2. Hold and Wait: A process holds a resource while waiting for another.
3. No Preemption: Resources cannot be forcibly taken away.
4. Circular Wait: A cycle exists in the resource allocation graph.

Deadlock Handling Strategies

Strategy	**Approach**	**Pros & Cons**
Prevention	Ensures at least one Coffman condition is false.	✔ Safe but restrictive.
Avoidance	Uses Banker's Algorithm to check safety.	✔ Safe but requires future knowledge.
Detection	Periodically checks for deadlocks.	✔ Flexible but recovery is costly.
Recovery	Kills processes or rolls back.	❌ Disruptive but sometimes necessary.

(1) Deadlock Prevention

- Eliminate Mutual Exclusion: Not possible for non-sharable resources (e.g., printers).
- Eliminate Hold & Wait: Request all resources at once (`atomic allocation`).
- Allow Preemption: Take resources away if needed (e.g., CPU scheduling).
- Break Circular Wait: Enforce a total ordering of resource requests.

(2) Deadlock Avoidance (Banker's Algorithm)

- Resource Allocation Graph (RAG): Checks for cycles.
- Banker's Algorithm:
  - Each process declares max need.
  - System checks if granting a request leads to a safe state (where all processes can complete).
- Example:

- text
- Copy
- Download

```
Available: [3, 3, 2]

Max Need:

  P0: [7, 5, 3]

  P1: [3, 2, 2]

  P2: [9, 0, 2]
```

- `If P1 requests [1, 0, 2], is it safe?`

(3) Deadlock Detection

- Wait-For Graph: A reduced RAG where edges represent waiting.
- Detection Algorithm: Runs periodically to find cycles.

(4) Deadlock Recovery

- Process Termination: Kill all deadlocked processes (or one at a time).

- Resource Preemption: Roll back a process and reassign resources.

---

5. Summary

- Critical Section Problem: Solved using mutexes, semaphores, monitors.
- Deadlock Conditions: Mutual Exclusion, Hold & Wait, No Preemption, Circular Wait.
- Deadlock Handling:
  - Prevention: Breaks one of the four conditions.
  - Avoidance: Banker’s Algorithm ensures safe allocation.
  - Detection & Recovery: Finds and resolves deadlocks post-occurrence.

---

6. Discussion Questions

1. Why is busy waiting undesirable in synchronization mechanisms?
2. How does the Banker’s Algorithm ensure a safe state?
3. What are the trade-offs between deadlock prevention and avoidance?
4. Can deadlocks be completely eliminated in real-world systems?

Lecture on Memory Management in Operating Systems

1. Introduction to Memory Management

Memory management is a critical OS function that handles allocation, deallocation, and optimization of memory for processes. Key objectives:

- Efficient utilization of physical memory.
- Protection & isolation between processes.
- Virtual memory abstraction (illusion of larger memory space).

2. Contiguous vs. Non-Contiguous Memory Allocation

(1) Contiguous Allocation

- Memory is allocated in continuous blocks.
- Types:
    - Fixed Partitioning: Memory divided into fixed-size partitions (early IBM OS).
        - Internal fragmentation: Unused memory within a partition.
    - Dynamic Partitioning: Memory allocated dynamically (best-fit, worst-fit, first-fit).
        - External fragmentation: Free memory gaps between allocations.
- Pros: Simple, low overhead.
- Cons: Fragmentation issues, inflexible.

(2) Non-Contiguous Allocation

- Memory allocated in scattered blocks.
- Techniques:
    - Paging: Divides memory into fixed-size pages.
    - Segmentation: Divides memory into logical segments (code, stack, heap).
- Pros: No external fragmentation, flexible.
- Cons: Overhead due to address translation.

Feature	**Contiguous Allocation**	**Non-Contiguous Allocation**
Fragmentation	External + Internal	No external (only internal in paging)

Flexibility	Low	High
Hardware Support	Simple (base + limit registers)	Complex (MMU, page tables)

---

3. Paging

(1) Basics of Paging

- Divides physical memory into frames and logical memory into pages (same size, e.g., 4KB).
- Page Table: Maps logical pages to physical frames.
  - Each process has its own page table.
  - Stored in main memory (slow access).

