Context Switching and Dispatcher

Week 4

SDB

Autumn 2025

Agenda: Orchestrating CPU Execution

Understanding How Processes Get Their Turn on the CPU

1. **Recap: Context Switching** *(The Essential Mechanism)*
2. **The Dispatcher's Role** *(Bridging Scheduler to CPU)*
3. **Process Scheduling Cycle** *(The Continuous Flow)*
4. **Context Switch Performance** *(Overhead & Optimization)*
5. **Degree of Multiprogramming** *(System Load & Performance)*
6. **Real-World Implementations** *(Linux vs. Windows)*
7. **Key Takeaways & Discussion**

Agenda: Orchestrating CPU Execution

Understanding How Processes Get Their Turn on the CPU

1. **Recap: Context Switching** *(The Essential Mechanism)*
2. **The Dispatcher's Role** *(Bridging Scheduler to CPU)*
3. **Process Scheduling Cycle** *(The Continuous Flow)*
4. **Context Switch Performance** *(Overhead & Optimization)*
5. **Degree of Multiprogramming** *(System Load & Performance)*
6. **Real-World Implementations** *(Linux vs. Windows)*
7. **Key Takeaways & Discussion**

Think Ahead: The CPU's Dance

The OS decides *which* process runs next (scheduling). But how does it *actually* make that happen, switching the CPU's attention seamlessly from one process to another? What are the practical costs involved, and how does the **number of active processes** impact these costs and overall system health?

Recap: What is Context Switching?

Review: The State Transfer Mechanism

Context switching is the fundamental operating system mechanism for saving the state (context) of a currently executing process and restoring the state of another process. This allows the CPU to resume execution for the latter process from precisely where it last left off.

Why is this Mechanism Indispensable?

- Enables **multitasking**: Allows multiple processes to share a single CPU.
- Supports **fair CPU sharing**: Prevents one process from monopolizing the CPU.
- Ensures **system responsiveness**: Guarantees interactive applications remain fluid.
- Facilitates **resource utilization**: Keeps I/O devices busy while CPU-bound tasks wait.

Recap: What is Context Switching?

Review: The State Transfer Mechanism

Context switching is the fundamental operating system mechanism for saving the state (context) of a currently executing process and restoring the state of another process. This allows the CPU to resume execution for the latter process from precisely where it last left off.

Why is this Mechanism Indispensable?

- Enables **multitasking**: Allows multiple processes to share a single CPU.
- Supports **fair CPU sharing**: Prevents one process from monopolizing the CPU.
- Ensures **system responsiveness**: Guarantees interactive applications remain fluid.
- Facilitates **resource utilization**: Keeps I/O devices busy while CPU-bound tasks wait.

Quick Recall

From Week 3, what critical data structure holds all the information needed for a context switch? Why is it crucial?

Context: The Snapshot of a Process

What exactly constitutes a Process's "Context" (stored in its PCB)?

- **CPU Registers:** All general-purpose registers, index registers, segment registers, etc.
- **Program Counter (PC):** The address of the next instruction to be executed.
- **Stack Pointer (SP):** Points to the top of the current process's stack.
- **Memory Management Information:** Base and limit registers, page tables (for virtual memory systems).
- **OS Metadata:** Process state, priority, Process ID (PID), pointers to various queues, open file descriptors, I/O status.

Context: The Snapshot of a Process

What exactly constitutes a Process's "Context" (stored in its PCB)?

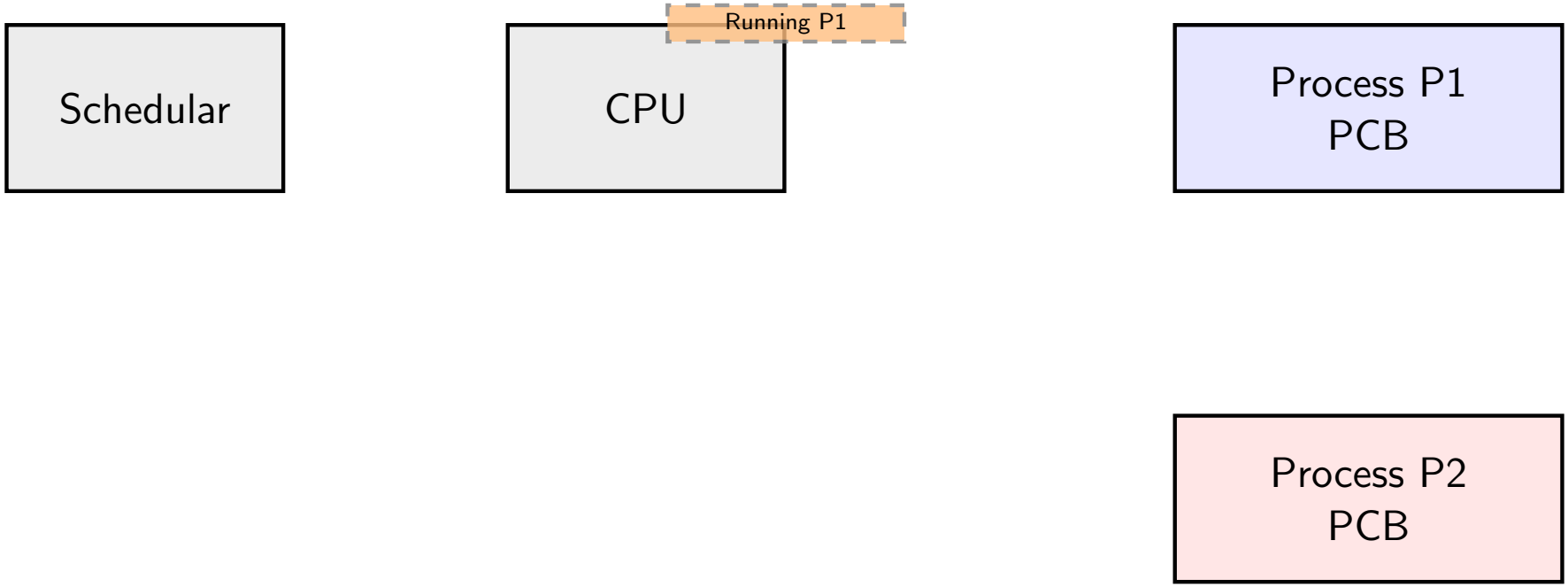
- **CPU Registers:** All general-purpose registers, index registers, segment registers, etc.
- **Program Counter (PC):** The address of the next instruction to be executed.
- **Stack Pointer (SP):** Points to the top of the current process's stack.
- **Memory Management Information:** Base and limit registers, page tables (for virtual memory systems).
- **OS Metadata:** Process state, priority, Process ID (PID), pointers to various queues, open file descriptors, I/O status.

