Operating Systems: Resource Management

Week 1-12

SDB

Autumn 2025

Week 1

Agenda

1. Why Memory Management?
2. Basic Memory Terminology
3. OS Memory Management Goals
4. Address Binding Techniques
5. Memory Protection Mechanisms
6. OS View vs. Process View
7. Summary & Key Takeaways
8. Next Week's Preview

Why Memory Management Matters

- **Limited, Fast Resource:** Main memory (RAM) is a finite and crucial resource that must be shared efficiently among multiple processes.
- **Contiguous Abstraction:** A program expects to run in a continuous, uninterrupted block of memory, even though the physical memory might be fragmented.
- **Safety & Security:** The OS must ensure that processes can only access their own memory space, preventing interference and unauthorized access.
- **Performance & Stability:** Effective memory management minimizes overhead, prevents crashes, and enables a responsive system.

Why Memory Management Matters

- **Limited, Fast Resource:** Main memory (RAM) is a finite and crucial resource that must be shared efficiently among multiple processes.
- **Contiguous Abstraction:** A program expects to run in a continuous, uninterrupted block of memory, even though the physical memory might be fragmented.
- **Safety & Security:** The OS must ensure that processes can only access their own memory space, preventing interference and unauthorized access.
- **Performance & Stability:** Effective memory management minimizes overhead, prevents crashes, and enables a responsive system.

Analogy

Think of memory as a large library. The OS is the librarian, responsible for organizing the shelves (physical memory) and making sure each reader (process) can only access their assigned books (memory).

Basic Memory Terminology

- **Logical Address:** The address generated by the CPU. This is the address that a program "sees" and works with.
- **Physical Address:** The actual address in the main memory (RAM). This is the real location where data is stored.
- **Address Space:** The set of all logical addresses that a program can reference.
- **Memory Management Unit (MMU):** A hardware device that translates logical addresses into physical addresses during program execution.
- **Binding:** The process of mapping a logical address to a physical address. This can happen at different times.

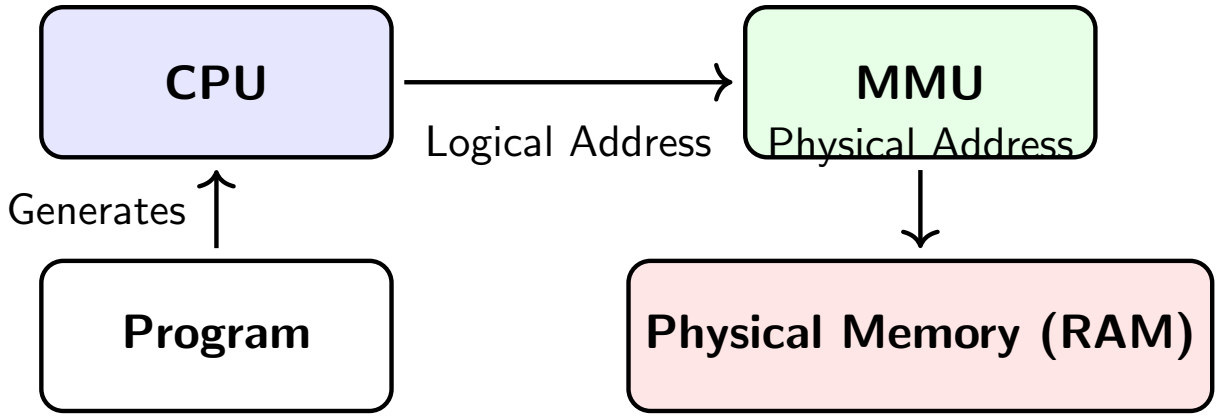
Address Binding Techniques

- **Compile-Time:** If the program's physical address is known in advance, the compiler can generate **absolute code**. This is inflexible as the program cannot be relocated.
- **Load-Time:** If the physical address is not known at compile time, the compiler generates **relocatable code**. A linker or loader then binds the addresses when the program is loaded into memory.
- **Execution-Time (Run-Time):** Addresses are bound during program execution using a special hardware component, the **MMU**. This is the most flexible method, allowing a process to be moved during execution. This is the foundation of modern memory management.

Memory Protection

- **Goal:** A process must be isolated to prevent it from accessing another process's memory or the OS's memory space.
- **Base and Limit Registers:** A simple hardware mechanism where the **base register** holds the starting physical address and the **limit register** holds the size of the process's memory block. The MMU checks every logical address to ensure it is within this valid range.
- **Advanced Mechanisms:** Modern operating systems use more complex methods like **page tables** and **access control bits** to enforce protection at a granular level.

Address Translation Diagram



OS View vs. Process View of Memory

- **Process View:** Each process sees a dedicated, linear, and contiguous address space starting from address 0. This is an illusion provided by the OS.
- **OS View:** The OS sees a collection of scattered and fragmented memory blocks, each belonging to a different process. The OS's job is to manage these non-contiguous physical locations while maintaining the contiguous illusion for each process.

OS View vs. Process View of Memory

- **Process View:** Each process sees a dedicated, linear, and contiguous address space starting from address 0. This is an illusion provided by the OS.
- **OS View:** The OS sees a collection of scattered and fragmented memory blocks, each belonging to a different process. The OS's job is to manage these non-contiguous physical locations while maintaining the contiguous illusion for each process.

Key Concept

Modern OSes use **paging** or **segmentation** to simulate an isolated, contiguous address space for each process, even if their physical memory is fragmented. This is the foundation of virtual memory.

Key Takeaways

- Memory management is essential for **multitasking, protection, and performance**.
- The **MMU** is a hardware component that provides the crucial abstraction of address translation.
- The distinction between a **logical address** (what the CPU sees) and a **physical address** (the real location in RAM) is fundamental.
- **Execution-time binding** is the most flexible approach and is used in all modern operating systems.

Next Week Preview: Allocation Strategies

This week, we covered the fundamental concepts of memory. Next week, we will dive into the practical implementation details.

- **Contiguous Memory Allocation:** The simplest method, which often leads to fragmentation.
- **Paging & Segmentation:** Advanced non-contiguous allocation schemes that overcome the limitations of contiguous allocation.
- **Fragmentation:** We will analyze internal and external fragmentation and discuss their trade-offs.

Appendix: Quiz, Exercises, and Advanced Topics

Quiz & Exercises

1. What is the primary role of the MMU?
2. Why is execution-time binding considered superior to compile-time binding for modern OSes?
3. **Exercise:** A process has a base register value of 2000 and a limit register of 500. Which of the following logical addresses are valid: 100, 450, 500, 550?

Advanced Topics to Explore

- **Translation Lookaside Buffer (TLB):** A specialized cache for the MMU to speed up address translation.
- **Non-Uniform Memory Access (NUMA):** Architectures where a processor's memory access time depends on the memory's location.
- **Memory Swapping and Paging:** How the OS manages moving processes between RAM and secondary storage.

Week 2

Agenda

1. Contiguous Memory Allocation
2. Fragmentation: The Problem
3. Paging: The Solution to External Fragmentation
4. Segmentation: The Logical View
5. Comparison & Hybrid Architectures

Contiguous Memory Allocation

- The entire process is loaded into a **single, continuous block of physical memory**.
- This approach is simple to implement but inflexible.
- The OS keeps track of which memory blocks are free and which are occupied.
- **Allocation Strategies** for finding a free block:
  - **First-Fit:** Allocate the first hole that is big enough. Fast but can leave small, useless fragments.
  - **Best-Fit:** Allocate the smallest hole that is big enough. Tries to minimize wasted space but can be slower and still create many tiny holes.
  - **Worst-Fit:** Allocate the largest hole available. Aims to leave a large hole behind, which might be useful for a later large process.

Drawback

This method inevitably leads to fragmentation, which we'll discuss next.

Fragmentation

- **Internal Fragmentation:** Wasted space *inside* an allocated memory block.
  - Occurs when a process is assigned a block of memory larger than it needs.
  - The unused space is unusable by other processes.
  - This is a common issue with **paging**.
- **External Fragmentation:** Wasted space *between* allocated blocks of memory.
  - Occurs when there is enough total memory for a new process, but it's not contiguous.
  - This is a major issue with **contiguous allocation**.
  - Can be solved with **compaction**, but this is a very costly operation.

Solution:

Paging and segmentation are designed to solve these fragmentation problems by not requiring contiguous physical memory.

Paging: The Solution to External Fragmentation

- Paging is a non-contiguous memory allocation technique.
- The OS divides the **physical memory** into fixed-size blocks called **frames**.
- The program's **logical memory** is divided into same-sized blocks called **pages**.
- A **page table** is used to map a process's pages to available frames in physical memory.

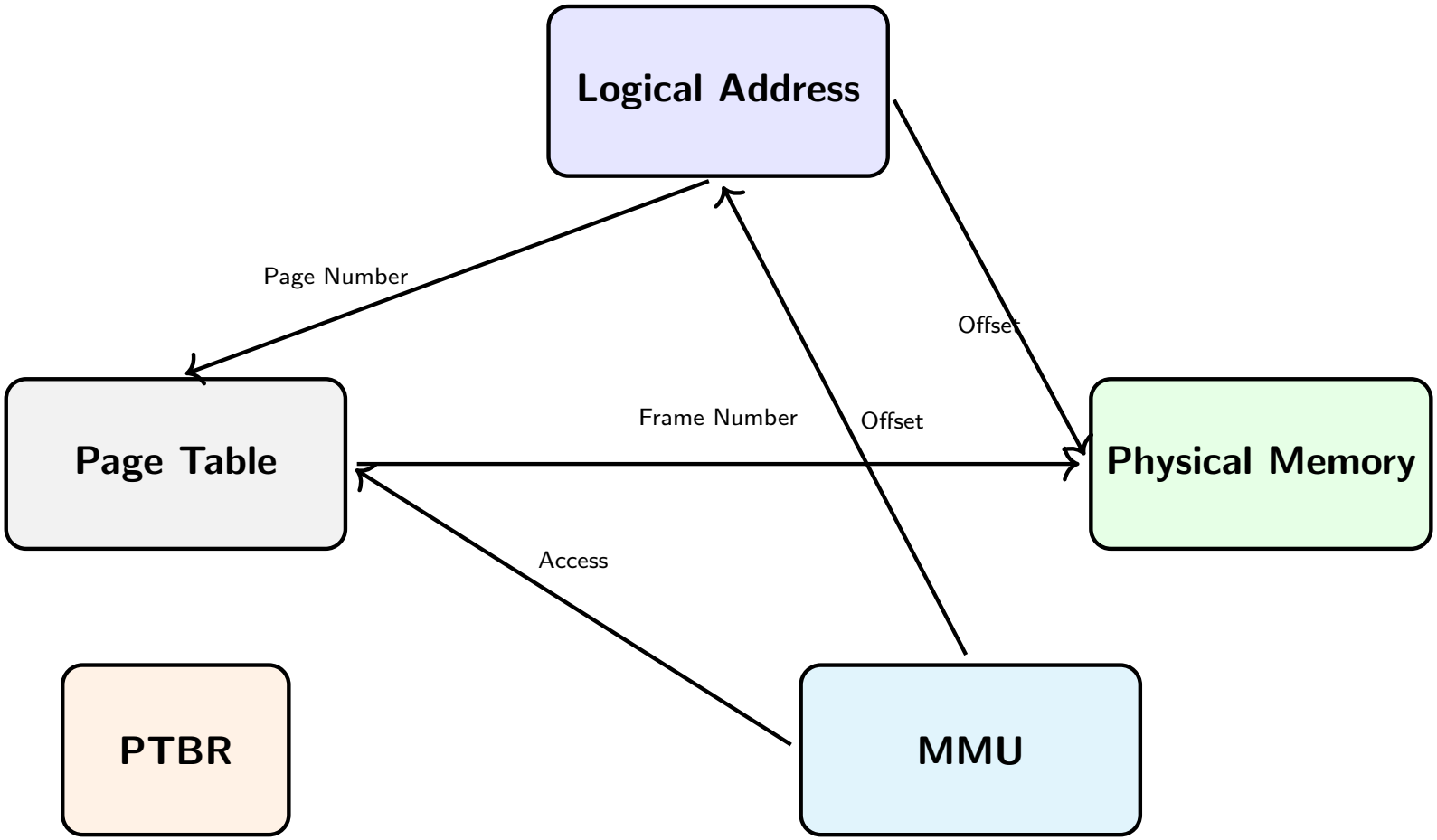
Paging: The Solution to External Fragmentation

- Paging is a non-contiguous memory allocation technique.
- The OS divides the **physical memory** into fixed-size blocks called **frames**.
- The program's **logical memory** is divided into same-sized blocks called **pages**.
- A **page table** is used to map a process's pages to available frames in physical memory.

Result:

A process's pages can be scattered throughout physical memory. The OS can allocate any free frame to any page, eliminating external fragmentation completely.

Paging Architecture



Note: The MMU combines the frame number from the page table with the original offset to form the final physical address.

Paging Hardware Support

- **Page Table Base Register (PTBR):** A dedicated CPU register that points to the start of the page table for the currently running process.
- **Translation Lookaside Buffer (TLB):** A high-speed hardware cache for page table entries.
  - It stores the most recently used logical-to-physical address translations.
  - A **TLB hit** (translation found in the cache) results in a very fast address translation.
  - A **TLB miss** (translation not found) requires the OS to perform a full page table lookup in main memory, which is much slower.