(2) Translation Lookaside Buffer (TLB)

- A CPU cache that stores recent page-to-frame translations.
- TLB Hit: Translation found in TLB (fast access).
- TLB Miss: OS must consult page table (slow, may require page fault handling).
- Effective Access Time (EAT):

- text
- Copy
- Download
- `EAT = (TLB Hit Time × Hit Ratio) + (TLB Miss Penalty × Miss Ratio)`

(3) Page Replacement Algorithms

When memory is full, the OS must evict a page to bring in a new one. Key algorithms:

Algorithm	**Description**	**Pros & Cons**
FIFO	Evicts the oldest page.	❌ Belady’s Anomaly: More frames → more page faults.
LRU	Evicts the least recently used page.	✔ Optimal for temporal locality (but hard to implement).
Optimal (OPT)	Evicts the page not used for the longest future time.	✔ Theoretical best (but requires future knowledge).
Clock (Second Chance)	Approximates LRU using a reference bit.	✔ Balances efficiency and simplicity.

Example (FIFO vs. LRU vs. OPT):

text

Copy

Download

```
Reference String: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

Frames = 3

FIFO: 9 page faults

LRU: 7 page faults
```

```
OPT: 6 page faults
```

---

4. Segmentation

- Divides memory into logical segments (e.g., code, stack, heap).
- Segment Table: Maps logical segments to physical addresses.
  - Each entry has base address + limit.
- Pros: Reflects program structure, flexible sharing.
- Cons: External fragmentation, complex management.

Combined Paging + Segmentation (e.g., x86):

- Segments are themselves paged.
- Used in modern systems for backward compatibility.

---

5. Virtual Memory & Demand Paging

(1) Virtual Memory

- Illusion of larger memory than physically available.
- Uses disk (swap space) as an extension of RAM.

(2) Demand Paging

- Pages are loaded only when needed (on a page fault).
- Page Fault Handling:
  1. Trap to OS (invalid page table entry).
  2. Check if page is valid (in disk).
  3. Find a free frame (or evict one).
  4. Load page from disk to memory.
  5. Update page table.
  6. Restart the instruction.

(3) Working Set Model

- Working Set: Set of pages a process actively uses.
- OS tries to keep the working set in memory to minimize page faults.

---

6. Thrashing

- Definition: Excessive paging due to high page fault rate.
- Causes:
    - Overcommitment: Too many processes competing for memory.
    - Poor locality: Processes frequently access non-resident pages.
- Solutions:
    - Working Set Algorithm: Allocate memory based on working set size.
    - Load Control: Reduce multiprogramming level (suspend processes).

---

7. Summary

Concept	**Key Takeaway**
Contiguous Allocation	Simple but suffers from fragmentation.
Paging	Fixed-size pages + page tables; TLB speeds up translation.
Page Replacement	LRU is optimal but hard to implement; Clock is a practical alternative.

Segmentation	Logical division (code, stack); suffers from external fragmentation.
Virtual Memory	Demand paging + disk swapping enables larger address spaces.
Thrashing	Caused by excessive page faults; mitigated by working set algorithms.

---

8. Discussion Questions

1. Why does FIFO suffer from Belady's anomaly while LRU does not?
2. How does the TLB improve paging performance?
3. What are the trade-offs between paging and segmentation?
4. How can an OS detect and recover from thrashing?

Lecture on File Systems & Disk Management

1. Introduction to File Systems

A file system is an OS component that manages storage devices, file organization, and access methods.

Key Functions:

- Abstract storage (files/directories instead of raw blocks).

- Manage disk space (allocation, free space tracking).
- Provide access control (permissions, security).

---

2. File Concepts

(1) File Attributes

Each file has metadata stored in its directory entry or inode:

- Name (human-readable identifier).
- Type (text, binary, executable).
- Location (pointer to disk blocks).
- Size (current + max allowed).
- Timestamps (creation, modification, access).
- Permissions (read/write/execute for owner/group/others).

(2) File Operations

Operation	**Description**
`create()`	Allocates space + adds directory entry.
`open()`	Loads metadata for access.
`read()`	Reads data from file.

`write()`	Writes data to file.
`seek()`	Moves file pointer to a specific offset.
`delete()`	Frees space + removes directory entry.
`truncate()`	Resizes file (discards data beyond new size).

(3) File Types

- Regular files (text, binaries, archives).
- Directories (containers for files/subdirectories).
- Special files (device files like `/dev/sda`).
- Links (hard links, symbolic links).

File Extensions (e.g., `.txt`, `.exe`) hint at type but aren't enforced by OS.

---

3. Directory Structures

(1) Single-Level Directory

- All files in one directory (no subfolders).
- Pros: Simple.
- Cons: No grouping, naming conflicts.
- Example: Early DOS systems.

(2) Two-Level Directory

- Separate directories per user (e.g., `/user1/file1`, `/user2/file1`).
- Pros: Avoids name clashes.
- Cons: No hierarchical organization.

(3) Tree-Structured Directory

- Hierarchy of directories/subdirectories (modern systems).
- Pathnames: Absolute (`/home/user/file`) or relative (`../docs/file`).
- Operations: `mkdir`, `rmdir`, `cd`.