The Mechanism: Save & Load

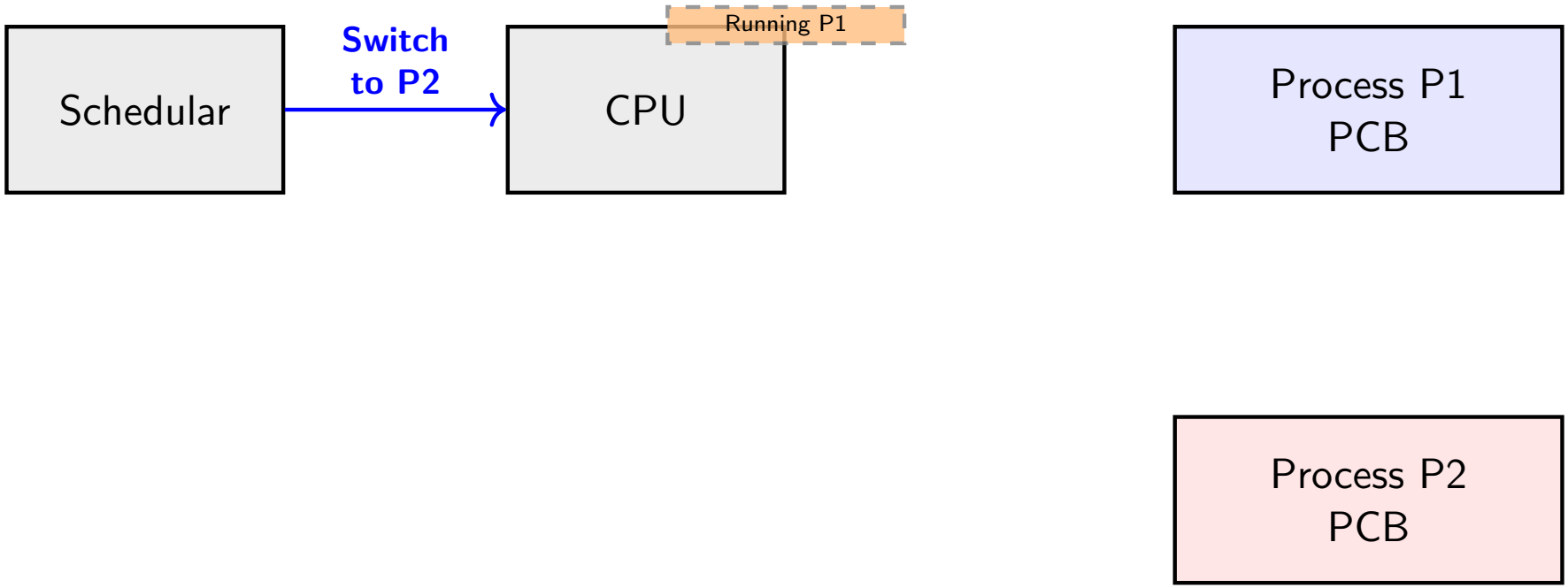
Saving: The CPU's current context (from the process about to stop) is meticulously stored into its corresponding PCB in main memory.
Loading: The CPU's new context (for the process selected to run next) is retrieved from **its** PCB in main memory and loaded into the CPU's registers.

This entire operation ensures that when a process regains the CPU, it can continue precisely from where it was interrupted, as if nothing happened.

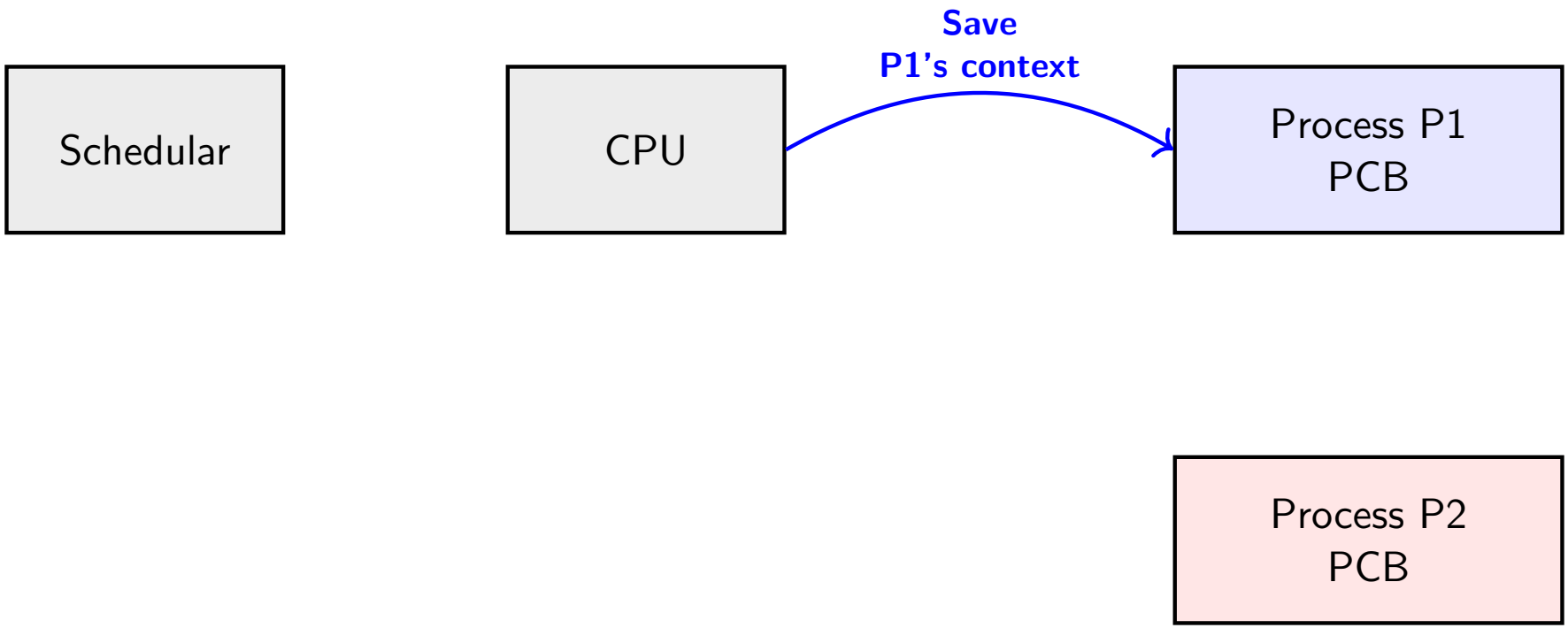
Context Switch Visualization: The CPU's Hand-off



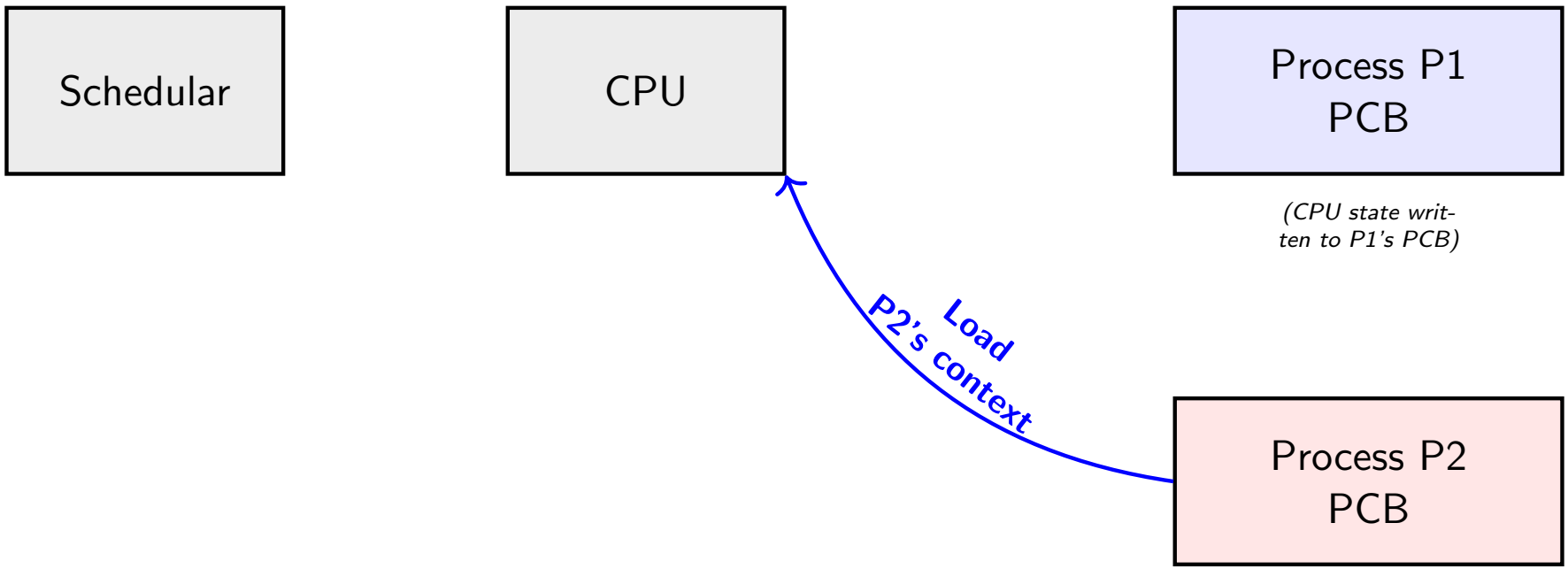
Context Switch Visualization: The CPU's Hand-off