Paging Hardware Support

- **Page Table Base Register (PTBR):** A dedicated CPU register that points to the start of the page table for the currently running process.
- **Translation Lookaside Buffer (TLB):** A high-speed hardware cache for page table entries.
  - It stores the most recently used logical-to-physical address translations.
  - A **TLB hit** (translation found in the cache) results in a very fast address translation.
  - A **TLB miss** (translation not found) requires the OS to perform a full page table lookup in main memory, which is much slower.

Note

The TLB is critical for making paging practical and performant on modern systems.

Segmentation: The Logical View

- Segmentation is a memory management scheme that supports the programmer's view of memory.
- A program is divided into logical units called **segments**, which can vary in size. Common segments include:
  - **Code Segment:** The program's instructions.
  - **Data Segment:** Global variables.
  - **Stack Segment:** The call stack.
  - **Heap Segment:** Dynamically allocated memory.
- Each logical address consists of a ‘(segment number, offset)‘.

Segmentation: The Logical View

- Segmentation is a memory management scheme that supports the programmer's view of memory.
- A program is divided into logical units called **segments**, which can vary in size. Common segments include:
  - **Code Segment:** The program's instructions.
  - **Data Segment:** Global variables.
  - **Stack Segment:** The call stack.
  - **Heap Segment:** Dynamically allocated memory.
- Each logical address consists of a '(segment number, offset)'.

Note

While segmentation avoids the overhead of fixed-size blocks (like in paging), it does reintroduce the problem of **external fragmentation**.

Paging vs. Segmentation

Feature	**Paging**	**Segmentation**
Division Basis	Fixed-size pages	Logical units (variable size)
Fragmentation	Internal	External
OS View	Program as a collection of pages	Program as a collection of segments
Hardware Support	Page table, PTBR, TLB	Segment table, Base/Limit registers
User View	Not visible to the programmer	Visible to the programmer

Summary & Hybrid Systems

- **Contiguous allocation** is simple but suffers from external fragmentation.
- **Paging** eliminates external fragmentation by using fixed-size blocks but can suffer from internal fragmentation.
- **Segmentation** aligns with the programmer's logical view of memory but reintroduces external fragmentation.
- Most modern OSes use a **hybrid approach** known as **segmented paging** or **paged segmentation** to combine the benefits of both schemes.

Next Week Preview: Virtual Memory

Now that we understand paging, we can tackle one of the most powerful concepts in modern operating systems: virtual memory.

- **Virtual Memory Concepts:** Extending memory space beyond physical RAM.
- **Page Faults:** What happens when a process tries to access a page not in main memory.
- **Demand Paging:** A technique for loading pages only when they are needed.
- **Locality of Reference:** The principle that makes demand paging work efficiently.

Appendix: Quiz, Exercises, and Advanced Topics

Quiz & Exercises

1. Explain the difference between internal and external fragmentation. Which is a primary concern for paging, and which for contiguous allocation?
2. Describe the role of the TLB. Why is it essential for the performance of a paged system?
3. **Exercise:** A system uses paging with a 4 KB page size. A process has a logical address of 6500. What is the page number and the offset?

Advanced Topics to Explore

- **Multilevel Page Tables:** How OSes deal with large page tables to save memory.
- **Inverted Page Tables:** An alternative approach to page tables used to reduce memory overhead.
- **Segmented Paging:** A hybrid model that combines the logical view of segmentation with the non-contiguous benefits of paging.

Week 3

Agenda

1. Virtual Memory Concept
2. Address Translation & Page Faults
3. Page Fault Handling
4. Locality of Reference & Working Set Model
5. TLB Management & Optimization
6. Summary & Key Takeaways
7. Next Week's Preview

What is Virtual Memory?

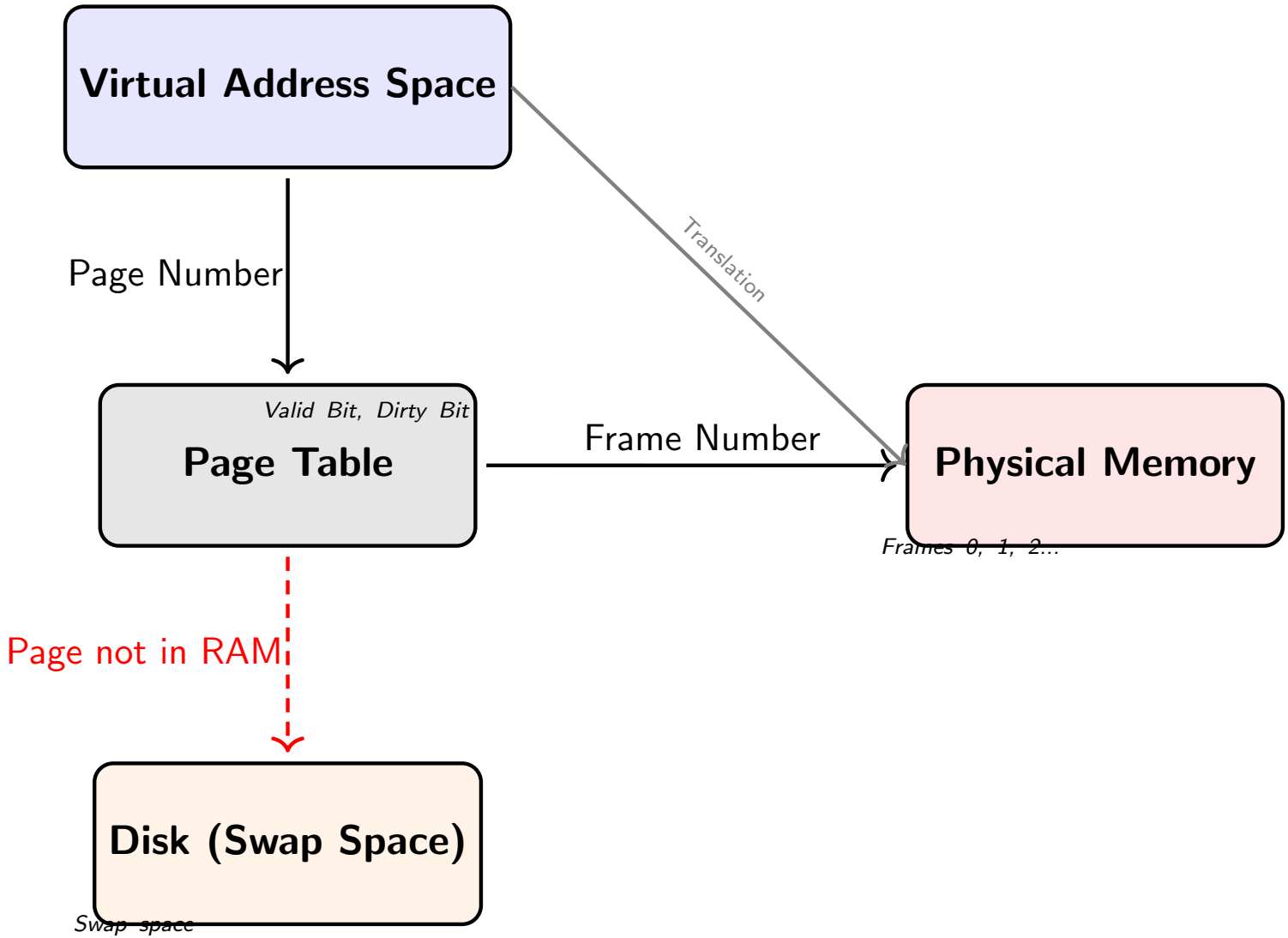
- **Decoupling Memory:** Virtual memory is a technique that separates the logical address space from the physical memory.
- **Memory Abstraction:** It creates the illusion that a process has a dedicated, large, contiguous block of memory, even if the physical memory is small and fragmented.
- **On-Demand Loading:** It allows programs to run even if they are not entirely loaded into physical memory. Only the necessary parts (pages) are loaded when a process requires them.
- **OS Role:** The operating system uses the hard disk (swap space) as an extension of RAM to hold pages that aren't currently in main memory.

What is Virtual Memory?

- **Decoupling Memory:** Virtual memory is a technique that separates the logical address space from the physical memory.
- **Memory Abstraction:** It creates the illusion that a process has a dedicated, large, contiguous block of memory, even if the physical memory is small and fragmented.
- **On-Demand Loading:** It allows programs to run even if they are not entirely loaded into physical memory. Only the necessary parts (pages) are loaded when a process requires them.
- **OS Role:** The operating system uses the hard disk (swap space) as an extension of RAM to hold pages that aren't currently in main memory.

Analogy: Think of a recipe book. You don't need to have the entire book open on the counter; you only need to look at the page for the specific recipe you're making at that moment. The rest of the book can stay on the shelf (disk) until you need it.

Virtual Memory Mapping



Page Fault Handling I

- A **page fault** is a hardware interrupt that occurs when a program tries to access a page that is marked as not present in physical memory.
- The MMU detects this and generates a **trap** to the operating system kernel.
- The OS takes over to resolve the issue transparently, so the running process is unaware of the disk I/O.

Page Fault Handling II

5. **Update Page Table:** Once the disk I/O is complete, the OS updates the page table entry to reflect the new frame number and sets the "present" bit.
6. **Restart Instruction:** The OS restarts the instruction that caused the page fault, allowing the process to continue as if no fault occurred.

Locality of Reference and Working Set

- **Locality of Reference:** A principle that states a program's memory accesses tend to cluster in small regions over time.
  - **Temporal Locality:** Recently accessed items are likely to be accessed again soon. (e.g., a loop variable)
  - **Spatial Locality:** Items near a recently accessed item are likely to be accessed soon. (e.g., accessing elements in an array)

Locality of Reference and Working Set

- **Locality of Reference:** A principle that states a program's memory accesses tend to cluster in small regions over time.
  - **Temporal Locality:** Recently accessed items are likely to be accessed again soon. (e.g., a loop variable)
  - **Spatial Locality:** Items near a recently accessed item are likely to be accessed soon. (e.g., accessing elements in an array)

The Working Set Model:

- The "working set" is the set of pages that a process is actively using at a given time.
- It is crucial for virtual memory performance. If the OS can keep a process's working set in memory, the page fault rate will be low.
- **Thrashing:** Occurs when the working set is larger than the number of available frames, leading to continuous page faults and very poor performance.

TLB Management & Optimization

- **TLB Caching:** The Translation Lookaside Buffer (TLB) is a small, fast hardware cache for page table entries. It mitigates the performance penalty of a two-step memory access in paging.
- **On TLB Hit:** The MMU finds the translation in the TLB. The physical address is generated instantly, avoiding a memory access. This is the fastest case.
- **On TLB Miss:** The MMU must consult the page table in main memory. This is slower, but still much faster than a page fault.

TLB Management & Optimization

- **TLB Caching:** The Translation Lookaside Buffer (TLB) is a small, fast hardware cache for page table entries. It mitigates the performance penalty of a two-step memory access in paging.
- **On TLB Hit:** The MMU finds the translation in the TLB. The physical address is generated instantly, avoiding a memory access. This is the fastest case.
- **On TLB Miss:** The MMU must consult the page table in main memory. This is slower, but still much faster than a page fault.

Optimization & Management:

- **TLB Shootdown:** When a page table is modified on one CPU (e.g., a process is terminated), other CPUs must be notified to invalidate their TLB entries for that process to maintain cache coherence.
- **Address Space Identifiers (ASID):** The TLB can use ASIDs to distinguish between pages from different processes, allowing TLB entries to persist across context switches.

Key Takeaways

- **Virtual Memory** provides a powerful abstraction that allows a large virtual address space with a smaller physical memory.
- A **page fault** is a hardware-supported mechanism that allows the OS to load pages from disk on-demand, making virtual memory possible.
- The concepts of **Locality of Reference** and the **Working Set** are what make virtual memory practical and efficient.
- The **TLB** is a critical hardware cache that minimizes the performance overhead of address translation, preventing constant memory lookups.

Next Week Preview: Page Replacement Algorithms

The performance of virtual memory hinges on how the OS manages its limited physical frames. Next week, we will explore this topic in detail.

- **The Challenge:** What to do when a page fault occurs and there are no free frames.
- **Page Replacement Algorithms:** Strategies for selecting a "victim" page to be evicted from memory.
- **Common Algorithms:** We will analyze FIFO, Optimal, and Least Recently Used (LRU).
- **Implementation & Performance:** We will discuss the performance trade-offs of these algorithms and explore more practical variants like the Clock Algorithm.

Appendix: Quiz, Exercises, and Advanced Topics

Quiz & Exercises

1. What is the main difference between a TLB miss and a page fault?
2. Describe a scenario where temporal locality would be high.
3. **Exercise:** A process's logical address space is 16 pages, but only 4 frames are available. Explain what happens when the process tries to access a page that is not in any of the four frames.

Advanced Topics to Explore

- **Dirty Bit and Copy-on-Write:** How the OS handles modified pages and shared memory.
- **Memory-Mapped Files:** A technique that uses virtual memory to treat a file on disk as if it were loaded into memory.
- **Segmented Paging Revisited:** How virtual memory works in a hybrid memory management system.

Week 4

Agenda

1. What is Page Replacement?
2. FIFO, Optimal, and LRU Algorithms
3. The Clock Algorithm (Second-Chance)
4. Belady's Anomaly
5. Performance Comparison & Trade-offs
6. Summary & Key Takeaways
7. Next Week's Preview

Page Replacement: The Core Challenge

- When a page fault occurs and all available physical frames are occupied, the OS must choose a "victim" page to evict.
- This evicted page is written to disk (if it was modified) and the new page is loaded into its frame.
- The primary goal of a page replacement algorithm is to minimize future page faults by making a smart choice for the victim page.