(4) Acyclic-Graph Directory

- Files/directories can have multiple parents (via hard links).
- Pros: Flexible sharing.
- Cons: Requires garbage collection (to delete unreferenced files).

(5) General Graph Directory

- Allows cycles (can lead to infinite loops during traversal).
- Rarely used due to complexity.

Structure	**Example**	**Advantages**	**Disadvantages**
Single-Level	`/file1`, `/file2`	Simple	No organization
Tree	`/home/user/docs`	Intuitive hierarchy	No shared files

Acyclic-Graph	Hard links (`ln`)	Flexible sharing	Garbage collection needed

---

4. File Allocation Methods

(1) Contiguous Allocation

- Files stored in consecutive disk blocks.
- Pros:
  - Fast sequential/random access (minimal seeks).
  - Simple metadata (just `start block + length`).
- Cons:
  - External fragmentation (free gaps between files).
  - Hard to resize files.
- Used in: CD-ROMs, DVDs.

(2) Linked Allocation

- Files as linked lists of blocks (each block points to next).
- Pros:
  - No external fragmentation.
  - Easy file growth.
- Cons:
  - Slow random access (must traverse links).
  - Unreliable (broken link → lost file).
- FAT (File Allocation Table):
  - Centralized table of links (used in old Windows).

(3) Indexed Allocation

- Index block stores pointers to all file blocks.
- Variants:
  - Single-level index (small files).
  - Multi-level index (large files, e.g., Unix inodes).
  - Combined schemes (e.g., direct + indirect blocks in ext4).

- Pros:
  - Fast random access.
  - No external fragmentation.
- Cons:
  - Overhead for index blocks.

Method	Access Speed	Fragmentation	Scalability
Contiguous	Fast (sequential)	External	Poor
Linked	Slow (random)	None	Good
Indexed	Fast (random)	None	Excellent

---

5. Disk Scheduling Algorithms

Goal: Minimize seek time (time to move disk head to desired track).

(1) FCFS (First-Come, First-Served)

- Requests served in arrival order.
- Pros: Fair.
- Cons: High seek time (no optimization).

(2) SSTF (Shortest Seek Time First)

- Picks nearest request to current head position.

- Pros: Low average seek time.
- Cons: Starvation for distant requests.

(3) SCAN (Elevator Algorithm)

- Head moves back-and-forth across disk, servicing requests along the way.
- Pros: No starvation, balanced performance.
- Cons: Long waits for edge tracks.

(4) C-SCAN (Circular SCAN)

- Head moves one direction only (loops back to start after end).
- Pros: More uniform wait times than SCAN.

(5) LOOK & C-LOOK

- LOOK: Reverses direction only when no more requests in current direction.
- C-LOOK: Like C-SCAN but stops at last request.
- More efficient than SCAN/C-SCAN (avoids unnecessary seeks).

Example (Request Queue: 98, 183, 37, 122, 14, 124, 65, 67; Head starts at 53):

Algorithm	**Order of Servicing**	**Total Seek Distance**
FCFS	53 → 98 → 183 → 37 → 122 → 14 → 124 → 65 → 67	640 tracks
SSTF	53 → 65 → 67 → 37 → 14 → 98 → 122 → 124 → 183	236 tracks

SCAN	53 → 37 → 14 → 0 → 65 → 67 → 98 → 122 → 124 → 183	236 tracks (with overhead to 0)
C-LOOK	53 → 65 → 67 → 98 → 122 → 124 → 183 → 14 → 37	299 tracks

---

6. Summary

Topic	**Key Takeaways**
File Attributes	Metadata (name, size, permissions) stored in inodes/directory entries.
Directory Structures	Tree most common; acyclic-graph allows sharing via links.
Allocation Methods	Contiguous (fast but fragmented), Linked (slow random access), Indexed (best balance).

Disk Scheduling	SSTF for low seek time, SCAN/LOOK for fairness.

---

7. Discussion Questions

1. Why does indexed allocation outperform linked allocation for random access?
2. How does the LOOK algorithm improve upon SCAN?
3. What are the trade-offs between contiguous and indexed file allocation?
4. How do modern file systems (e.g., ext4, NTFS) combine allocation strategies?

Lecture on I/O Systems & Disk Management

1. Introduction to I/O Systems

The I/O subsystem bridges applications and physical hardware (disks, keyboards, GPUs).

Key Challenges:

- Performance gap between CPU and I/O devices.
- Device diversity (block vs. character devices).
- Error handling (bad sectors, connection failures).