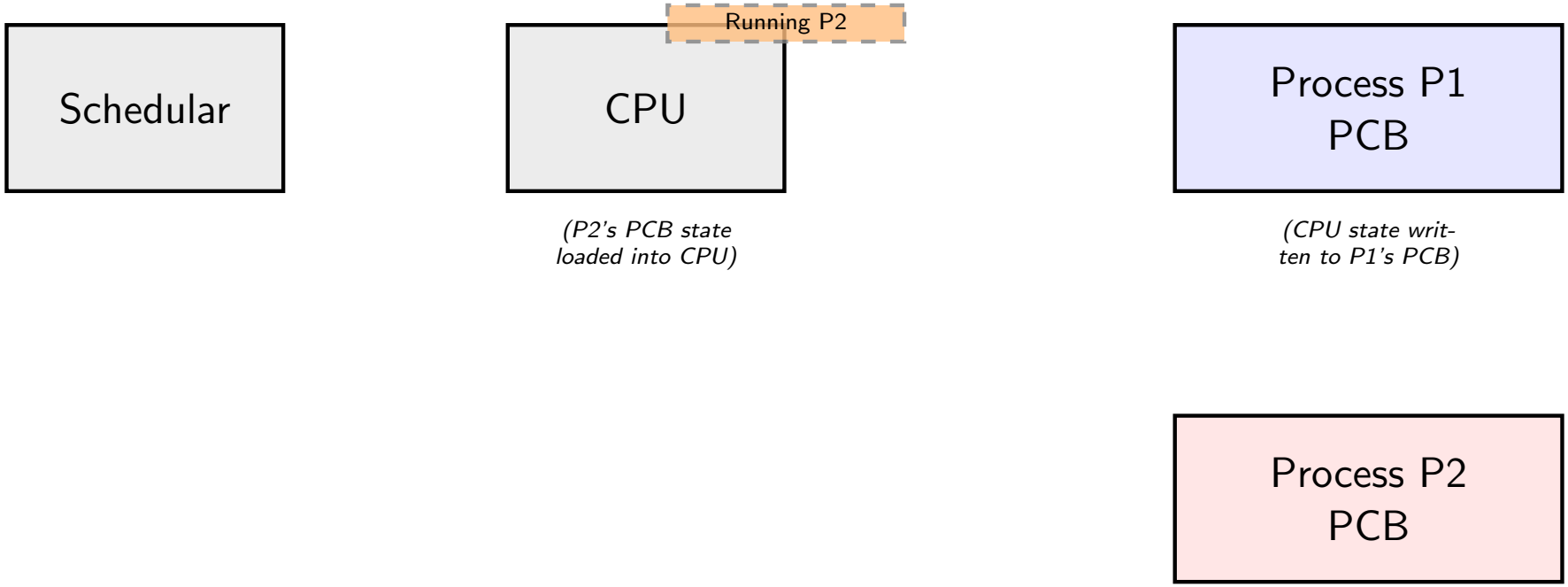
Context Switch Visualization: The CPU's Hand-off



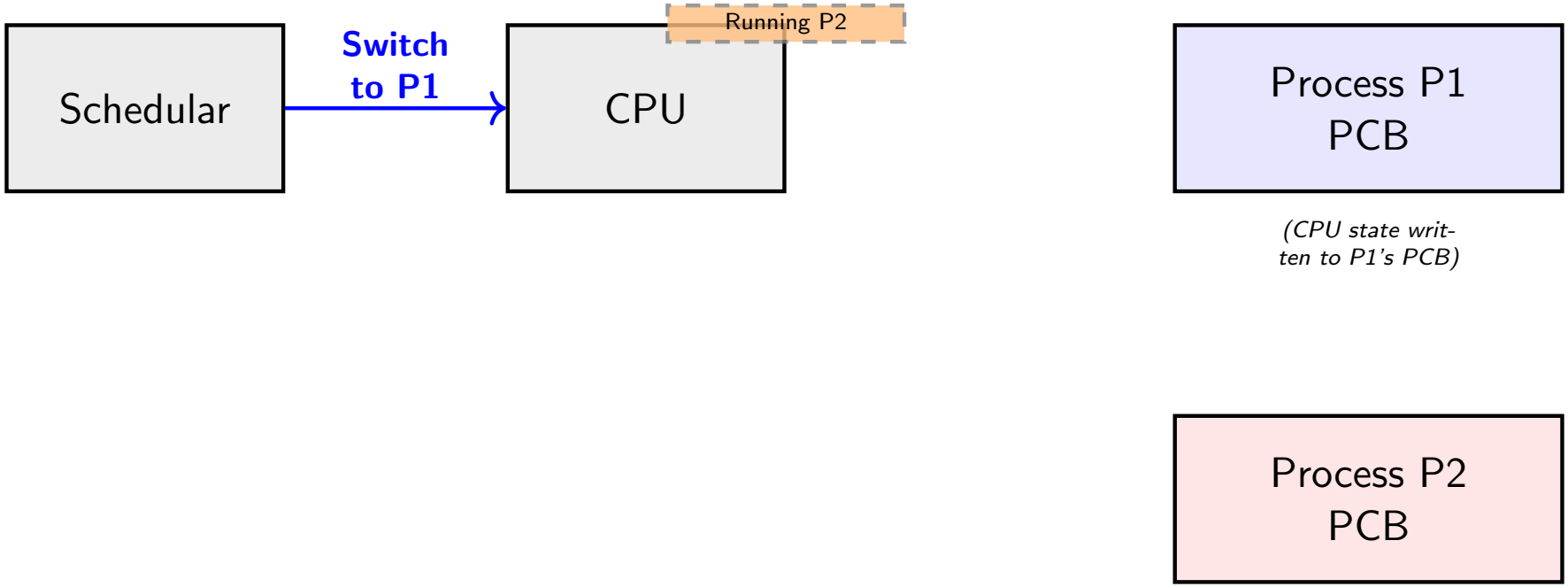
Context Switch Visualization: The CPU's Hand-off



Context Switch Visualization: The CPU's Hand-off



Context Switch Visualization: The CPU's Hand-off



This rapid save-and-load operation ensures that each process perceives it has exclusive access to the CPU.

CPU Scheduling: Orchestrating Process Execution I

After understanding how the OS switches between processes (Context Switching) and hands over CPU control (Dispatcher), the next fundamental question is: **Who** decides **which** process runs next?

The CPU Scheduler: The Decision-Maker

- The **CPU Scheduler** (or Short-Term Scheduler) is the OS component responsible for **selecting a process from the Ready Queue** and allocating the CPU to it.
- It's a critical component in **multiprogramming**, ensuring the CPU is always busy (or that waiting processes get a chance).

Why is CPU Scheduling Necessary?

- **Efficient Resource Utilization:** Keeps the CPU as busy as possible, especially during I/O waits.
- **Multitasking/Multiprogramming:** Allows multiple processes to share the CPU, giving the illusion of concurrent execution.
- **Responsiveness:** Ensures interactive applications remain responsive to user input.

CPU Scheduling: Orchestrating Process Execution II

- **Fairness:** Provides a reasonable share of CPU time to all competing processes.
- **Throughput:** Maximizes the number of processes completed per unit of time.

Scheduler's Place in the Execution Flow (Recap & Connection):

- **Input:** The Ready Queue (processes ready to run).
- **Decision:** The Scheduler applies a specific algorithm (policy) to choose the "next" process.
- **Output/Execution:** The Dispatcher takes the process chosen by the scheduler and loads its context onto the CPU.
- **Mechanism:** Context switching is the underlying operation that enables the transfer of CPU control as decided by the scheduler.

The choice of scheduling algorithm significantly impacts system performance and user experience.

The Dispatcher: The Execution Enabler

Definition: The OS Module for Execution

The **Dispatcher** is the operating system module responsible for giving control of the CPU to the process selected by the short-term scheduler. It is the final component in the scheduling hierarchy.