Page Replacement: The Core Challenge

- When a page fault occurs and all available physical frames are occupied, the OS must choose a "victim" page to evict.
- This evicted page is written to disk (if it was modified) and the new page is loaded into its frame.
- The primary goal of a page replacement algorithm is to minimize future page faults by making a smart choice for the victim page.

The Challenge:

The OS must make a decision about the future without having perfect knowledge of which pages will be accessed next. It must rely on heuristics.

First-In, First-Out (FIFO)

- **Policy:** Replace the page that has been in memory for the longest time.
- **Implementation:** A simple queue or linked list is used to keep track of the pages in memory. The oldest page is at the head, and new pages are added to the tail.
- **Logic:** It assumes that pages that have been in memory the longest are less likely to be used again.
- **Drawback:** FIFO is often a poor choice because it may evict a heavily used page just because it's old. This can lead to a high page fault rate.

First-In, First-Out (FIFO)

- **Policy:** Replace the page that has been in memory for the longest time.
- **Implementation:** A simple queue or linked list is used to keep track of the pages in memory. The oldest page is at the head, and new pages are added to the tail.
- **Logic:** It assumes that pages that have been in memory the longest are less likely to be used again.
- **Drawback:** FIFO is often a poor choice because it may evict a heavily used page just because it's old. This can lead to a high page fault rate.

Alert:

A **key flaw of FIFO** is that it is susceptible to **Belady's Anomaly**—a counter-intuitive phenomenon where increasing the number of available frames can actually lead to **more** page faults.

Optimal Page Replacement (Theoretical Best)

- **Policy:** Replace the page that will **not be used for the longest period of time** in the future.
- **Implementation:** This algorithm is impossible to implement in a real operating system because it requires perfect knowledge of the future reference string.
- **Purpose:** It serves as the **benchmark** for all other page replacement algorithms. The performance of any practical algorithm is measured by how close it comes to the optimal algorithm's page fault rate.

Least Recently Used (LRU)

- **Policy:** Replace the page that has not been used for the longest period of time.
- **Heuristic:** It operates on the principle of **temporal locality**: pages that were used recently will likely be used again soon.
- **Effectiveness:** LRU is a very good approximation of the Optimal algorithm, as it uses past behavior to predict the future.

Least Recently Used (LRU)

- **Policy:** Replace the page that has not been used for the longest period of time.
- **Heuristic:** It operates on the principle of **temporal locality**: pages that were used recently will likely be used again soon.
- **Effectiveness:** LRU is a very good approximation of the Optimal algorithm, as it uses past behavior to predict the future.

Implementation Challenges:

- **Counter-based:** Each page table entry has a counter. On every memory access, the counter for that page is updated. This requires costly hardware support.
- **Stack-based:** A stack is maintained where the most recently used page is at the top. On a page access, the page is moved to the top. This is also computationally expensive.

Note:

These high overhead costs make a pure LRU implementation expensive for an OS.

The Clock Algorithm (Second-Chance)

- **Policy:** A more practical and efficient approximation of LRU. It gives a page a "second chance" before evicting it.
- **Mechanism:** Uses a circular list of pages in memory and a "use" or "reference" bit for each page. A clock hand (pointer) scans the list.
- **Operation:**
  1. When a page needs to be replaced, the hand advances.
  2. If the page's reference bit is 1, the bit is set to 0 and the hand moves to the next page (the "second chance").
  3. If the reference bit is already 0, the page is evicted.

The Clock Algorithm (Second-Chance)

- **Policy:** A more practical and efficient approximation of LRU. It gives a page a "second chance" before evicting it.
- **Mechanism:** Uses a circular list of pages in memory and a "use" or "reference" bit for each page. A clock hand (pointer) scans the list.
- **Operation:**
  1. When a page needs to be replaced, the hand advances.
  2. If the page's reference bit is 1, the bit is set to 0 and the hand moves to the next page (the "second chance").
  3. If the reference bit is already 0, the page is evicted.

Advantage:

The Clock algorithm provides a good balance between performance and overhead, as it doesn't require complex data structures or expensive hardware support.

Belady's Anomaly

- **Definition:** Belady's Anomaly is a phenomenon where increasing the number of available physical frames can lead to an increase in the number of page faults.
- **Relevance:** This anomaly exists in some algorithms, most famously FIFO, but does not occur in LRU or Optimal algorithms. This highlights a key problem with a replacement policy that ignores a page's recent usage.

Belady's Anomaly

- **Definition:** Belady's Anomaly is a phenomenon where increasing the number of available physical frames can lead to an increase in the number of page faults.
- **Relevance:** This anomaly exists in some algorithms, most famously FIFO, but does not occur in LRU or Optimal algorithms. This highlights a key problem with a replacement policy that ignores a page's recent usage.

Example Trace (FIFO):

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

Frames	Memory State	Page Faults
3	[1, 2, 3], [4, 2, 3], [4, 5, 3]...	**9**
4	[1, 2, 3, 4], [5, 2, 3, 4], [5, 1, 3, 4]...	**10**

As the number of frames increases from 3 to 4, the number of page faults increases from 9 to 10.

Comparison & Trade-offs

Algorithm	Fault Rate	Implementation	Overhead	Anomaly Risk
Optimal	Lowest	Impossible	N/A	No
LRU	Low	Complex	High	No
FIFO	High	Very Simple	Low	Yes
Clock	Medium/Low	Simple	Low	No

Conclusion:

The Clock algorithm is often a preferred choice for real-world OSes because it offers a very good compromise between performance and implementation cost.

Key Takeaways

- A **page replacement algorithm** is triggered when a page fault occurs and a "victim" page must be chosen for eviction.
- The **Optimal** algorithm is the theoretical best, but is impossible to implement.
- **LRU** is a highly effective, but expensive, approximation of Optimal.
- The **Clock algorithm** is a simple and efficient approximation of LRU that is widely used in real systems.
- **Belady's Anomaly** is a key flaw of algorithms like FIFO, demonstrating that more memory doesn't always lead to better performance.

Next Week Preview: Demand Paging and Thrashing

Building on our understanding of virtual memory and page replacement, next week we will explore the practicalities of these systems.

- **Demand Paging:** The concept of lazy page loading and how it improves system efficiency.
- **Thrashing:** A pathological state where a system spends more time swapping pages than executing instructions.
- **Working Set Model:** A closer look at how the OS manages a process's active pages to prevent thrashing.
- **Page Fault Frequency Control:** Dynamic strategies to adjust the number of frames a process has to prevent thrashing.

Appendix: Quiz, Exercises, and Advanced Topics

Quiz & Exercises

1. Explain why the Optimal page replacement algorithm is impractical for real-world operating systems.
2. A system has 4 frames. Using the reference string ‘4, 3, 2, 1, 4, 3, 5, 4, 3, 2, 1, 5‘, determine the number of page faults for both the **FIFO** and **LRU** algorithms.
3. Describe how a real OS might implement an efficient approximation of LRU without using counters or a stack.

Advanced Topics to Explore

- **Second-Chance and Enhanced Second-Chance Algorithms:** More detailed versions of the Clock algorithm that also consider the ”dirty bit.”
- **Not Recently Used (NRU) Algorithm:** A simple and efficient algorithm that uses reference and dirty bits.
- **Aging Algorithm:** A more sophisticated variant that keeps track of recent usage by shifting bits in a counter.

Week 5

Agenda

1. Demand Paging Concept
2. The Valid/Invalid Bit Mechanism
3. Performance Impact: Effective Access Time
4. Thrashing: The Problem of Over-commitment
5. Thrashing Prevention Techniques
6. Summary & Key Takeaways
7. Next Week's Preview

What is Demand Paging?

- **Lazy Loading:** Demand paging is a strategy where a page is loaded into main memory only when it is actually needed (i.e., when a process attempts to access it).
- **Partial Execution:** It allows a process to begin execution even if only a small subset of its pages are in physical memory. All other pages remain on disk.
- **Benefits:** Reduces the memory required for each process, allows for faster process startup times, and supports a higher degree of multiprogramming.

What is Demand Paging?

- **Lazy Loading:** Demand paging is a strategy where a page is loaded into main memory only when it is actually needed (i.e., when a process attempts to access it).
- **Partial Execution:** It allows a process to begin execution even if only a small subset of its pages are in physical memory. All other pages remain on disk.
- **Benefits:** Reduces the memory required for each process, allows for faster process startup times, and supports a higher degree of multiprogramming.

Trade-off

While it saves memory, the first access to a page that isn't in RAM will incur a significant performance penalty due to the page fault.

The Valid/Invalid Bit Mechanism

- Each entry in a process's page table has a dedicated **valid-invalid bit**.
- **Valid Bit (= 1):** The page is currently in physical memory. The address translation can proceed normally.
- **Invalid Bit (= 0):** The page is not in physical memory. Accessing this page will trigger a page fault trap to the OS.
- The bit can also be used to indicate an illegal address space. For example, a page beyond the logical address space of a program would also be marked as invalid.

The Valid/Invalid Bit Mechanism

- Each entry in a process's page table has a dedicated **valid-invalid bit**.
- **Valid Bit (= 1):** The page is currently in physical memory. The address translation can proceed normally.
- **Invalid Bit (= 0):** The page is not in physical memory. Accessing this page will trigger a page fault trap to the OS.
- The bit can also be used to indicate an illegal address space. For example, a page beyond the logical address space of a program would also be marked as invalid.

Mechanism

This single bit is the hardware-level mechanism that supports demand paging. It tells the MMU whether a translation is possible or if a page fault is required.

Effective Access Time (EAT)

The performance of a system using demand paging is not just about the speed of memory; it's heavily influenced by the page fault rate. We use **Effective Access Time (EAT)** to quantify this.

$$EAT = (1 - p) \times \text{Memory Access Time} + p \times \text{Page Fault Service Time}$$

where p is the page fault rate (a value between 0 and 1).

Effective Access Time (EAT)

The performance of a system using demand paging is not just about the speed of memory; it's heavily influenced by the page fault rate. We use **Effective Access Time (EAT)** to quantify this.

$$EAT = (1 - p) \times \text{Memory Access Time} + p \times \text{Page Fault Service Time}$$

where p is the page fault rate (a value between 0 and 1).

Example:

- Memory Access Time = **100 ns**
- Average Page Fault Service Time = **10 ms** $(10,000,000$ ns$)$
- Even with a very low page fault rate of $p = 0.001$ (1 in 1000 accesses), the EAT is calculated as:

$$EAT = (0.999 \times 100 \text{ ns}) + (0.001 \times 10,000,000 \text{ ns}) \approx 10,099 \text{ ns}$$

- This means a tiny page fault rate causes a performance degradation of over **100 times**. This highlights why keeping the page fault rate low is paramount.

What is Thrashing?

- **Definition:** Thrashing is a state where the system is spending more time servicing page faults and swapping pages in and out of memory than it is doing useful work (executing instructions).
- **Cause:** It occurs when the total working sets of all active processes exceed the total amount of available physical memory.
- **Symptoms:** The CPU utilization drops significantly, as it is constantly waiting for disk I/O, while the page fault rate and disk activity are extremely high.

What is Thrashing?

- **Definition:** Thrashing is a state where the system is spending more time servicing page faults and swapping pages in and out of memory than it is doing useful work (executing instructions).
- **Cause:** It occurs when the total working sets of all active processes exceed the total amount of available physical memory.
- **Symptoms:** The CPU utilization drops significantly, as it is constantly waiting for disk I/O, while the page fault rate and disk activity are extremely high.

Analogy:

Imagine a group of people trying to work on a single small desk. They constantly have to put their current work away in a file cabinet and retrieve new work, causing more time to be spent on shuffling paper than on the actual tasks.

Thrashing Prevention Techniques

The goal is to provide enough memory to each process to hold its working set, thus preventing excessive page faults.

- **Working-Set Model:**
  - The OS monitors a process's **working set**—the set of pages it has recently referenced.
  - The OS ensures that each process has enough physical frames to hold its entire working set.
  - If the sum of all working sets exceeds available memory, the OS may **suspend** one or more processes to free up frames and prevent thrashing.
- **Page Fault Frequency (PFF) Control:**
  - The OS sets a desired upper and lower bound on the page fault rate.
  - If a process's fault rate is too high (above the upper bound), the OS assumes it needs more frames and allocates one.
  - If a process's fault rate is too low (below the lower bound), the OS may take away one of its frames.
  - This is an adaptive approach that dynamically adjusts memory allocation to match demand.

Key Takeaways

- **Demand Paging** provides a powerful abstraction but introduces the risk of page faults.
- The **Valid/Invalid Bit** is the hardware component that makes demand paging possible.
- The performance of a demand-paged system is incredibly sensitive to the page fault rate, as seen in the **Effective Access Time (EAT)** formula.
- **Thrashing** is a dangerous state where system performance collapses due to excessive page swapping.
- The **Working-Set Model** and **PFF Control** are key strategies used by the OS to prevent thrashing and maintain system stability.

Next Week Preview: File System Architecture

So far, we've focused on memory management. Next week, we begin our exploration of file systems—how data is stored and organized on secondary storage.

- **File System Design Goals:** Reliability, performance, and security.
- **Core Components:** Superblocks, inodes, and directories.
- **File Allocation Methods:** How files are laid out on the disk (e.g., contiguous, linked, indexed).