---

2. I/O Hardware Architecture

(1) Device Controllers

- Purpose: Interface between OS and physical device (e.g., SATA controller for disks).
- Components:
  - Registers (status, control, data).
  - Local buffer (temporary storage, e.g., disk cache).
- Communication:
  - Memory-Mapped I/O: Device registers mapped to memory addresses.
  - Port-Mapped I/O: Special CPU instructions (e.g., `in/out` in x86).

(2) Device Drivers

- OS kernel modules that handle device-specific commands.
- Functions:
  - Translate OS calls (e.g., `read()`) to device operations.
  - Handle interrupts (e.g., data ready, error signals).
- Example:

- c
- Copy
- Download

```
struct file_operations fops = {

    .read = my_device_read,

    .write = my_device_write,

    .open = my_device_open,

};
```

(3) I/O Techniques

Method	**Mechanism**	**Pros & Cons**

Programmed I/O	CPU polls device status (busy-wait).	❌ Wastes CPU cycles.
Interrupt-Driven	Device sends interrupt when ready.	✔ Efficient but high overhead for small transfers.
DMA (Direct Memory Access)	Offloads data transfer to DMA controller.	✔ Best for large blocks (e.g., disk I/O).

---

3. Buffering, Caching, and Spooling

(1) Buffering

- Goal: Smooth out speed mismatches (e.g., slow disk vs. fast CPU).
- Types:
  - Single Buffer: One buffer holds data (e.g., keyboard input).
  - Double Buffering: Two buffers alternate (used in graphics rendering).
  - Circular Buffer: FIFO queue for streaming data (network packets).

(2) Caching

- Goal: Store frequently accessed data in faster storage (RAM vs. disk).
- Write Policies:
  - Write-Through: Immediate write to disk (safe but slow).
  - Write-Back: Delay writes (faster but risky on crash).

(3) Spooling (Simultaneous Peripheral Operations Online)

- Goal: Queue I/O jobs (e.g., printer queue).

- Example:
    - Multiple users send print jobs → spooler serializes them.

---

4. RAID (Redundant Array of Independent Disks)

Combines multiple disks for performance, redundancy, or both.

(1) RAID Level 0 (Striping)

- Data split across disks (no redundancy).
- Pros: High performance (parallel reads/writes).
- Cons: No fault tolerance (1 disk failure → total loss).
- Use Case: Video editing, temporary data.

(2) RAID Level 1 (Mirroring)

- Exact copy on 2+ disks.
- Pros: Fault-tolerant (survives 1 disk failure).
- Cons: 50% storage overhead.
- Use Case: Critical databases.

(3) RAID Level 5 (Striping + Parity)

- Data + parity distributed across disks.
- Pros: Balances performance + redundancy (survives 1 disk failure).
- Cons: Write penalty (parity calculation).
- Use Case: Enterprise storage.

(4) RAID Level 6 (Double Parity)

- Two parity blocks per stripe.
- Pros: Survives 2 disk failures.
- Cons: Higher write penalty than RAID 5.

RAID Level	Description	Min Disks	Fault Tolerance	Storage Efficiency
RAID 0	Striping	2	❌ None	100%
RAID 1	Mirroring	2	✔ 1 disk	50%
RAID 5	Striping + Parity	3	✔ 1 disk	(N-1)/N
RAID 6	Double Parity	4	✔ 2 disks	(N-2)/N

(5) Nested RAID (e.g., RAID 10)

- RAID 1+0: Mirroring + striping (high performance + redundancy).
- Survives 1 disk failure per mirror set.

---

5. Disk Management

(1) Bad Sector Handling

- ECC (Error-Correcting Codes): Detect/correct minor errors.
- Sector Remapping: Redirect writes to spare sectors.

(2) Swap Space Management

- Virtual memory extension on disk (Linux: `swap partition`, Windows: `pagefile.sys`).
- Placement Strategies:
    - Separate disk (better performance).
    - File-based swap (flexible but slower).

(3) Storage Area Networks (SANs)

- High-speed network for block-level storage access.
- Protocols: iSCSI, Fibre Channel.

---

6. Summary

Topic	**Key Takeaways**
I/O Hardware	Device controllers interface with drivers; DMA offloads CPU.
Buffering/Caching	Buffers smooth I/O speed gaps; caches reduce disk access.
RAID	RAID 0 (speed), RAID 1 (safety), RAID 5/6 (balance).

Disk Management	Bad sector remapping, swap space, SANs optimize storage.

7. Discussion Questions

1. Why is DMA preferred over interrupt-driven I/O for disk transfers?
2. How does RAID 5 tolerate a single disk failure?
3. What are the trade-offs between write-through and write-back caching?
4. When would you choose RAID 10 over RAID 5?