Key Responsibilities of the Dispatcher:

- **Context Switching:** Performs the actual saving of the old process's state and loading of the new process's state.
- **Mode Switching:** Transitions the CPU from kernel mode (where the OS runs) to user mode (where the process runs).
- **Program Jump:** Jumps to the appropriate location (Program Counter) in the newly loaded process to resume its execution.

The Dispatcher: The Execution Enabler

Definition: The OS Module for Execution

The **Dispatcher** is the operating system module responsible for giving control of the CPU to the process selected by the short-term scheduler. It is the final component in the scheduling hierarchy.

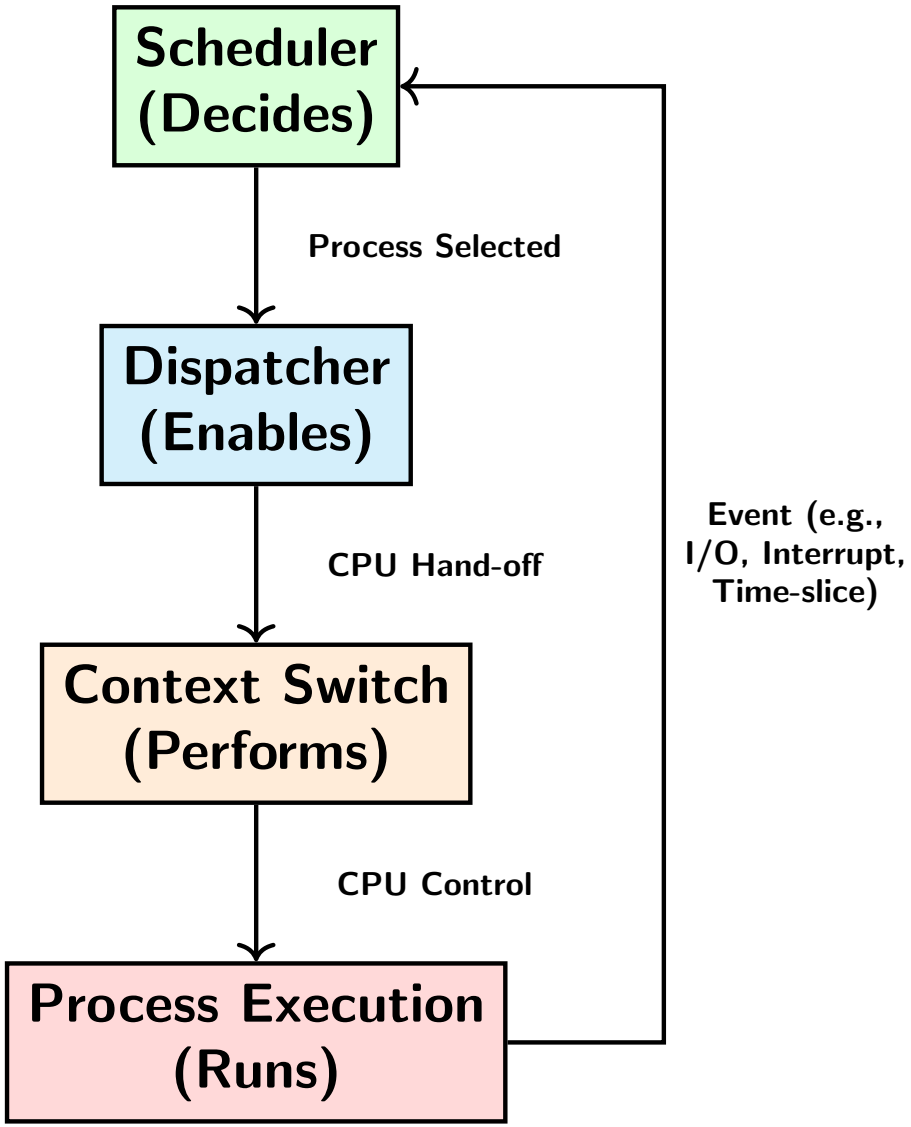
Key Responsibilities of the Dispatcher:

- **Context Switching:** Performs the actual saving of the old process's state and loading of the new process's state.
- **Mode Switching:** Transitions the CPU from kernel mode (where the OS runs) to user mode (where the process runs).
- **Program Jump:** Jumps to the appropriate location (Program Counter) in the newly loaded process to resume its execution.

Understanding Dispatch Latency

Dispatch Latency: This is the time taken by the dispatcher to stop one process and start another. It represents the overhead of context switching and is a critical factor in system responsiveness. Minimizing this latency is a key OS design goal.

The Process Scheduling Cycle: A Continuous Loop



This cycle continuously repeats, ensuring efficient CPU utilization and responsiveness across all processes.

Who Schedules the Scheduler? (The "Paradox") I

The Question Posed: "If the OS scheduler decides which process runs on the CPU, and the scheduler itself is a piece of software, doesn't it also need to be a process that gets scheduled? If so, who schedules **that** scheduler process?"

Understanding the "Paradox": Two Worlds of Execution

This question arises from assuming that **all** software execution, including the OS itself, operates under the same rules as user applications. The key to resolving this is understanding the fundamental distinction between:

- **User Mode Execution:** Where application processes run. They are limited in privileges, have their own Process Control Blocks (PCBs), and **are** explicitly scheduled by the kernel.
- **Kernel Mode Execution:** Where the Operating System kernel's code runs. It has full hardware privileges, directly manages resources, and operates fundamentally differently.

Who Schedules the Scheduler? (The "Paradox") II

Why the Scheduler is NOT a Scheduled Process:

- **It's Kernel Code, Not a Process:** The scheduler is not a separate process with its own PCB that competes for CPU time. Instead, it's a collection of **functions and routines within the kernel's privileged code**.
- **Direct Control Acquisition:** The kernel doesn't "wait for its turn." It gains control of the CPU directly and immediately when certain events occur, through a hardware mechanism (a trap to kernel mode):
  1. **System Calls:** A user process **voluntarily** requests an OS service (e.g., 'read()', 'fork()'). This instruction explicitly switches the CPU to kernel mode.
  2. **Interrupts:** Hardware events (e.g., timer expiring, I/O completion) force the CPU to stop current execution and jump into a kernel routine.
  3. **Exceptions:** Program errors (e.g., page fault, divide-by-zero) also force a switch to kernel mode for handling.

Who Schedules the Scheduler? (The "Paradox") III

- **The Arbiter, Not a Participant:** Once the CPU is operating in kernel mode, it is executing the OS's directives. If a scheduling decision is required (e.g., a time slice expired, or a process is now runnable), the kernel simply **invokes** its internal scheduler functions. It doesn't put "the scheduler" on a run queue.

Analogy: Think of the scheduler as the **CPU's operating manual** or the **air traffic controller for processes**. The manual isn't an airplane on the runway needing to be cleared for takeoff. It's the set of rules and procedures that dictate which plane takes off when. When an event happens (a plane requests takeoff, or its time slot is up), the controller (kernel code) directly applies these rules to the situation.

Context Switch Costs: More Than Just CPU Cycles

- **Direct Overhead (CPU Cycles):** The time spent by the OS kernel saving and loading CPU registers, Program Counter, Stack Pointer, and other PCB data. This is typically in the range of $1 - 10\,\mu s$ (microseconds) for modern systems.
- **Indirect Overhead (Memory & Cache):**
  - **TLB (Translation Lookaside Buffer) Flush:** When memory mapping changes (e.g., switching processes with different address spaces), the TLB, a cache for page table entries, often needs to be flushed or partially invalidated. This causes subsequent memory accesses to be slower until the TLB refills.
  - **Cache Pollution:** The new process's data and instructions may evict the old process's data from CPU caches (L1, L2, L3). This leads to increased cache misses and slower initial execution for the new process until its working set is loaded into caches.