Appendix: Quiz, Exercises, and Advanced Topics

Quiz & Exercises

1. Why is a page fault service time so much longer than a typical memory access time?
2. Explain how the **Working Set Model** can be used to prevent thrashing.
3. **Exercise:** A computer has a memory access time of 50 ns. The average page fault service time is 5 ms. What is the maximum acceptable page fault rate to ensure that the Effective Access Time does not exceed 100 ns?

Advanced Topics to Explore

- **Copy-on-Write (CoW):** A technique that uses demand paging to optimize the ‘fork()‘ system call.
- **Prepaging:** A strategy to proactively load pages that a process will likely need in the future.
- **Shared Pages:** How demand paging facilitates sharing code segments (like libraries) among multiple processes.

Week 6

Agenda

1. File System Objectives and Abstractions
2. File Attributes and Metadata
3. File Control Blocks (FCBs) and Inodes
4. Directory Structures
5. File System Layout on Storage
6. Summary & Key Takeaways
7. Next Week's Preview

What is a File System?

- A **file system** is the component of the operating system that manages and organizes files on a storage device (like a disk or SSD).
- It provides a logical abstraction, allowing users and applications to interact with data using easy-to-remember names and a hierarchical structure, rather than raw physical addresses.
- Key responsibilities include:
  - **Persistence:** Ensuring data survives a system reboot.
  - **Access Control:** Managing permissions and security.
  - **Structure:** Organizing files in a logical hierarchy.

What is a File System?

- A **file system** is the component of the operating system that manages and organizes files on a storage device (like a disk or SSD).
- It provides a logical abstraction, allowing users and applications to interact with data using easy-to-remember names and a hierarchical structure, rather than raw physical addresses.
- Key responsibilities include:
  - **Persistence:** Ensuring data survives a system reboot.
  - **Access Control:** Managing permissions and security.
  - **Structure:** Organizing files in a logical hierarchy.

Analogy:

Think of a file system as the librarian for your hard drive. It maintains the card catalog (directory structure), tracks each book's details (metadata), and knows exactly where each book is located on the shelves (physical disk blocks).

File Attributes and Metadata

- **File Attributes** are the descriptive properties of a file, separate from its actual content. This information is a file's **metadata**.
- Key attributes typically include:
  - **Name:** The symbolic name for the file.
  - **Type:** The file's format (e.g., text, executable, image).
  - **Size:** The number of bytes in the file.
  - **Timestamps:** Dates of creation, last modification, and last access.
  - **Permissions:** Read, Write, and Execute bits for the owner, group, and others.
  - **Owner and Group Information:** The user and group IDs associated with the file.

File Attributes and Metadata

- **File Attributes** are the descriptive properties of a file, separate from its actual content. This information is a file's **metadata**.
- Key attributes typically include:
  - **Name:** The symbolic name for the file.
  - **Type:** The file's format (e.g., text, executable, image).
  - **Size:** The number of bytes in the file.
  - **Timestamps:** Dates of creation, last modification, and last access.
  - **Permissions:** Read, Write, and Execute bits for the owner, group, and others.
  - **Owner and Group Information:** The user and group IDs associated with the file.

ssignificance:

This metadata allows the OS to manage and protect files without needing to read their content.

File Control Block (FCB) & Inodes

- A **File Control Block (FCB)** is a data structure used by the OS to store the metadata for a single file. Every file has an FCB.
- In UNIX-like systems, the FCB is a special structure called an **inode** (index node).
- An inode contains:
  - **File Attributes:** All the metadata we just discussed (size, permissions, timestamps, owner, etc.).
  - **Disk Block Pointers:** A list of pointers to the physical data blocks on the disk where the file's content is stored.

File Control Block (FCB) & Inodes

- A **File Control Block (FCB)** is a data structure used by the OS to store the metadata for a single file. Every file has an FCB.
- In UNIX-like systems, the FCB is a special structure called an **inode** (index node).
- An inode contains:
  - **File Attributes:** All the metadata we just discussed (size, permissions, timestamps, owner, etc.).
  - **Disk Block Pointers:** A list of pointers to the physical data blocks on the disk where the file's content is stored.

Key Concept

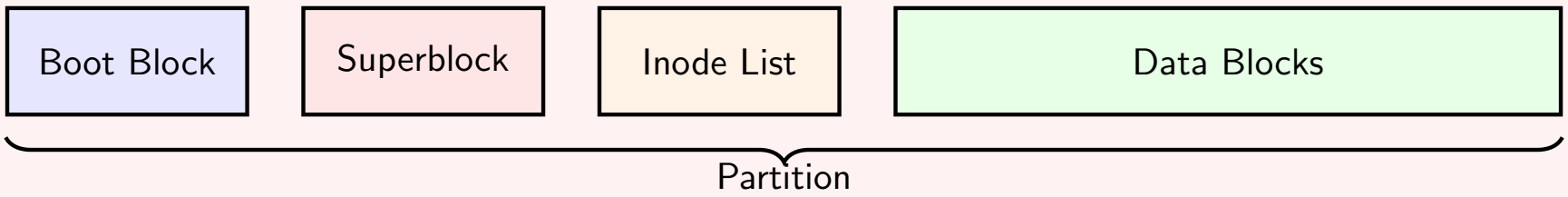
A file's name is stored in a directory entry, while all other metadata and the pointers to its data are stored in its inode. This separation is crucial for flexible file systems.

Directory Structures

- A **directory** is a file that contains a list of other files and/or directories. It provides a mapping between a file's name and its FCB/inode number.
- Common directory structures include:
  - **Single-Level Directory:** All files are in a single, simple directory. Not scalable.
  - **Two-Level Directory:** A root directory contains user directories, and each user has their own personal directory.
  - **Tree-Structured Directory:** The most common structure, allowing for a hierarchical organization of files and directories (e.g., Linux, Windows).
  - **Acyclic Graph Directory:** A generalization of the tree structure that allows files to be shared via links (shortcuts/symlinks) without creating loops.

File System Layout on Disk

File System Layout (Example)



- **Boot Block:** Contains the bootloader program to start the OS.
- **Superblock:** Stores critical metadata about the entire file system (file system type, size, block counts, location of the inode list, etc.).
- **Inode List:** A contiguous block on disk that stores all the inodes for every file on the partition.
- **Data Blocks:** The main area where the actual content of files is stored.

Key Takeaways

- A **file system** is a fundamental OS component that provides a logical abstraction over raw disk storage.
- File **metadata** is stored in a data structure called a **File Control Block (FCB)**, or an **inode** in UNIX-like systems.
- The **directory structure** maps human-readable names to FCBs/inodes, enabling a hierarchical file organization.
- The physical layout on disk is carefully structured with a **superblock**, **inode list**, and **data blocks** to ensure a file system is self-contained and manageable.

Next Week Preview: File Allocation and Free Space Management

With our understanding of file system architecture, we can now address a critical question: how does the OS store the content of a file on the disk?

- **File Allocation Methods:** We will examine the three main strategies for storing files on disk:
  - Contiguous Allocation
  - Linked Allocation
  - Indexed Allocation
- **Free Space Management:** We will also look at how the file system keeps track of which blocks are free and available for new files.

Appendix: Quiz, Exercises, and Advanced Topics

Quiz & Exercises

1. What is the primary advantage of storing a file's name in a directory entry and its metadata in a separate inode?
2. In a UNIX-like file system, where would you find the information about a file's size and permissions?
3. **Exercise:** Draw a simple directory tree showing the '/usr', '/usr/bin', and '/etc' directories, and a file named 'passwd' in '/etc'. Show the links between them.

Advanced Topics to Explore

- **Journaling File Systems:** A method for preventing data corruption in the event of a system crash.
- **Virtual File System (VFS):** A software layer in the kernel that provides a uniform interface to different file system types.
- **RAID (Redundant Array of Independent Disks):** Techniques for improving performance and reliability of disk systems.

Week 7

Agenda

1. The File Allocation Problem
2. Contiguous, Linked, and Indexed Allocation
3. Free Space Management Techniques
4. Block Size and Performance Trade-offs
5. Comparative Analysis
6. Summary & Key Takeaways
7. Next Week's Preview

File Allocation: The Problem

- **Objective:** The file system must efficiently map the logical structure of a file onto the physical storage blocks on a disk.
- **Key Challenges:**
  - **Storage Utilization:** Minimizing wasted space (fragmentation).
  - **Access Performance:** Enabling fast sequential and random access to file data.
  - **Flexibility:** Allowing files to grow or shrink dynamically.
  - **Metadata Overhead:** Keeping track of block pointers without excessive overhead.

File Allocation: The Problem

- **Objective:** The file system must efficiently map the logical structure of a file onto the physical storage blocks on a disk.
- **Key Challenges:**
  - **Storage Utilization:** Minimizing wasted space (fragmentation).
  - **Access Performance:** Enabling fast sequential and random access to file data.
  - **Flexibility:** Allowing files to grow or shrink dynamically.
  - **Metadata Overhead:** Keeping track of block pointers without excessive overhead.

Each of the primary allocation methods we will discuss represents a different trade-off in solving these challenges.

Contiguous Allocation

- **Principle:** Each file is stored in a **single, contiguous set** of blocks on the disk.
- **How it works:** A directory entry only needs to store the starting block address and the length of the file.
- **Performance:**
  - **Excellent Sequential Access:** Reading a file is a single, fast I/O operation.
  - **Excellent Random Access:** The address of any block can be calculated instantly (‘start_block + offset‘).

Contiguous Allocation

- **Principle:** Each file is stored in a **single, contiguous set** of blocks on the disk.
- **How it works:** A directory entry only needs to store the starting block address and the length of the file.
- **Performance:**
  - **Excellent Sequential Access:** Reading a file is a single, fast I/O operation.
  - **Excellent Random Access:** The address of any block can be calculated instantly ('start_block + offset').

Drawbacks

- **External Fragmentation:** As files are created and deleted, the free space becomes a collection of non-contiguous fragments.
- **Inflexibility:** The file size must be known at creation time, and files cannot easily grow or shrink.

Linked Allocation

- **Principle:** Each file is a **linked list of disk blocks**. The blocks can be scattered anywhere on the disk.
- **How it works:** Each block contains a pointer to the next block in the file. A directory entry only stores the starting block and the last block address.
- **Benefits:**
  - **No External Fragmentation:** Any free block can be used.
  - **Flexible File Size:** Files can grow dynamically by simply linking a new block to the end.

Linked Allocation

- **Principle:** Each file is a **linked list of disk blocks**. The blocks can be scattered anywhere on the disk.
- **How it works:** Each block contains a pointer to the next block in the file. A directory entry only stores the starting block and the last block address.
- **Benefits:**
  - **No External Fragmentation:** Any free block can be used.
  - **Flexible File Size:** Files can grow dynamically by simply linking a new block to the end.

Drawbacks

- **Poor Random Access:** To find a specific block, the OS must traverse the linked list from the beginning.
- **Metadata Overhead:** A portion of each block is dedicated to storing the pointer, reducing the space available for data.

Indexed Allocation

- **Principle:** All block pointers for a file are collected into a single location called an **index block**.
- **How it works:** The directory entry points to the index block. To access any data block, the OS first reads the index block and then follows the appropriate pointer.
- **Benefits:**
  - **Excellent Random Access:** Any block can be accessed directly from the index block.
  - **No External Fragmentation:** Blocks can be scattered anywhere.

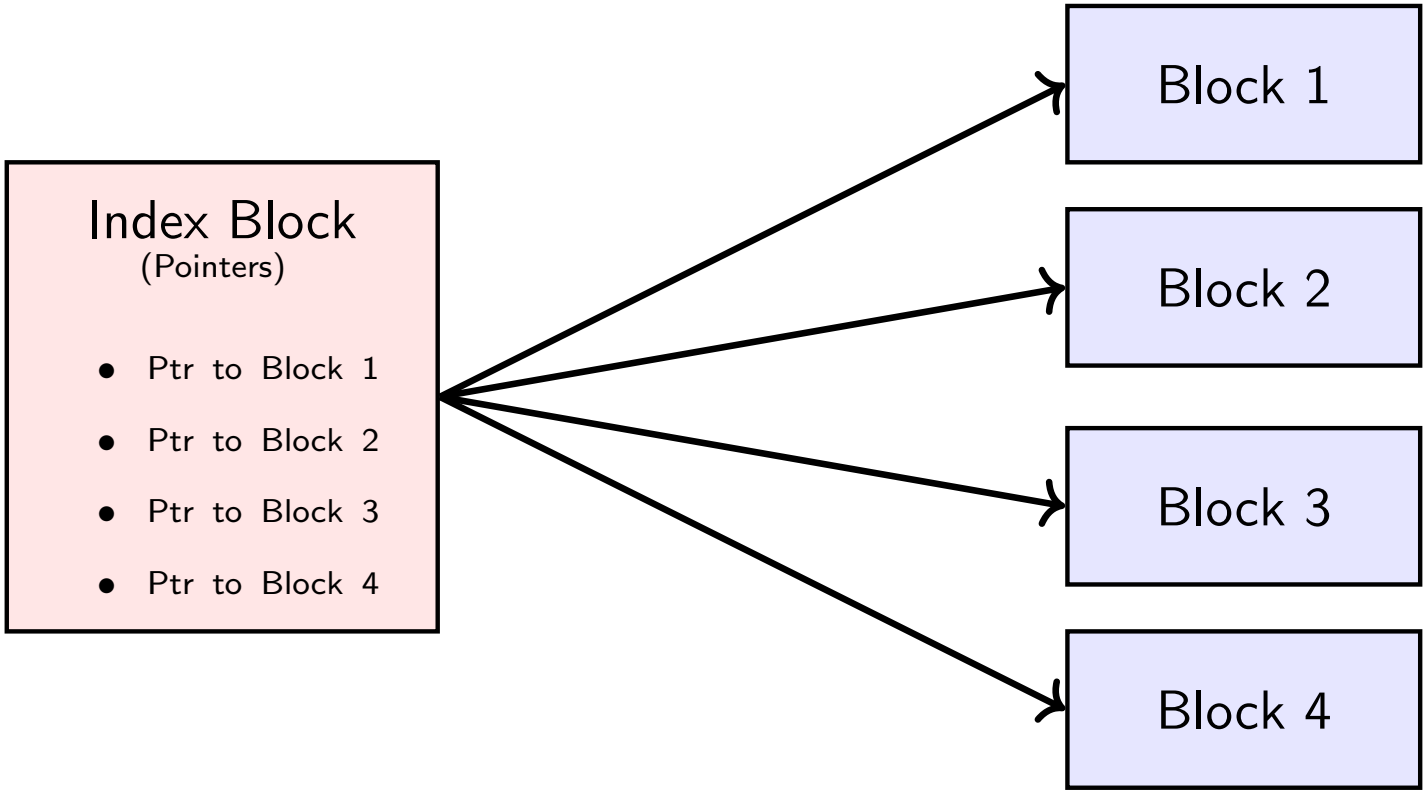
Indexed Allocation

- **Principle:** All block pointers for a file are collected into a single location called an **index block**.
- **How it works:** The directory entry points to the index block. To access any data block, the OS first reads the index block and then follows the appropriate pointer.
- **Benefits:**
  - **Excellent Random Access:** Any block can be accessed directly from the index block.
  - **No External Fragmentation:** Blocks can be scattered anywhere.