Context Switch Costs: More Than Just CPU Cycles

- **Direct Overhead (CPU Cycles):** The time spent by the OS kernel saving and loading CPU registers, Program Counter, Stack Pointer, and other PCB data. This is typically in the range of $1 - 10\,\mu s$ (microseconds) for modern systems.
- **Indirect Overhead (Memory & Cache):**
  - **TLB (Translation Lookaside Buffer) Flush:** When memory mapping changes (e.g., switching processes with different address spaces), the TLB, a cache for page table entries, often needs to be flushed or partially invalidated. This causes subsequent memory accesses to be slower until the TLB refills.
  - **Cache Pollution:** The new process's data and instructions may evict the old process's data from CPU caches (L1, L2, L3). This leads to increased cache misses and slower initial execution for the new process until its working set is loaded into caches.

Impact of Multiprogramming

The **frequency of context switches** directly correlates with the **degree of multiprogramming**. A higher degree generally means more frequent switches, increasing overall overhead and potentially reducing useful work.

Minimizing both direct and indirect overhead is a significant challenge in OS design and a key factor in overall system performance.

Degree of Multiprogramming: Managing Concurrent Tasks

Definition

The **degree of multiprogramming** refers to the number of processes (or threads) currently loaded into main memory and actively competing for the CPU.

Why is it Important?

- **CPU Utilization:** The primary goal is to keep the CPU busy. A certain degree of multiprogramming is essential to have a ready process when the current one blocks for I/O.
- **System Throughput:** More active processes can lead to more work completed per unit of time, up to a point.
- **Responsiveness:** A higher degree of multiprogramming can improve responsiveness for interactive users by allowing quick switching between tasks.

Degree of Multiprogramming: Managing Concurrent Tasks

Definition

The **degree of multiprogramming** refers to the number of processes (or threads) currently loaded into main memory and actively competing for the CPU.

Why is it Important?

- **CPU Utilization:** The primary goal is to keep the CPU busy. A certain degree of multiprogramming is essential to have a ready process when the current one blocks for I/O.
- **System Throughput:** More active processes can lead to more work completed per unit of time, up to a point.
- **Responsiveness:** A higher degree of multiprogramming can improve responsiveness for interactive users by allowing quick switching between tasks.

The Optimization Challenge:

- The OS constantly tries to find the optimal degree of multiprogramming.
- Too low: CPU may sit idle too often (under-utilization).
- Too high: Excessive context switching overhead, leading to diminishing returns or even system degradation.

Thrashing: The Collapse of Performance

Problem Definition

Thrashing occurs when the degree of multiprogramming is too high, causing processes to spend more time paging (swapping pages between memory and disk) than executing instructions. The system becomes excessively busy swapping.

Symptoms of Thrashing

- **Extremely Low CPU Utilization:** The CPU is mostly idle, waiting for I/O (disk I/O for page swaps).
- **High Paging Activity:** Constant disk reads and writes as the OS tries to bring in required pages.
- **Slow System Response:** User applications become extremely sluggish or unresponsive.
- **Increased Context Switching:** Though not the **cause**, frequent page faults trigger more OS intervention and context switches.

Thrashing: The Collapse of Performance

Problem Definition

Thrashing occurs when the degree of multiprogramming is too high, causing processes to spend more time paging (swapping pages between memory and disk) than executing instructions. The system becomes excessively busy swapping.

Symptoms of Thrashing

- **Extremely Low CPU Utilization:** The CPU is mostly idle, waiting for I/O (disk I/O for page swaps).
- **High Paging Activity:** Constant disk reads and writes as the OS tries to bring in required pages.
- **Slow System Response:** User applications become extremely sluggish or unresponsive.
- **Increased Context Switching:** Though not the **cause**, frequent page faults trigger more OS intervention and context switches.

Key Insight

While fundamentally a **memory management problem** (specifically virtual memory and paging), thrashing is triggered by an **excessive degree of multiprogramming**. The OS tries to mitigate it by reducing the number of active processes.

Real-World Implementations: OS Specifics

Linux Kernel

- **Data Structure:** ‘task_struct‘ (Linux’s equivalent of PCB).
- **Key Functions:**
  - ‘schedule()‘: The main scheduling function, selects the next task.
  - ‘switch_to()‘: Architecture-dependent assembly function that performs the actual saving/restoring of registers.
- **Optimization:** Highly optimized, often uses processor-specific instructions (e.g., ‘iret‘, ‘sysret‘ for ring transitions) to minimize overhead.

Windows (NT Kernel)

- **Data Structure:** Thread Control Block (TCB). Windows schedules threads, not processes directly. The Process Control Block (EPROCESS) contains process-level info.
- **Key Components:**
  - **Dispatcher Object:** Handles thread ready queues and scheduling decisions.
  - **KTHREAD (Kernel Thread):** Represents the actual thread object managed by the kernel.
- **Optimization:** Uses optimized assembly routines and leverages hardware features for rapid context switching.

Real-World Implementations: OS Specifics

Linux Kernel

- **Data Structure:** ‘task_struct‘ (Linux’s equivalent of PCB).
- **Key Functions:**
  - ‘schedule()‘: The main scheduling function, selects the next task.
  - ‘switch_to()‘: Architecture-dependent assembly function that performs the actual saving/restoring of registers.
- **Optimization:** Highly optimized, often uses processor-specific instructions (e.g., ‘iret‘, ‘sysret‘ for ring transitions) to minimize overhead.

Windows (NT Kernel)

- **Data Structure:** Thread Control Block (TCB). Windows schedules threads, not processes directly. The Process Control Block (EPROCESS) contains process-level info.
- **Key Components:**
  - **Dispatcher Object:** Handles thread ready queues and scheduling decisions.
  - **KTHREAD (Kernel Thread):** Represents the actual thread object managed by the kernel.
- **Optimization:** Uses optimized assembly routines and leverages hardware features for rapid context switching.

While underlying principles are similar, specific implementations vary based on OS design philosophy and target hardware.

Practical: Observing Context Switches in Linux

Using ‘perf‘ for System Performance Analysis: ‘perf‘ is a powerful Linux profiling tool that can record and analyze various performance events, including context switches.

```
# Record context switch events system-wide for 5
    seconds
sudo perf record -e context-switches -a sleep 5

# Display the recorded data in a human-readable report
sudo perf report
```

Expected Output & Interpretation:

- ‘perf report‘ will show a list of processes and functions, along with the percentage of total context switches attributed to them.
- **What to look for:** Identify processes that are causing frequent context switches, which might indicate I/O-bound tasks, high contention, or inefficient scheduling.

Practical: Observing Context Switches in Linux

Using ‘perf‘ for System Performance Analysis: ‘perf‘ is a powerful Linux profiling tool that can record and analyze various performance events, including context switches.

```
# Record context switch events system-wide for 5
    seconds
sudo perf record -e context-switches -a sleep 5

# Display the recorded data in a human-readable report
sudo perf report
```

Expected Output & Interpretation:

- ‘perf report‘ will show a list of processes and functions, along with the percentage of total context switches attributed to them.
- **What to look for:** Identify processes that are causing frequent context switches, which might indicate I/O-bound tasks, high contention, or inefficient scheduling.

This command provides a quantitative insight into the overhead of multitasking on a live system, allowing for performance tuning.

Discussion & Trade-off Analysis

The Time Slice & Multiprogramming Dilemma

- "Is it always good for an Operating System to perform context switches very frequently?"
- "Is a higher degree of multiprogramming always better for system performance?"

Discussion & Trade-off Analysis

The Time Slice & Multiprogramming Dilemma

- "Is it always good for an Operating System to perform context switches very frequently?"
- "Is a higher degree of multiprogramming always better for system performance?"

Consider the following aspects in your discussion:

- **Responsiveness:** How does frequent vs. infrequent switching, and high vs. low multiprogramming, affect user perception and interactive applications?
- **Throughput:** What is the impact on the total amount of useful work completed by the system under varying time slices and degrees of multiprogramming?
- **Overhead:** How does context switching overhead scale with frequency? How does excessive multiprogramming lead to other forms of overhead (e.g., paging)?
- **CPU Utilization:** Is there an optimal point for both time slice and degree of multiprogramming?

Discussion & Trade-off Analysis

The Time Slice & Multiprogramming Dilemma

- "Is it always good for an Operating System to perform context switches very frequently?"
- "Is a higher degree of multiprogramming always better for system performance?"