Drawbacks

- Overhead of the index block itself.
- The file size is limited by the size of the index block. Modern systems use multi-level indexing to solve this.

Indexed Allocation Diagram



Note:

The index block stores all the pointers to the file's data blocks, enabling direct access to any block.

Free Space Management

The file system must keep a record of all unallocated disk blocks to be able to create new files.

- **Bitmap (Bit Vector):**
  - A bit array where each bit represents a disk block.
  - ‘1‘ = allocated, ‘0‘ = free.
  - Fast to find a free block but requires a large contiguous chunk of memory.
- **Linked List:**
  - All free blocks are linked together in a list.
  - Simple to implement, but finding a specific number of free blocks can be slow.
- **Grouping and Counting:**
  - Store the addresses of free blocks in a group. When the group is exhausted, the last block in the group points to the next group of free blocks.
  - A counter can also be used to store a range of contiguous free blocks.

Block Size and Performance Trade-offs

The choice of block size is a crucial design decision for a file system.

- **Larger Block Size:**
  - **Advantages:** Fewer disk accesses and less metadata overhead for large files. Better performance for sequential access.
  - **Disadvantages:** More internal fragmentation, as the last block of a file is often only partially used.
- **Smaller Block Size:**
  - **Advantages:** Less internal fragmentation, saving space for smaller files.
  - **Disadvantages:** More disk accesses for large files and higher overhead for storing pointers (e.g., a larger inode table).

Block Size and Performance Trade-offs

The choice of block size is a crucial design decision for a file system.

- **Larger Block Size:**
  - **Advantages:** Fewer disk accesses and less metadata overhead for large files. Better performance for sequential access.
  - **Disadvantages:** More internal fragmentation, as the last block of a file is often only partially used.
- **Smaller Block Size:**
  - **Advantages:** Less internal fragmentation, saving space for smaller files.
  - **Disadvantages:** More disk accesses for large files and higher overhead for storing pointers (e.g., a larger inode table).

The ideal block size is a balance between these factors, typically chosen to be between 4 KB and 16 KB for modern systems, to handle both small and large files efficiently.

File Allocation Strategy Comparison

Method	Sequential Access	Random Access	Fragmentation	
Contiguous	Excellent	Excellent	External	
Linked	Fair	Poor	None	
Indexed	Good	Excellent	None	

Key Takeaways

- **File Allocation** is the process of mapping a file to physical disk blocks, and each method involves a different set of trade-offs.
- **Contiguous** allocation is fast but suffers from external fragmentation and is inflexible.
- **Linked** allocation is flexible but slow for random access.
- **Indexed** allocation, used in most modern file systems, provides a balance of performance, flexibility, and good random access.
- **Free Space Management** is a core file system task, often handled with a bitmap or a linked list of free blocks.

Next Week Preview: Disk Scheduling and I/O Performance

Now that we understand how files are stored, we will explore how the OS optimizes the physical movement of the disk's read/write head.

- **Disk Structure:** Platters, tracks, and sectors.
- **Disk Access Delay:** The components of latency (seek time, rotational latency).
- **Disk Scheduling Algorithms:** FCFS, SSTF, SCAN, and LOOK.
- **RAID Basics:** An introduction to using multiple disks to improve performance and reliability.

Appendix: Quiz, Exercises, and Advanced Topics

Quiz & Exercises

1. A file system uses a bitmap for free space management. The disk has 1000 blocks. How much memory is needed to store the bitmap?
2. For which type of file (e.g., a large video file vs. a small configuration file) would contiguous allocation provide the biggest performance benefit?
3. **Exercise:** A file system with 1KB blocks needs to store a 10MB file using indexed allocation. How many index blocks would be needed if each index block can hold 256 block pointers?

Advanced Topics to Explore

- **FAT (File Allocation Table):** A classic example of linked allocation with a separate table.
- **Extents-based Allocation:** A modern approach that combines aspects of both contiguous and indexed allocation to improve performance.
- **B-tree Based File Allocation:** Advanced file systems (like XFS) use B-trees for efficient storage and retrieval of file block pointers.

Week 8

Agenda

1. Disk Structure & Access Latency
2. I/O Bottlenecks & Queueing
3. Disk Scheduling Algorithms
4. Performance Comparison & Trade-offs
5. Choosing a Scheduling Strategy
6. Summary & Key Takeaways
7. Next Week's Preview

Disk Structure & Access Breakdown

- Hard Disk Drives (HDDs) are electro-mechanical devices that store data on rapidly rotating circular platters.
- Data is organized into concentric circles called **tracks**, which are divided into **sectors**. A read/write **head** moves across the platters.
- Disk access time is composed of three main parts:
  - **Seek Time:** The time it takes for the disk arm to move the read/write head to the correct track. This is typically the **dominant component** of access time.
  - **Rotational Latency:** The time it takes for the desired sector to rotate under the read/write head.
  - **Transfer Time:** The time required to actually transfer the data from the disk to memory.

Disk Structure & Access Breakdown

- Hard Disk Drives (HDDs) are electro-mechanical devices that store data on rapidly rotating circular platters.
- Data is organized into concentric circles called **tracks**, which are divided into **sectors**. A read/write **head** moves across the platters.
- Disk access time is composed of three main parts:
  - **Seek Time:** The time it takes for the disk arm to move the read/write head to the correct track. This is typically the **dominant component** of access time.
  - **Rotational Latency:** The time it takes for the desired sector to rotate under the read/write head.
  - **Transfer Time:** The time required to actually transfer the data from the disk to memory.

Optimization Focus

Because seek time is the largest contributor to latency, disk scheduling algorithms primarily aim to minimize head movement.

I/O Bottlenecks & Queueing

- Due to the mechanical nature of HDDs, disk I/O operations are orders of magnitude slower than CPU operations.
- When multiple processes request disk I/O concurrently, these requests accumulate in an **I/O queue** (managed by the OS or the disk controller).
- If these requests are serviced in an unoptimized order, the disk head can move back and forth erratically (often called "head thrashing"), drastically increasing total seek time and reducing overall disk throughput.

I/O Bottlenecks & Queueing

- Due to the mechanical nature of HDDs, disk I/O operations are orders of magnitude slower than CPU operations.
- When multiple processes request disk I/O concurrently, these requests accumulate in an **I/O queue** (managed by the OS or the disk controller).
- If these requests are serviced in an unoptimized order, the disk head can move back and forth erratically (often called "head thrashing"), drastically increasing total seek time and reducing overall disk throughput.

Solution

Solution: Disk scheduling algorithms reorder the pending I/O requests in the queue to minimize the total head movement and improve response time.

First-Come, First-Served (FCFS) Scheduling

- **Principle:** Requests are serviced strictly in the order they arrive in the queue.
- **Pros:**
  - Simple to implement.
  - Inherently **fair** (no starvation).
- **Cons:** Very inefficient for disk performance if requests are scattered across the disk, leading to excessive head movement.

First-Come, First-Served (FCFS) Scheduling

- **Principle:** Requests are serviced strictly in the order they arrive in the queue.
- **Pros:**
  - Simple to implement.
  - Inherently **fair** (no starvation).
- **Cons:** Very inefficient for disk performance if requests are scattered across the disk, leading to excessive head movement.

Example:

- Current head position: **53**
- Request queue: `[98, 183, 37, 122, 14, 124, 65, 67]` (Cylinder numbers)
- **Sequence of movements:** $53 \rightarrow 98 \rightarrow 183 \rightarrow 37 \rightarrow 122 \rightarrow 14 \rightarrow 124 \rightarrow 65 \rightarrow 67$
- **Total Head Movement:** $(98 - 53) + (183 - 98) + |37 - 183| + |122 - 37| + |14 - 122| + |124 - 14| + |65 - 124| + |67 - 65|$
  $= 45 + 85 + 146 + 85 + 108 + 110 + 59 + 2 =$ **640 cylinders**

Shortest Seek Time First (SSTF)

- **Principle:** Service the request that is **closest to the current head position** next.
- **Pros:** Significantly reduces the average seek time and increases disk throughput compared to FCFS.
- **Cons:**
  - Can lead to **starvation** for requests that are far from the frequently accessed areas.
  - Requires the OS to constantly know the next closest request, adding minor overhead.

Shortest Seek Time First (SSTF)

- **Principle:** Service the request that is **closest to the current head position** next.
- **Pros:** Significantly reduces the average seek time and increases disk throughput compared to FCFS.
- **Cons:**
  - Can lead to **starvation** for requests that are far from the frequently accessed areas.
  - Requires the OS to constantly know the next closest request, adding minor overhead.

Example (using same queue):

- Current head: **53**
- Queue: `[98, 183, 37, 122, 14, 124, 65, 67]`
- **Sequence (SSTF):** $53 \rightarrow 65 \rightarrow 67 \rightarrow 37 \rightarrow 14 \rightarrow 98 \rightarrow 122 \rightarrow 124 \rightarrow 183$
- **Total Head Movement:** $(65 - 53) + (67 - 65) + |37 - 67| + |14 - 37| + |98 - 14| + |122 - 98| + |124 - 122| + |183 - 124|$
  $= 12 + 2 + 30 + 23 + 84 + 24 + 2 + 59 =$ **236 cylinders** (Much lower than FCFS!)

SCAN (Elevator Algorithm)

- **Principle:** The disk head starts at one end of the disk and moves towards the other end, servicing all requests in its path. Once it reaches the end, it reverses direction and continues servicing requests.
- **Analogy:** It mimics the behavior of an **elevator** that picks up and drops off passengers as it travels up and down a building.
- **Pros:**
  - Provides a more uniform wait time than SSTF.
  - **Eliminates starvation** for requests.
- **Cons:** Requests just bypassed (e.g., at the end of a scan) will have to wait for the entire sweep of the disk.

SCAN (Elevator Algorithm)

- **Principle:** The disk head starts at one end of the disk and moves towards the other end, servicing all requests in its path. Once it reaches the end, it reverses direction and continues servicing requests.
- **Analogy:** It mimics the behavior of an **elevator** that picks up and drops off passengers as it travels up and down a building.
- **Pros:**
  - Provides a more uniform wait time than SSTF.
  - **Eliminates starvation** for requests.
- **Cons:** Requests just bypassed (e.g., at the end of a scan) will have to wait for the entire sweep of the disk.

Example: Head at 53, moving towards 199. Queue = `[98, 183, 37, 122, 14, 124, 65, 67]`
Sequence: 53 $\rightarrow$ 65 $\rightarrow$ 67 $\rightarrow$ 98 $\rightarrow$ 122 $\rightarrow$ 124 $\rightarrow$ 183 $\rightarrow$ 199 (reaches end) $\rightarrow$ 37 $\rightarrow$ 14 $\rightarrow$ 0 (reaches other end).

LOOK Scheduling

- **Principle:** A practical variation of SCAN. Instead of sweeping all the way to the end of the disk, the head only goes as far as the **last request in the current direction**, and then reverses.
- **Pros:**
  - Avoids unnecessary travel to the very ends of the disk.
  - Reduces total head movement compared to SCAN.
  - Still provides good fairness and avoids starvation.

LOOK Scheduling

- **Principle:** A practical variation of SCAN. Instead of sweeping all the way to the end of the disk, the head only goes as far as the **last request in the current direction**, and then reverses.
- **Pros:**
  - Avoids unnecessary travel to the very ends of the disk.
  - Reduces total head movement compared to SCAN.
  - Still provides good fairness and avoids starvation.

Variants:

- **C-SCAN (Circular SCAN):** Similar to SCAN, but when the head reaches one end, it immediately returns to the other end without servicing requests on the return trip. This provides more uniform wait times for all requests (useful in heavily loaded systems).
- **C-LOOK (Circular LOOK):** The practical version of C-SCAN, where the head only goes to the last request in one direction before quickly jumping back to the last request in the opposite direction without servicing requests during the jump.