Consider the following aspects in your discussion:

- **Responsiveness:** How does frequent vs. infrequent switching, and high vs. low multiprogramming, affect user perception and interactive applications?
- **Throughput:** What is the impact on the total amount of useful work completed by the system under varying time slices and degrees of multiprogramming?
- **Overhead:** How does context switching overhead scale with frequency? How does excessive multiprogramming lead to other forms of overhead (e.g., paging)?
- **CPU Utilization:** Is there an optimal point for both time slice and degree of multiprogramming?

The Design Challenge

In designing a general-purpose operating system, how does the OS scheduler balance the conflicting goals of maximizing CPU throughput and ensuring good system responsiveness, specifically considering the cost of context switching **and the optimal degree of multiprogramming**?

Key Takeaways

- **Context switching** is the core mechanism that allows a single CPU to handle multiple processes by rapidly saving and restoring their execution states.
- The **Dispatcher** is the kernel component responsible for the actual context switch, mode switch, and jumping to the new process's execution point.
- **Dispatch latency** and overall context switch overhead (direct and indirect, e.g., TLB/cache) are critical performance considerations in OS design.
- The **degree of multiprogramming** is a critical factor influencing system performance, affecting both CPU utilization and overhead.
- **Thrashing** is a severe performance degradation caused by an excessively high degree of multiprogramming, leading to constant paging.
- Different operating systems (like Linux and Windows) employ specific data structures and functions to optimize this crucial operation.

Key Takeaways

- **Context switching** is the core mechanism that allows a single CPU to handle multiple processes by rapidly saving and restoring their execution states.
- The **Dispatcher** is the kernel component responsible for the actual context switch, mode switch, and jumping to the new process's execution point.
- **Dispatch latency** and overall context switch overhead (direct and indirect, e.g., TLB/cache) are critical performance considerations in OS design.
- The **degree of multiprogramming** is a critical factor influencing system performance, affecting both CPU utilization and overhead.
- **Thrashing** is a severe performance degradation caused by an excessively high degree of multiprogramming, leading to constant paging.
- Different operating systems (like Linux and Windows) employ specific data structures and functions to optimize this crucial operation.

Reflection Prompt

How might the frequency of context switches directly impact the perceived "snappiness" of a user interface, particularly when the system is under heavy load? And how does the system's total RAM capacity relate to its ability to support a high degree of multiprogramming without thrashing?

Next Week Preview: CPU Scheduling Algorithms

Preparing for the Art of Resource Allocation:

- **Scheduling Criteria:** In-depth look at metrics like CPU Utilization, Throughput, Turnaround Time, Waiting Time, and Response Time.
- **Core Algorithms:** Detailed analysis of:
  - First-Come, First-Served (FCFS)
  - Shortest Job First (SJF) and Shortest Remaining Time First (SRTF)
  - Round Robin (RR)
- **Visualizing Scheduling:** Learning to use Gantt Charts for algorithm simulation and comparison.
- **Connecting to Reality:** How these algorithms implicitly or explicitly manage the degree of multiprogramming to optimize performance.

Next Week Preview: CPU Scheduling Algorithms

Preparing for the Art of Resource Allocation:

- **Scheduling Criteria:** In-depth look at metrics like CPU Utilization, Throughput, Turnaround Time, Waiting Time, and Response Time.
- **Core Algorithms:** Detailed analysis of:
  - First-Come, First-Served (FCFS)
  - Shortest Job First (SJF) and Shortest Remaining Time First (SRTF)
  - Round Robin (RR)
- **Visualizing Scheduling:** Learning to use Gantt Charts for algorithm simulation and comparison.
- **Connecting to Reality:** How these algorithms implicitly or explicitly manage the degree of multiprogramming to optimize performance.

Prep Tip for Next Session

Review basic queueing theory concepts if you're unfamiliar. Think about simple real-world scenarios where you have to prioritize tasks (e.g., supermarket checkout, restaurant orders). Also, consider how the number of available checkout counters (CPU capacity) and the number of customers (processes) affect overall waiting times.

Outline

Quick Check: Context Switching Fundamentals

Test Your Basic Understanding:

1. **True/False:** Dispatch latency is the time it takes for the scheduler to select the next process to run. Explain your answer.
2. **Multiple Choice:** Which of the following is NOT a direct responsibility of the Dispatcher?
   - (a) Performing context switch
   - (b) Selecting the next process from the Ready Queue
   - (c) Switching to user mode
   - (d) Jumping to the new process's Program Counter
3. **Fill in the Blank:** During a context switch, the _____ of the old process is saved, and the _____ of the new process is loaded.
4. **Short Answer:** Name two types of overhead associated with context switching.

Think & Discuss

Take a moment to formulate your answers. We'll discuss these briefly.

Advanced Conceptual Quiz: Performance & Design I

Apply Your Knowledge:

1. **Scenario Analysis:** A system experiences very high cache miss rates immediately after a context switch. Explain why this happens and what specific component of context switch overhead is primarily responsible.

2. **Critical Thinking:** Consider a real-time operating system (RTOS) used in an embedded system (e.g., an airbag controller). Would such a system prioritize minimizing context switch overhead or maximizing CPU throughput? Justify your answer.

3. **Process Cycle Interruption:** In the "Process Scheduling Cycle" we discussed, identify at least three different types of events that could cause the currently running process to yield the CPU and return control to the scheduler.

Advanced Conceptual Quiz: Performance & Design II

4. **Degree of Multiprogramming & Thrashing:** You observe a server that is extremely slow, with its hard drive light constantly on, even though CPU usage reported by ‘top‘ is very low.
   - What phenomenon is likely occurring?
   - How does this relate to the degree of multiprogramming?
   - What immediate action might the OS (or an administrator) take to alleviate this problem?

Collaborate & Challenge

Discuss these challenges with a peer. There might be more than one valid approach!

Exercise: Context Switch Calculation & Analysis I

Instructions: Solve the following problems, showing your calculations and reasoning.

1. **Overhead Calculation:** A system has a context switch overhead of $5\,\mu s$.
   - If the time slice (quantum) is $20\,ms$, what percentage of CPU time is spent on context switching?
   - If the time slice is reduced to $1\,ms$, what is the new percentage overhead?
   - What conclusion can you draw about the relationship between time slice duration and context switch overhead?

2. **Impact on Responsiveness:** Consider a user interacting with a text editor. If the time slice is very long (e.g., $500\,ms$), explain how this might affect the perceived responsiveness of the editor, especially when another CPU-intensive process is running in the background. Relate this to dispatch latency.

Exercise: Context Switch Calculation & Analysis II

3. **Identifying Overhead Sources:** You observe that a system's context switch time is significantly higher than the typical $1 - 10\,\mu s$ range, even for simple process switches. List three potential reasons for this increased overhead, beyond just saving/restoring registers.

Remember Unit Conversions!

Ensure all time units are consistent before performing calculations (e.g., convert everything to microseconds).

Exercise: OS Design Choices & Context Switching I

Instructions: Answer the following questions, providing detailed explanations.

1. **Thread vs. Process Context Switch:** Explain why a context switch between two threads belonging to the **same** process is generally faster than a context switch between two **different** processes. What specific PCB/TCB fields might be omitted or handled differently in the thread-to-thread switch?

2. **Hardware vs. Software Support:** Modern CPUs often provide hardware support for context switching (e.g., specific instructions or dedicated register sets). Discuss how such hardware support reduces context switch overhead compared to a purely software-managed context switch.

Exercise: OS Design Choices & Context Switching II

3. **Challenge: Optimizing Multiprogramming for Mixed Workloads:** Imagine an OS running a mix of CPU-bound (e.g., video rendering) and I/O-bound (e.g., web server) processes.
   - How would the OS attempt to set an optimal degree of multiprogramming to maximize overall system throughput?
   - What risks are involved if the degree of multiprogramming becomes too high or too low in such a mixed environment?
   - Briefly, how might a scheduler implicitly or explicitly adjust the degree of multiprogramming?