Disk Scheduling Algorithm Comparison

Algorithm	Average Seek Time	Fairness	Starvation Risk	Implementation Complexity
FCFS	Poor	High	None	Low
SSTF	Good (Lowest)	Low	Yes	Medium
SCAN	Better	Moderate	None	Medium
LOOK	Best (Practical)	Good	None	Medium

Choosing a Scheduling Strategy

The optimal disk scheduling algorithm depends heavily on the specific workload and the type of storage device.

- **Workload Characteristics:**
  - **Heavy Random Access:** Algorithms that minimize average seek time, like **SSTF** or **LOOK**, tend to perform well.
  - **Mixed Workloads + Fairness: SCAN** or **LOOK** are generally preferred as they provide a good balance of performance and equitable service.
  - **Real-time constraints:** None of these general-purpose algorithms provide strict real-time guarantees.
- **Device Type:**
  - **HDDs:** Disk scheduling is crucial due to significant mechanical seek times.
  - **SSDs (Solid State Drives):** Since SSDs have no moving parts, seek time is virtually zero. Therefore, traditional disk scheduling algorithms are largely irrelevant for SSDs; requests can often be serviced in FCFS order.

Key Takeaways

- Disk access latency is dominated by **seek time** and **rotational latency**.
- **Disk scheduling algorithms** reorder I/O requests to minimize head movement and improve overall I/O throughput.
- Algorithms like **SSTF** prioritize performance by minimizing seek time but can lead to starvation.
- **SCAN** and **LOOK** (and their circular variants) offer a better balance of performance and fairness, effectively preventing starvation.
- The choice of algorithm depends on the **workload** and the **storage device type**; for SSDs, disk scheduling is less relevant.

Next Week Preview: RAID and Modern Storage Architectures

Building on our understanding of individual disk performance, next week we'll explore how multiple disks can be combined to enhance both performance and reliability.

- **RAID (Redundant Array of Independent Disks):** Combining multiple physical disks into a single logical unit.
- **RAID Levels:** Understanding different configurations like Striping (RAID 0), Mirroring (RAID 1), and parity-based RAID levels (RAID 5, 6).
- **SSD Architecture:** A deeper dive into how Solid State Drives work and their implications for OS design.
- **Storage Trends:** Emerging storage technologies and their impact on system performance.

Appendix: Quiz, Exercises, and Advanced Topics I

Quiz & Exercises

1. Explain why disk scheduling algorithms are less critical for Solid State Drives (SSDs) compared to Hard Disk Drives (HDDs).
2. What is Belady's Anomaly, and why is it not observed in disk scheduling algorithms like SCAN or LOOK?
3. **Exercise:** A disk has 200 cylinders (0-199). The head is currently at cylinder 50. The queue of requests is: `[82, 170, 43, 140, 24, 16, 190]`. Calculate the total head movement for:
   1. FCFS
   2. SSTF
   3. SCAN (moving towards 199 first)

Advanced Topics to Explore

- **Elevator Algorithm with Shortest Seek Priority:** Combining SCAN/LOOK with elements of SSTF to optimize within a scan direction.

Appendix: Quiz, Exercises, and Advanced Topics II

- **I/O Schedulers in Linux:** Exploring specific implementations like CFQ (Completely Fair Queuing), Deadline, and NOOP.
- **Flash Translation Layer (FTL) in SSDs:** The firmware layer that manages data placement and wear leveling on SSDs.
- **NVMe (Non-Volatile Memory Express):** A new communication protocol for SSDs that reduces latency and improves parallelism.

Week 9

Agenda

1. Introduction to RAID
2. RAID Levels: 0, 1, 5, & 6
3. Striping, Mirroring, & Parity
4. SSD vs. HDD: Architecture and Access Model
5. Storage Trends and OS Implications
6. Summary & Key Takeaways
7. Next Week's Preview

What is RAID?

- **RAID** stands for **R**edundant **A**rray of **I**ndependent **D**isks.
- It is a storage technology that combines multiple physical disk drives into a single logical unit.
- The primary goals of RAID are to improve:
  - **Performance:** By distributing data across multiple drives.
  - **Reliability:** By adding redundancy to protect against drive failure.

What is RAID?

- **RAID** stands for **R**edundant **A**rray of **I**ndependent **D**isks.
- It is a storage technology that combines multiple physical disk drives into a single logical unit.
- The primary goals of RAID are to improve:
  - **Performance:** By distributing data across multiple drives.
  - **Reliability:** By adding redundancy to protect against drive failure.

The two core mechanisms used in RAID are **data striping** (for performance) and **redundancy** (for reliability, using either mirroring or parity).

RAID 0 and RAID 1

- **RAID 0 (Striping):**
  - Data is broken into stripes and written across all disks in the array.
  - This provides a significant boost in performance, as read/write operations can happen concurrently on multiple disks.
  - **Drawback:** It offers **no redundancy**. If any single disk in the array fails, all data is lost. It is a performance-only solution.
- **RAID 1 (Mirroring):**
  - All data is written identically to two separate drives. The drives are an exact mirror of each other.
  - This provides **100% redundancy** and excellent fault tolerance. If one disk fails, the other can take over seamlessly.
  - **Drawback:** It's inefficient in terms of storage capacity, as only half of the total raw disk space is usable.

RAID 5 and RAID 6

These levels use a more efficient form of redundancy called **parity**. Parity is a calculated value (typically from an XOR operation) that allows the system to reconstruct lost data from a failed drive.

- **RAID 5 (Striping with Distributed Parity):**
  - Data and parity information are striped across all disks. The parity for a block of data is stored on a different disk than the data itself.
  - It can tolerate the loss of **one** disk in the array without data loss.
  - It offers a good balance of performance, capacity, and fault tolerance.
- **RAID 6 (Striping with Double Parity):**
  - Similar to RAID 5, but it stores a second, independent parity block for each stripe.
  - This allows the array to survive the simultaneous failure of **two** disks.
  - It is more reliable than RAID 5 but has a slightly higher write overhead.

RAID Level Comparison

RAID Level	Core Mechanism	Capacity Efficiency	Fault Tolerance
RAID 0	Striping	100%	None
RAID 1	Mirroring	50%	1 drive
RAID 5	Striping + Parity	$(n-1)/n$	1 drive
RAID 6	Striping + Double Parity	$(n-2)/n$	2 drives

Note: n is the number of drives in the array.

Solid-State Drives (SSD)

- SSDs use non-volatile NAND flash memory chips instead of spinning platters and moving heads.
- **Key Architectural Differences:**
  - **No moving parts:** Eliminates mechanical delays like seek time and rotational latency.
  - **Near-uniform access time:** Accessing a block at the "end" of the drive is as fast as accessing one at the "beginning."
  - **Flash Translation Layer (FTL):** A critical firmware layer that maps logical block addresses to physical ones, handles garbage collection, and performs **wear leveling** to extend the drive's lifespan.

Solid-State Drives (SSD)

- SSDs use non-volatile NAND flash memory chips instead of spinning platters and moving heads.
- **Key Architectural Differences:**
  - **No moving parts:** Eliminates mechanical delays like seek time and rotational latency.
  - **Near-uniform access time:** Accessing a block at the "end" of the drive is as fast as accessing one at the "beginning."
  - **Flash Translation Layer (FTL):** A critical firmware layer that maps logical block addresses to physical ones, handles garbage collection, and performs **wear leveling** to extend the drive's lifespan.

These architectural differences have profound implications for operating system design and performance optimization.

SSD vs. HDD – OS Implications

- **Disk Scheduling:** Since SSDs have virtually zero seek time, traditional disk scheduling algorithms like SSTF or LOOK are largely irrelevant and are often disabled or replaced with simpler policies (e.g., FCFS) in modern kernels.
- **I/O Models:** The non-linear access behavior of SSDs (where erasing a block is slow) has led to new I/O models and file systems.
- **Garbage Collection & TRIM:** OSes now need to support the 'TRIM' command, which informs the SSD of deleted files so the FTL can reclaim the physical blocks more efficiently.
- **SSD-Optimized File Systems:** File systems like **F2FS** (Flash-Friendly File System) are designed specifically for the unique characteristics of flash memory to improve performance and longevity.

Key Takeaways

- **RAID** is a key technology for combining multiple disks to improve performance and/or reliability.
- **RAID 0** provides a performance boost with no redundancy, while **RAID 1** offers full data mirroring.
- **RAID 5 and RAID 6** use parity to provide fault tolerance with better capacity efficiency than mirroring.
- **SSDs** fundamentally change the I/O landscape by eliminating mechanical delays, rendering traditional disk scheduling algorithms obsolete.
- The OS must adapt to modern storage technologies by providing new features like **TRIM** and supporting specialized file systems.

Next Week Preview: File I/O System Calls in Linux/UNIX

Now that we understand the underlying storage architecture, we will move up the stack to the interface that applications use to interact with files.

- **System Calls:** A detailed look at the fundamental I/O system calls: ‘open()‘, ‘read()‘, ‘write()‘, and ‘close()‘.
- **File Descriptors:** The integer handle used by the OS to track open files for each process.
- **Standard I/O vs. System Calls:** The difference between the ‘stdio‘ library functions and the low-level kernel system calls.
- **Buffering:** How the OS and libraries use buffers to optimize I/O performance.

Appendix: Quiz, Exercises, and Advanced Topics

Quiz & Exercises

1. What is the main difference between mirroring and parity-based redundancy in RAID?
2. A system has 5 disks of 1 TB each. What is the total usable capacity for a RAID 1, RAID 5, and RAID 6 configuration?
3. **Exercise:** Why is it impossible to create a RAID 5 array with only two disks?

Advanced Topics to Explore

- **RAID 10 and 50:** Nested RAID levels that combine the benefits of striping and mirroring/parity.
- **Erasure Coding:** A more advanced form of redundancy used in large-scale distributed storage systems.
- **Wear Leveling:** The specific mechanisms used in an SSD's FTL to evenly distribute write operations across all flash blocks.

Week 10

Agenda

1. File Descriptors and the UNIX I/O Model
2. System Calls: `open()`, `read()`, `write()`, `close()`
3. File Access Modes and Flags
4. Standard I/O vs. System I/O
5. Buffering and Performance Considerations
6. Summary & Key Takeaways
7. Next Week's Preview

File Descriptors and the UNIX I/O Model

- A **file descriptor (FD)** is a non-negative integer returned by the `open()` system call. It's an abstract handle used by the OS to refer to an open file.
- Every process maintains its own file descriptor table, which maps these integers to specific open files.
- By convention, every process starts with three standard file descriptors automatically opened:
  - **0** = Standard Input (`stdin`)
  - **1** = Standard Output (`stdout`)
  - **2** = Standard Error (`stderr`)

File Descriptors and the UNIX I/O Model

- A **file descriptor (FD)** is a non-negative integer returned by the `open()` system call. It's an abstract handle used by the OS to refer to an open file.
- Every process maintains its own file descriptor table, which maps these integers to specific open files.
- By convention, every process starts with three standard file descriptors automatically opened:
  - **0** = Standard Input (`stdin`)
  - **1** = Standard Output (`stdout`)
  - **2** = Standard Error (`stderr`)

Analogy

Think of a file descriptor as a "ticket number" you get from the OS when you request to work with a file. You use this number for all subsequent operations, and the OS uses it to look up the file's information (current position, status flags, etc.).

`open()` and `close()` System Calls

The `open()` system call is the first step to performing I/O. It asks the OS to open a file and returns a file descriptor. `close()` releases that descriptor.

```
int fd = open("data.txt", O_RDONLY);
if (fd < 0) {
    perror("open failed");
    return 1;
}
// Use fd for read/write operations...
close(fd); // Release the file descriptor
```

open() and close() System Calls

The open() system call is the first step to performing I/O. It asks the OS to open a file and returns a file descriptor. close() releases that descriptor.

```
int fd = open("data.txt", O_RDONLY);
if (fd < 0) {
    perror("open failed");
    return 1;
}
// Use fd for read/write operations...
close(fd); // Release the file descriptor
```

Flags (Access Modes): The second argument to open() uses bitwise OR (|) to combine flags that specify how the file should be opened:

- O_RDONLY, O_WRONLY, O_RDWR: Read-only, write-only, or read-write.
- O_CREAT: Creates the file if it does not exist. (Requires a third 'mode' argument for permissions).
- O_APPEND: Appends data to the end of the file.
- O_TRUNC: Truncates the file to a size of zero if it already exists.

`read()` and `write()` System Calls

These are the fundamental system calls for transferring data to and from a file.

```
char buf[100];
int n_read = read(fd, buf, 100);  // Read up to 100
    bytes from 'fd'

if (n_read > 0) {
     int n_written = write(1, buf, n_read); // Write '
         n_read' bytes to stdout
}
```

`read()` and `write()` System Calls

These are the fundamental system calls for transferring data to and from a file.

```
char buf[100];
int n_read = read(fd, buf, 100);  // Read up to 100
    bytes from 'fd'

if (n_read > 0) {
     int n_written = write(1, buf, n_read); // Write '
         n_read' bytes to stdout
}
```

Return Value Significance: The return value is critical for error checking and flow control.

- **¿ 0**: The number of bytes successfully read or written.
- **0**: For `read()`, this indicates the **end of file (EOF)** has been reached. For `write()`, it means no bytes were written.
- **-1**: An error occurred. The global variable `errno` is set to indicate the specific error.

System I/O vs. Standard I/O

- **System I/O** (`open()`, `read()`, `write()`) is the low-level, unbuffered interface to the kernel. Each call typically results in a system call and a context switch, making it relatively slow for small, frequent I/O operations.
- **Standard I/O** (`fopen()`, `fread()`, `fwrite()`) is a higher-level library (e.g., `stdio.h` in C) built on top of System I/O. It provides a more convenient, buffered interface.

System I/O vs. Standard I/O

- **System I/O** (`open()`, `read()`, `write()`) is the low-level, unbuffered interface to the kernel. Each call typically results in a system call and a context switch, making it relatively slow for small, frequent I/O operations.
- **Standard I/O** (`fopen()`, `fread()`, `fwrite()`) is a higher-level library (e.g., `stdio.h` in C) built on top of System I/O. It provides a more convenient, buffered interface.

Code Example (Standard I/O):

```
FILE *fp = fopen("data.txt", "r");
if (fp == NULL) { /* error handling */ }

char buf[100];
fgets(buf, 100, fp); // Reads a line into the buffer
fclose(fp);
```

Standard I/O is generally preferred for application-level code due to its ease of use and performance optimizations through buffering.

Buffering and Performance

- **What is a Buffer?** A buffer is a temporary storage area in user space (part of the application's memory).
- **How it Works:** The Standard I/O library reads large chunks of data from the disk into this buffer with a single system call. Subsequent small reads (`fgetc()`, etc.) are then satisfied directly from the buffer, avoiding frequent, expensive system calls.
- **Trade-off:** Buffering introduces a slight delay before data is actually written to the disk, but it significantly reduces the number of costly context switches, leading to a major performance improvement for most workloads.
- **Buffer Control:** The `setvbuf()` function allows programmers to control the buffering policy (e.g., full buffering, line buffering, or no buffering at all).

Key Takeaways

- The **UNIX I/O model** uses a small integer called a **file descriptor** to represent an open file, providing a consistent interface for all I/O devices.
- Low-level **System Calls** like `open()`, `read()`, `write()`, and `close()` are the kernel's fundamental interface to file systems and devices.
- **File flags** such as `O_RDWR` and `O_CREAT` are used to specify the desired file access mode and behavior.
- **Standard I/O** is a library that provides a high-level, buffered abstraction over the raw system calls, optimizing performance by reducing context switches.

Next Week Preview: Final Wrap-Up and Review

This marks the final content lecture for the OS module. Next week, we will transition to a comprehensive review.

- **Core Concepts Recap:** We will revisit and connect key topics from the entire course, including Process Management, Concurrency, Virtual Memory, and File Systems.
- **Integration Review:** We will discuss how different OS components interact with each other (e.g., how virtual memory and file systems are intertwined).
- **Exam Preparation:** A focused Q&A session to address student questions and clarify concepts in preparation for the final exam.

Appendix: Quiz, Exercises, and Advanced Topics I

Quiz & Exercises

1. What would the file descriptor be for a file opened after a process has started and before any other files are opened?
2. Explain the difference in a C program between using `write(1, buf, n)` and `fprintf(stdout, "...", ...)`.
3. **Exercise:** Write the `open()` call to open a file named ‘log.txt‘ for writing, creating it if it doesn’t exist, and appending all new data to the end of the file.

Appendix: Quiz, Exercises, and Advanced Topics II

Advanced Topics to Explore

- **‘mmap()‘ (Memory-mapped I/O):** A powerful system call that maps a file directly into a process’s virtual address space.
- **‘ioctl()‘:** A device-specific system call used for special control operations on a file or device.
- **File Locking:** Mechanisms to control concurrent access to a file by multiple processes.
- **Non-blocking I/O:** Using flags like `O_NONBLOCK` to prevent a process from being put to sleep during I/O operations.

Week 11

Agenda

1. Memory Management Recap
2. File Systems Recap
3. Disk I/O and Storage Review
4. Interconnections and Integration
5. Project Prompts and Lab Alignment
6. Summary & Key Takeaways
7. Next Week's Preview

Memory Management Recap

- **Logical $\leftrightarrow$ Physical Address Translation:** We saw how the OS provides an abstraction of a process's memory space, translating its logical addresses into physical addresses in RAM.
- **Paging and Segmentation:** These are techniques for dividing memory into fixed-size pages or variable-size segments, respectively, to allow for non-contiguous allocation and better memory utilization.
- **Virtual Memory and Page Faults:** We explored how virtual memory creates the illusion of infinite memory, triggering a *page fault* when a requested page is not present in RAM.
- **Page Replacement:** Algorithms like FIFO, LRU, and Clock are used to decide which page to evict from memory to make room for a new one, with the goal of minimizing future page faults.

Memory Management Recap

- **Logical $\leftrightarrow$ Physical Address Translation:** We saw how the OS provides an abstraction of a process's memory space, translating its logical addresses into physical addresses in RAM.
- **Paging and Segmentation:** These are techniques for dividing memory into fixed-size pages or variable-size segments, respectively, to allow for non-contiguous allocation and better memory utilization.
- **Virtual Memory and Page Faults:** We explored how virtual memory creates the illusion of infinite memory, triggering a *page fault* when a requested page is not present in RAM.
- **Page Replacement:** Algorithms like FIFO, LRU, and Clock are used to decide which page to evict from memory to make room for a new one, with the goal of minimizing future page faults.

Core Challenge

The memory manager's role is to balance performance, protection, and flexibility for all running processes.

File System Concepts Recap

- **Metadata via Inodes/FCBs:** A file is more than just its data; we learned about the importance of **metadata** (permissions, size, timestamps) stored in an **inode** (or File Control Block).
- **Directory Structures:** Directories map human-readable filenames to inodes, organizing files in hierarchical structures like trees or acyclic graphs.
- **File Allocation Strategies:** We compared the trade-offs of **contiguous**, **linked**, and **indexed** allocation in balancing performance, fragmentation, and flexibility.
- **Free Space Management:** We reviewed techniques like bitmaps and linked lists, which the file system uses to efficiently track and allocate unallocated disk blocks.

File System Concepts Recap

- **Metadata via Inodes/FCBs:** A file is more than just its data; we learned about the importance of **metadata** (permissions, size, timestamps) stored in an **inode** (or File Control Block).
- **Directory Structures:** Directories map human-readable filenames to inodes, organizing files in hierarchical structures like trees or acyclic graphs.
- **File Allocation Strategies:** We compared the trade-offs of **contiguous**, **linked**, and **indexed** allocation in balancing performance, fragmentation, and flexibility.
- **Free Space Management:** We reviewed techniques like bitmaps and linked lists, which the file system uses to efficiently track and allocate unallocated disk blocks.

Goal

To provide a reliable, fast, and consistent interface for an application to access persistent data.

Disk and I/O Management Recap

- **Access Latency:** We identified that **seek time** (the time to move the disk head) is the dominant factor in access latency for Hard Disk Drives (HDDs).
- **Disk Scheduling:** Algorithms like **FCFS**, **SSTF**, **SCAN**, and **LOOK** were introduced to reorder I/O requests and minimize total head movement, thereby improving I/O throughput.
- **RAID:** We discussed how RAID (Redundant Array of Independent Disks) combines multiple physical drives to improve either performance (RAID 0) or reliability (RAID 1, 5, 6).
- **Modern Storage:** The advent of **SSDs** has revolutionized I/O by eliminating seek time, necessitating new OS considerations like the Flash Translation Layer (FTL) and the 'TRIM' command.

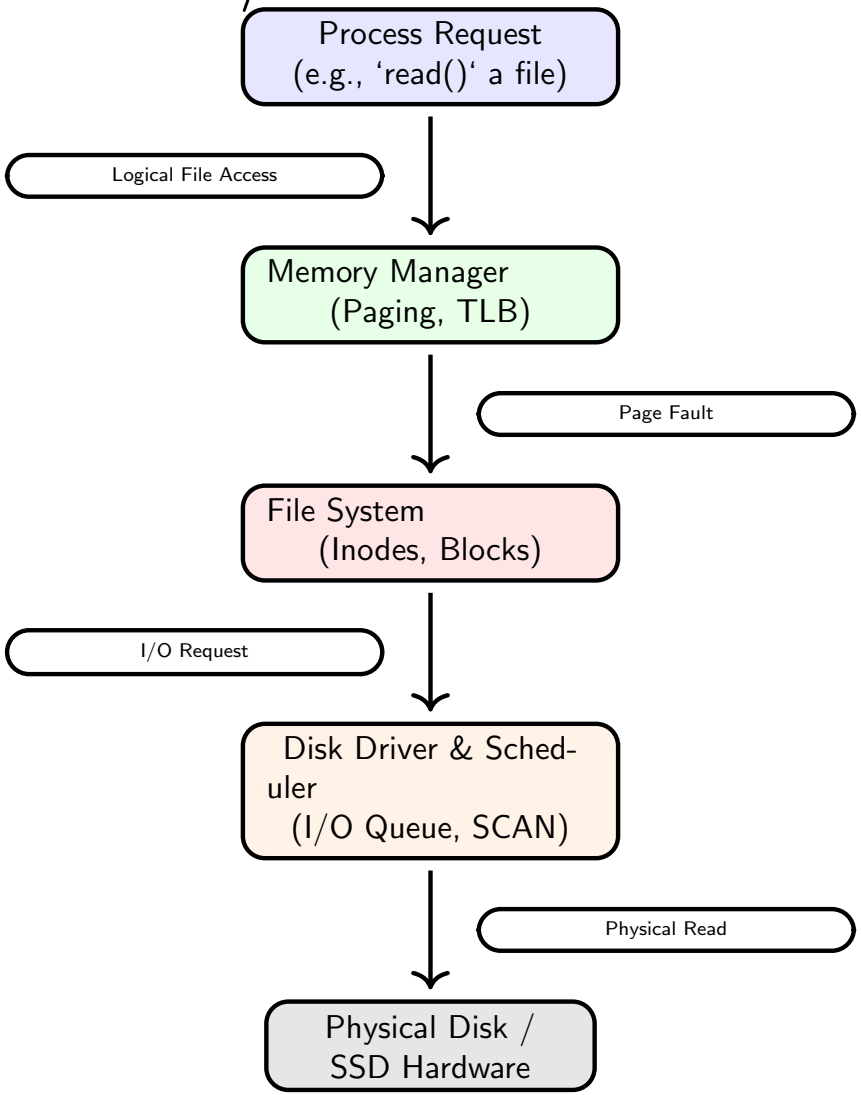
Disk and I/O Management Recap

- **Access Latency:** We identified that **seek time** (the time to move the disk head) is the dominant factor in access latency for Hard Disk Drives (HDDs).
- **Disk Scheduling:** Algorithms like **FCFS**, **SSTF**, **SCAN**, and **LOOK** were introduced to reorder I/O requests and minimize total head movement, thereby improving I/O throughput.
- **RAID:** We discussed how RAID (Redundant Array of Independent Disks) combines multiple physical drives to improve either performance (RAID 0) or reliability (RAID 1, 5, 6).
- **Modern Storage:** The advent of **SSDs** has revolutionized I/O by eliminating seek time, necessitating new OS considerations like the Flash Translation Layer (FTL) and the 'TRIM' command.

I/O Path

The performance of an I/O operation is a function of the device's physical latency, the OS's queuing policy, and the chosen scheduling algorithm.

Integration View: The I/O Stack



The path of a file access request from a process to the physical hardware, highlighting the role of each OS component.

Project / Lab Alignment Ideas

These project ideas build upon the concepts covered in this module, allowing for hands-on application of theory.

- **Paging Simulator:** Visualize a page replacement algorithm (e.g., LRU) by showing a reference string and tracking page faults in real-time.
- **Simple File System Emulator:** Implement a small, in-memory file system that manages blocks, inodes, and a directory structure.
- **Disk Scheduling Animator:** Create an animated simulation of different disk scheduling algorithms, showing the head's movement across the cylinders.
- **I/O Benchmarker:** Use system utilities like 'dd' to measure and compare the I/O performance of different block sizes or access patterns on an SSD vs. an HDD.

Project / Lab Alignment Ideas

These project ideas build upon the concepts covered in this module, allowing for hands-on application of theory.

- **Paging Simulator:** Visualize a page replacement algorithm (e.g., LRU) by showing a reference string and tracking page faults in real-time.
- **Simple File System Emulator:** Implement a small, in-memory file system that manages blocks, inodes, and a directory structure.
- **Disk Scheduling Animator:** Create an animated simulation of different disk scheduling algorithms, showing the head's movement across the cylinders.
- **I/O Benchmarker:** Use system utilities like 'dd' to measure and compare the I/O performance of different block sizes or access patterns on an SSD vs. an HDD.

Advanced Project Concepts:

- Build a simulated *Flash Translation Layer* for an SSD, including policies for wear leveling and garbage collection.
- Implement a multi-level indexed file allocation scheme to see how it scales for very large files.

Key Takeaways

- The OS's primary role is to act as a **resource manager**, providing an abstraction over physical hardware to make it easy for applications to run.
- Performance is a delicate balance of **allocation strategies** (memory, file blocks), **access patterns**, and **scheduling algorithms**.
- The different layers of the OS—the memory manager, file system, and disk scheduler—are **not isolated**. They form a tightly integrated stack, where a decision at one layer directly impacts the performance of others.
- A deep understanding of these layers is crucial for writing efficient and reliable software.

Next Week: Transition to Faculty C Topics

This marks the end of our module on resource management and system abstractions. Next week, we will shift our focus from **what** the OS manages to **how** it coordinates and secures these resources across different environments.