Consider the Shared Resources

Think about what processes share versus what threads share within a process. For the last question, think about how I/O operations can 'hide' CPU idleness.

Advanced Topics & Important Details I

Expanding Your Understanding of CPU Execution Management

- **Hardware Support for Context Switching:**
  - Specific CPU instructions (e.g., x86 ‘LSS‘, ‘LOADALL‘) that can load multiple registers in one go, significantly reducing context switch time.
  - Task State Segments (TSS) in x86 architecture and their role in hardware-assisted task switching.
- **Lazy Context Switching / Lazy TLB Reload:**
  - Optimizations where the OS only reloads TLB entries (or even floating-point registers) when they are actually needed by the new process, saving overhead.
- **Context Switching in Virtualized Environments (VM Exits/Entries):**
  - How hypervisors manage CPU transitions between guest OSes, which is a form of context switching at a higher level.
- **Working Set Model & Locality of Reference:**

Advanced Topics & Important Details II

- More in-depth understanding of how the working set size of processes (the set of pages actively used) influences the optimal degree of multiprogramming and relates to thrashing.

- **Admission Control:**
  - How operating systems limit the number of processes in memory to prevent thrashing and maintain system stability.

- **Kernel Stack vs. User Stack:**
  - During a context switch (or system call/interrupt), the kernel uses its own stack for the kernel's execution, distinct from the user process's stack.
- **Interrupt Descriptor Table (IDT) / Interrupt Vector Table (IVT):**
  - How the CPU finds the correct interrupt handler (kernel routine) when an interrupt (like a timer interrupt causing a context switch) occurs.
- **Race Conditions & Synchronization:**

Advanced Topics & Important Details III

- Context switching can expose race conditions if multiple processes/threads access shared resources without proper synchronization. This leads to the need for semaphores, mutexes, etc.

- **Swap Space / Paging File:**
  - The crucial role of disk swap space as an extension of RAM, especially when dealing with high multiprogramming and potential thrashing.

Interrupts & Traps - The Kernel's Entry Points

Building on "Who Schedules the Scheduler?", these mechanisms are precisely how the OS kernel gains control from user-space programs or hardware.

Interrupts: Asynchronous Hardware Events

Definition: Signals generated by hardware (external to the CPU) or other CPUs, asynchronously to the currently executing instruction. (**Purpose:** Notify the CPU of an external event that requires immediate attention.)

- **Examples:**
  - **Timer Interrupt:** A programmable timer chip signals the end of a time slice. *(Crucial for preemptive scheduling!)*
  - **I/O Completion Interrupt:** A disk controller or network card signals that a data transfer (often via DMA) is complete.
  - **Hardware Errors:** Power failure, memory parity error.
- **Mechanism:** CPU halts current execution, saves its state, and jumps to a predefined kernel routine (Interrupt Service Routine - ISR) via the Interrupt Vector Table (or Interrupt Descriptor Table on x86).

Interrupts & Traps - The Kernel's Entry Points

Traps (Exceptions): Synchronous Software Events

Definition: Synchronous events triggered by the CPU itself while executing instructions (internal to the CPU/program). Often called "software interrupts" or "exceptions." (**Purpose:** Handle abnormal program behavior or execute privileged instructions.)

- **Examples:**
  - **System Call (Software Interrupt):** A user program intentionally executes a special instruction (e.g., 'INT 80h' on x86 Linux, 'syscall' instruction) to request a kernel service.
  - **Page Fault:** A program tries to access a memory page not currently in physical RAM.
  - **Divide-by-Zero:** An arithmetic error.
  - **Invalid Instruction:** A program tries to execute a non-existent or privileged instruction in user mode.
- **Mechanism:** Similar to interrupts, CPU switches to kernel mode and executes a predefined exception handler.

Both interrupts and traps facilitate the necessary transfer of control from user mode to kernel mode, allowing the OS to intervene and manage the system.

The Timer Interrupt - Fueling Preemptive Scheduling I

The timer interrupt is arguably the single most important hardware mechanism enabling modern multi-tasking operating systems.

What is a Timer Interrupt?

- A periodic, hardware-generated interrupt.
- A programmable timer chip (e.g., HPET, APIC timer) is configured by the kernel to generate an interrupt after a specific interval (e.g., every 1-10 milliseconds, configurable as the "tick rate" or "HZ").

Why is it Critical for Preemptive Scheduling?

- Without a timer interrupt, the kernel would only gain control if a process voluntarily yielded the CPU (e.g., by making a system call or explicitly calling a 'yield' function). This is called **cooperative scheduling**.
- Cooperative scheduling is unreliable because a malicious or buggy process could run indefinitely, starving all other processes and making the system unresponsive.
- **Preemptive scheduling** (e.g., Round Robin, Priority-based with time slicing) requires the OS to be able to forcibly take the CPU away from a running process.

The Timer Interrupt - Fueling Preemptive Scheduling II

How it Works (Simplified Flow):

1. The OS sets up the hardware timer to generate interrupts periodically.
2. A user process is running on the CPU.
3. The timer "fires," sending an interrupt signal to the CPU.
4. The CPU immediately **halts the current user process**, saves its context, and switches to kernel mode.
5. The kernel's **Timer Interrupt Service Routine (ISR)** is executed.
6. The ISR performs tasks like:
   - Decrementing the time slice of the current process.
   - Updating system time.
   - Checking for expired timers (e.g., for 'sleep()' calls).
7. If the current process's time slice has expired, the ISR invokes the **CPU scheduler**.
8. The scheduler selects the next process to run, and a **context switch** occurs.
9. The CPU returns to user mode, executing the newly scheduled process.

The timer interrupt ensures that the OS regains control regularly, guaranteeing fairness and responsiveness in a multitasking environment.

DMA - Direct Memory Access and its Scheduling Impact I

While not a mechanism for the scheduler to **gain** control directly, Direct Memory Access (DMA) is fundamental to efficient I/O and plays a significant role in how processes interact with the scheduler during I/O operations.

What is DMA?

- **Definition:** A hardware feature that allows I/O devices (like disk controllers, network cards, GPUs) to transfer data directly to and from main memory **without involving the CPU**.
- **Purpose:** To offload data transfer tasks from the CPU, significantly improving system efficiency, especially for high-volume I/O operations.
- **Contrast with Programmed I/O (PIO):** In PIO, the CPU itself moves data word by word between the device controller and memory, wasting valuable CPU cycles.

How DMA Works (Simplified):

1. A process requests an I/O operation (e.g., ‘read()‘ from disk) via a system call.

DMA - Direct Memory Access and its Scheduling Impact II

2. The kernel initiates the I/O:
   - It sets up the DMA controller with the source (device buffer), destination (memory address), and size of the transfer.
   - The process making the request is often moved to a **Waiting state** and yields the CPU (voluntarily context switch).
3. The **CPU is now free** to run other processes/threads.
4. The DMA controller takes over, moving data directly between the I/O device and memory.
5. Once the DMA transfer is complete, the DMA controller generates an **I/O Completion Interrupt** to the CPU.
6. The kernel's I/O Interrupt Service Routine (ISR) is invoked.
7. The ISR:
   - Checks the status of the I/O operation.
   - Moves the waiting process that initiated the I/O to the **Ready state**.

DMA - Direct Memory Access and its Scheduling Impact III

8. The scheduler is then invoked to decide which process (potentially including the newly ready one) should run next.

Impact on Scheduling & System Performance:

- Allows the CPU to remain busy with computational tasks while I/O is happening concurrently.
- Facilitates better CPU utilization and overall system throughput.
- Enables processes to effectively **block on I/O** without wasting CPU cycles polling, making the waiting/ready state transitions meaningful for performance.

DMA is a cornerstone of modern OS efficiency, ensuring the CPU is not bottlenecked by slow I/O operations and allowing the scheduler to optimize CPU usage.