- **Synchronization in Distributed Systems:** Moving beyond a single machine.
- **Deadlocks:** A deeper dive into prevention, avoidance, and detection.
- **Virtualization and Containers:** The new frontier in resource isolation and coordination.
- **Security and Protection:** OS mechanisms for secure, coordinated execution.

A great deal of what we've learned this term is the foundation for these advanced topics!

Week 12

Agenda: Integrated Review & Advanced Exercises

1. Integrated Conceptual & Analytical Questions
2. Algorithmic & Coding Exercises
3. Practical Linux Commands & System Exploration
4. Case Studies & Research Prompts
5. Simulation & Visualization Project Ideas
6. Summary & Key Takeaways
7. Final Review & Next Steps

Integrated Conceptual & Analytical Questions I

These questions require understanding concepts from multiple weeks.

- **(Memory Management & File Systems):**
  - When a process opens a file, describe the journey of a logical address (within the process's address space) used to access a byte in that file, all the way to its physical location on disk. What OS components are involved at each stage of translation and retrieval?
  - How does the OS prevent a process from accessing sensitive data in a file that is memory-mapped into its virtual address space, even if the file contains privileged information? (Consider virtual memory protection mechanisms).
- **(Paging & Disk I/O):**
  - Explain the potential performance impact of a consistently high page fault rate on a system using an HDD, particularly considering disk scheduling algorithms. How would this impact differ if the system used an SSD?

Integrated Conceptual & Analytical Questions II

- In a system with limited physical memory, why might adding more processes (increasing multiprogramming) lead to thrashing, and how can the OS detect and mitigate this condition using concepts from both memory management and process scheduling?

- **(Concurrency & File I/O):**
  - Discuss the challenges of implementing a multi-threaded application that performs concurrent writes to the same section of a single file. What synchronization mechanisms (e.g., mutexes, semaphores, file locks) would be necessary, and what are the trade-offs?
  - If two processes simultaneously attempt to ‘open()‘ the same file with different access modes (e.g., one for read, one for write), how does the OS typically handle this? What metadata is involved in enforcing proper access?
- **(Overall System Design):**
  - How does the choice of disk block size affect both internal fragmentation in file allocation and the efficiency of demand paging? Provide a scenario where a very large block size could be detrimental to overall system performance.

Integrated Conceptual & Analytical Questions III

- Describe a scenario where a system using RAID 0 could lead to a catastrophic failure that a system using RAID 1 would survive. What are the implications for a high-performance, write-intensive application?

Algorithmic & Coding Exercises I

Implement solutions to these problems in C/C++ or Python.

- **(Page Replacement Simulator):**
  - Implement a simulator for **FIFO**, **LRU**, and **Clock** page replacement algorithms. Your program should take a reference string and number of frames as input and output the number of page faults for each algorithm. Include a feature to demonstrate *Belady's Anomaly* for FIFO.
- **(File Allocation Simulator):**
  - Create a program that simulates a simplified file system. It should manage a fixed number of disk blocks using a **bitmap** for free space management. Implement 'create_file()', 'delete_file()', and 'read_file()' functions using **indexed allocation**.
- **(Disk Scheduling Simulator):**
  - Write a program that simulates **FCFS**, **SSTF**, **SCAN**, and **LOOK** disk scheduling algorithms. Take an initial head position and a queue of pending requests (cylinder numbers) as input. Output the sequence of head movements and the total head movement for each algorithm.

Algorithmic & Coding Exercises II

- **(Producer-Consumer with File I/O):**
  - Extend the Producer-Consumer problem to involve file I/O. The producer thread reads lines from an input file, processes them, and puts them into a shared buffer. The consumer thread takes processed lines from the buffer and writes them to an output file. Use pthreads (`pthread_mutex_t`, `pthread_cond_t`) for synchronization.

Practical Linux Commands & System Exploration I

Use a Linux environment (VM or WSL) for these exercises.

- **(Memory Usage Analysis):**
  - Use `free -h` and `cat /proc/meminfo` to understand the system's current RAM usage, swap usage, and kernel buffer/cache sizes. Explain what `MemAvailable` truly represents.
  - Run a memory-intensive program (e.g., a simple C program that allocates a large array). While it's running, use `top`, `htop`, and `pmap <pid>` to analyze its virtual (`VSZ`) and resident (`RSS`) memory usage. How do these values change over time?
- **(Disk I/O Performance):**
  - Use `iostat -x 1` to monitor disk I/O statistics (read/write throughput, I/O requests per second, queue size) while simultaneously performing large file operations using `dd` (e.g., `dd if=/dev/zero of=testfile bs=1M count=1024 conv=fdatasync`). Analyze the impact of `conv=fdatasync` on performance.

Practical Linux Commands & System Exploration II

- Explore different I/O schedulers in Linux: `cat /sys/block/sda/queue/scheduler`. Change the scheduler (e.g., to ‘noop‘, ‘deadline‘, ‘mq-deadline‘) using ‘echo‘ and re-run your ‘dd‘ tests. Observe if there's a significant performance difference on an HDD vs. an SSD.

- **(File System Metadata):**
  - Create a file, then create a hard link to it using `ln`. Use `ls -li` to inspect their inode numbers and link counts. Explain the implications for deletion.
  - Create a symbolic link (symlink) using `ln -s`. Compare its inode number and size to the original file. How does the OS handle deleting a symlink vs. a hard link?

Case Studies & Research Prompts I

These topics encourage deeper dives into real-world systems and research.

- **(Advanced OS Memory Management):**
  - Research and compare the memory management techniques used in two different modern operating systems (e.g., Linux's CFS & memory management, Windows' Virtual Memory Manager, or macOS's memory compression). Focus on their page replacement policies, TLB management, and handling of memory overcommit.
  - Investigate **Transparent HugePages (THP)** in Linux. How does it work, what problem does it solve, and what are its potential performance benefits and drawbacks?
- **(Modern File System Features):**
  - Choose a modern file system (e.g., **Btrfs**, **ZFS**, **Ext4**'s advanced features). Research its key innovations beyond basic allocation (e.g., copy-on-write, snapshots, checksums, data deduplication, subvolumes, built-in RAID). Discuss how these features address real-world data management challenges.

Case Studies & Research Prompts II

- Research the concept of a **Log-Structured File System (LSFS)**. How does it fundamentally differ from traditional file systems in its write strategy, and what are its advantages and disadvantages, especially for flash-based storage?

- **(Emerging Storage Technologies):**
  - Research **Non-Volatile Memory Express (NVMe)** and **Persistent Memory (PMem)** technologies. How do these new storage classes challenge traditional CPU-I/O models and memory hierarchies? What implications do they have for future OS designs and application programming?
  - Explore the concept of **Computational Storage Drives (CSD)**. How do they offload processing directly to the storage device, and what problems are they designed to solve in big data and AI workloads?

Simulation & Visualization Project Ideas I

Design and potentially implement a visual simulator for understanding key OS concepts.

- **(Virtual Memory Visualization):**
  - Create an interactive visualization tool that shows the virtual-to-physical address translation process. Allow users to input logical addresses and see how the MMU, page table, and TLB cooperate to find the physical address. Highlight page faults and TLB hits/misses.
- **(Concurrency Visualization):**
  - Develop a simulation of multiple threads accessing a critical section. Visualize the use of mutexes or semaphores to prevent race conditions. Show threads attempting to enter, waiting, and exiting the critical section.
- **(File Fragmentation Analyzer):**

Simulation & Visualization Project Ideas II

- Build a simple graphical tool that represents disk blocks. Allow users to "create" and "delete" files (using contiguous allocation) and visualize how external fragmentation develops over time. Implement a "compaction" feature to show its effect.

- **(RAID Configuration Explorer):**
  - Create an interactive diagram that demonstrates data striping and parity distribution for different RAID levels (0, 1, 5, 6). Allow users to "simulate" a disk failure and show how data is reconstructed (or lost).

Key Takeaways from the Entire Course

- The OS acts as a complex **resource manager**, providing crucial abstractions over raw hardware.
- Understanding the interplay between **hardware support** (MMU, TLB, Disk Controllers) and **software policies** (scheduling, memory management, file systems) is fundamental.
- Concepts like **concurrency, synchronization, virtual memory, and persistent storage** are deeply interconnected and essential for building reliable and performant systems.
- **Trade-offs** are inherent in OS design: performance vs. fairness, simplicity vs. flexibility, capacity vs. redundancy.
- The OS is constantly evolving to adapt to new hardware (SSDs, NVMe, multi-core CPUs) and new demands (cloud computing, containers, virtualization).

Final Review & Next Steps

Congratulations on completing the core Operating Systems curriculum!

- This week's exercises are designed to help you synthesize knowledge across all modules.
- Use these prompts to solidify your understanding and prepare for the final assessment.
- Review all past lecture slides, notes, and lab materials.
- Don't hesitate to ask questions and discuss concepts with your peers and instructors.

What's Next?

- Your next steps may involve exploring more advanced topics in distributed systems, security, or specific kernel programming.
- The foundational knowledge you've gained here will be invaluable for any deeper dive into computer science and engineering.

Appendix: Comprehensive Quiz & Exercises I

Section A: Conceptual & Analytical Questions

1. A user process attempts to execute an instruction at logical address ‘0x1000‘. This address falls within a page that is currently not in physical memory but is valid. Trace the full sequence of events that the OS and hardware undertake to service this request, from the CPU generating the address to the instruction successfully executing.
2. Compare and contrast the memory management challenges posed by multi-threaded processes versus multi-process systems. What common mechanisms solve these, and what unique issues arise in each context?
3. Discuss the "write amplification" problem in SSDs. How does it affect the lifespan of an SSD, and what mechanisms (both at the firmware and OS level) are in place to mitigate it?
4. Explain how the principle of **Locality of Reference** is exploited by both caching (e.g., TLB) and virtual memory (e.g., demand paging). Provide examples for both temporal and spatial locality in typical program execution.

Appendix: Comprehensive Quiz & Exercises II

5. You are designing a file system for a video streaming server. Which file allocation method (contiguous, linked, or indexed) would you choose and why? Consider sequential access performance, random access performance, and file size flexibility.

Section B: Algorithmic & Coding Challenges

1. **Page Table Implementation (C/C++):** Implement a simplified multi-level page table. Given a virtual address, determine its physical address. Assume a 32-bit logical address space, 4KB page size, and a two-level page table structure. Handle page faults by printing a message, but no actual disk I/O.
2. **Synchronization Challenge (C/C++ using pthreads):** Implement a classic "Readers-Writers" problem where multiple reader threads can access a shared resource concurrently, but only one writer thread can access it exclusively. Use mutexes and condition variables.

Appendix: Comprehensive Quiz & Exercises III

3. **Disk Block Defragmenter (Python):** Write a Python script that simulates a disk with a fixed number of blocks. Allow creation of files (as lists of block numbers). Implement a function to "defragment" a file, reorganizing its blocks to be contiguous if possible.
4. **RAID 5 Parity Calculator (Python):** Given a set of data blocks (e.g., integers or bytes), write a function that calculates the parity block for a RAID 5 array. Then, simulate a disk failure and write a function to reconstruct the lost data block using the remaining data and parity.

Section C: Practical Linux & System Exploration

1. **Process Memory Map (Linux):** Pick any running process (e.g., a web browser, a simple text editor). Use 'cat /proc/¡pid¿/maps' to examine its virtual memory layout. Identify sections like stack, heap, code (.text), data (.data), and shared libraries. Explain what each section typically contains.

Appendix: Comprehensive Quiz & Exercises IV

2. **Swap Space Observation (Linux):** Create and activate a swap file (`dd`, `mkswap`, `swapon`). Run a program that consumes a lot of memory until swap usage increases (`free -h`). Then, terminate the program and observe how the swap space is reclaimed.
3. **File System Block Usage (Linux):** Using `debugfs` (or a similar tool for your file system, e.g., `btrfs inspect-fibs` for Btrfs), try to find the actual physical block numbers on disk that a specific file occupies. (Note: This might require root privileges and careful usage).
4. **I/O Benchmarking with Different Buffering (C/Python):** Write a simple program that performs a large file copy operation. Implement two versions: one using low-level system calls (`read()`, `write()`) and another using buffered standard I/O (`fread()`, `fwrite()`). Compare their performance for different buffer sizes and file sizes.

Section D: Case Studies & Advanced Concepts

Appendix: Comprehensive Quiz & Exercises V

1. **Virtualization vs. Containers (Comparative Analysis):**
   - Create a detailed comparison of **Virtual Machines (VMs)** and **Containers (e.g., Docker)** in terms of resource isolation, startup time, overhead, and typical use cases. Explain which OS components are leveraged by each technology to achieve isolation.
2. **Real-time Operating Systems (RTOS):**
   - Research the fundamental differences between a general-purpose OS (like Linux or Windows) and a Real-time Operating System (RTOS). Focus on their scheduling algorithms, memory management, and how they provide guaranteed response times for critical operations. Provide examples of where RTOS are used.
3. **Distributed File Systems (Case Study):**
   - Pick a distributed file system (e.g., NFS, HDFS, Ceph). Research its architecture, how it handles data replication and consistency across multiple nodes, and its fault tolerance mechanisms. Discuss its advantages and disadvantages compared to local file systems.
4. **OS Security Primitives:**

Appendix: Comprehensive Quiz & Exercises VI

- Research fundamental OS security primitives beyond simple file permissions, such as **Capabilities**, **Access Control Lists (ACLs)**, and **Security-Enhanced Linux (SELinux)**. Explain how these provide fine-grained control over resources and enhance system security.