User Mode vs. Kernel Mode - The Foundation of OS Control I

The CPU's Privilege Levels:

- Modern CPUs support **multiple execution modes** or "rings" of privilege.
- The Operating System (OS) leverages these modes to enforce security and protect critical system resources.
- In Linux/Unix and Windows, typically two main modes are used:
  - **Ring 0: Kernel Mode** (or Supervisor Mode, Privileged Mode)
  - **Ring 3: User Mode** (or Restricted Mode, Unprivileged Mode)

User Mode vs. Kernel Mode - The Foundation of OS Control II

1. User Mode Execution

- **Privilege Level:** Lowest privilege.
- **Running Code:** All user applications (web browsers, word processors, games, your C programs).
- **Access Restrictions:**
  - **Limited Hardware Access:** Cannot directly access I/O devices (disk, network card, keyboard).
  - **Limited Memory Access:** Can only access its own designated virtual address space. Cannot directly read/write kernel memory or memory of other processes.
  - **Restricted Instruction Set:** Cannot execute "privileged instructions" (e.g., instructions to enable/disable interrupts, change memory mapping registers).
- **Purpose:** Provides isolation and protection. A buggy or malicious user application cannot crash the entire system or compromise other programs.

User Mode vs. Kernel Mode - The Foundation of OS Control III

2. Kernel Mode Execution

- **Privilege Level:** Highest privilege (full access).
- **Running Code:** The Operating System kernel code.
- **Full Access:**
  - **Complete Hardware Access:** Can directly control all hardware components.
  - **Full Memory Access:** Can access all of physical memory, including kernel memory and any user process's memory.
  - **Complete Instruction Set:** Can execute all CPU instructions, including privileged ones.
- **Purpose:** To perform critical system operations, manage resources, enforce security policies, and respond to hardware events.

The OS acts as a mediator, allowing user programs to request privileged operations safely.

Execution Characteristics: Process vs. Kernel Routine I

While both involve CPU execution, the nature of execution in User Mode versus Kernel Mode is fundamentally different, especially regarding how they are managed and "scheduled."

User Mode Execution: The Process

- **Unit of Execution:** A distinct **User Process**.
- **Independent Identity:** Each user process has its own **Process Control Block (PCB)** that stores its unique state, resources, and context.
- **Scheduling Unit:** User processes are the primary units that the CPU **scheduler manages and schedules**. They compete for CPU time.
- **Context:** Requires a **full context save and restore** (including registers, memory mappings, program counter) when switching between processes.
- **Memory:** Operates within its **isolated virtual address space**.
- **Lifecycle:** Has a well-defined lifecycle (New, Ready, Running, Waiting, Terminated).
- **Initiation:** Typically starts from an 'exec()' system call.

Execution Characteristics: Process vs. Kernel Routine II

Kernel Mode Execution: The Kernel Routine/Function

- **Unit of Execution:** A **routine or function within the OS kernel code**, not an independent "process" in the user-mode sense.
- **No Independent PCB (for routine):** The kernel routines themselves don't have separate PCBs that get scheduled.
- **Not Scheduled by User Scheduler:** Kernel code is **NOT scheduled** by the CPU scheduler that manages user processes. It gains control by privileged transfers (system calls, interrupts, exceptions).
- **Context:**
  - Often runs **in the context of the user process** that initiated the mode switch (e.g., during a system call).
  - Some kernel operations might occur in the context of **kernel threads** (special processes managed by the kernel for its own tasks), which **are** scheduled, but these are distinct from user processes.
- **Memory:** Has access to the **entire physical memory** and the current user process's address space.
- **Lifecycle:** Transient execution; it performs its task and then returns control to user mode (or switches to another process). It doesn't have an independent "life" like a user process.
- **Initiation:** Triggered by system calls, interrupts, or exceptions.

Key Differences in Execution & Control Transfer I

The distinction between user mode and kernel mode profoundly affects how code runs and interacts with the system.

Fundamental Differences Summarized:

- **Access to System Resources:**
  - **User Mode:** Restricted access (via OS mediation).
  - **Kernel Mode:** Direct and unrestricted access.
- **Memory Management:**
  - **User Mode:** Operates within its own isolated virtual address space.
  - **Kernel Mode:** Operates within a global kernel space (often mapped to all physical memory or with special mappings), and can access any process's memory space.
- **Instruction Set:**
  - **User Mode:** Can only execute "unprivileged" instructions.
  - **Kernel Mode:** Can execute all "privileged" and unprivileged instructions.
- **Error Handling:**

Key Differences in Execution & Control Transfer II

- **User Mode Errors:** Typically result in termination of the offending user process (e.g., segmentation fault).
- **Kernel Mode Errors:** Can lead to a system crash (Kernel Panic, Blue Screen of Death), as there's no higher privilege level to handle them safely.

How Control Transfers Between Modes: The CPU automatically switches modes when specific events occur. This is often called a "trap" to kernel mode.

1. **System Calls:**
   - **User to Kernel:** Explicit request from a user program for an OS service (e.g., 'open()', 'read()', 'fork()', 'exit()').
   - The user program executes a special "software interrupt" or 'syscall' instruction.
   - CPU detects this, saves user context, switches to kernel mode, and jumps to the appropriate system call handler.
2. **Hardware Interrupts:**
   - **Hardware to Kernel:** Asynchronous signals from devices (e.g., timer, disk controller, network card).
   - CPU immediately switches to kernel mode and executes an Interrupt Service Routine (ISR).

Key Differences in Execution & Control Transfer III

- **3. Exceptions/Traps (Hardware Errors):**
  - **Hardware to Kernel:** Synchronous events caused by program execution errors (e.g., divide-by-zero, page fault, accessing protected memory).
  - CPU switches to kernel mode and executes an exception handler.
- **Kernel to User:**
  - After processing a system call, interrupt, or exception, the kernel restores the user process's context and switches the CPU back to user mode, resuming user execution.

This controlled transfer is fundamental for maintaining system stability and security.

Practical Understanding: Observing Mode Differences I

Observing these differences firsthand can solidify your understanding. These examples are for Linux.

1. **Attempting a Privileged Operation (Causes a Crash/Fault):**
   - **Concept:** A user-mode program cannot directly access hardware registers or perform privileged I/O operations.
   - **Try it:**
     - Write a simple C program that tries to access a protected memory address (e.g., ‘*(int*)0 = 1;‘).
     - Compile and run it: ‘gcc -o crashme crashme.c‘ then ‘./crashme‘
     - **Observe:** You will likely get a "Segmentation fault" (SIGSEGV). This is the kernel intervening and terminating your process because it violated memory access rules, demonstrating memory protection enforced by CPU modes.
     - **Note:** Directly trying to execute CPU privileged instructions from C is harder as compilers won't generate them for user-mode code.
2. **Observing System Calls (Mode Switches) with ‘strace‘:**
   - **Concept:** Every time your user program needs an OS service (like opening a file, writing to screen, allocating memory), it must make a system call, which involves a mode switch to kernel mode.

Practical Understanding: Observing Mode Differences II

- **Try it:**
  - Run a simple command or program with ‘strace‘: ‘strace ls‘ or ‘strace echo ”Hello”‘
  - **Observe:** You will see a flood of system calls (‘openat‘, ‘read‘, ‘write‘, ‘execve‘, ‘mmap‘, ‘stat‘, ‘close‘, ‘exit_group‘). Each of these represents a transition from user mode to kernel mode and back.
  - This demonstrates the constant interaction and mode switching required for even basic operations.

3. **Accessing ‘/proc‘ vs. Direct Hardware Access:**

- **Concept:** User processes interact with system hardware indirectly through the OS (kernel mode), which then controls the hardware. ‘/proc‘ is a kernel-provided interface.
- **Try it:**
  - ‘cat /proc/cpuinfo‘ (This is a user-mode process reading a file that the kernel dynamically generates using its privileged access to CPU information).

Practical Understanding: Observing Mode Differences III

- **Contrast:** You cannot directly ‘mmap‘ or ‘read‘ from a physical memory address (e.g., ‘0xFED00000‘ which might be a hardware register) from user-space without specific kernel drivers or permissions. If you try, you’ll get a segmentation fault.

These hands-on exercises illustrate the fundamental protective barriers enforced by CPU modes and the OS.