Priority Scheduling and Multilevel Queues

Week 6

SDB

Autumn 2025

Agenda: Advanced CPU Scheduling Strategies

Beyond the Basics: Fairer and Finer-Grained Control

1. **Priority Scheduling** *(Prioritizing critical tasks)*
2. **Starvation & Aging** *(The challenge of fairness)*
3. **Multilevel Queue Scheduling** *(Organizing diverse workloads)*
4. **Real-World Architectures** *(How OSes actually do it)*
5. **Summary & Discussion**

Agenda: Advanced CPU Scheduling Strategies

Beyond the Basics: Fairer and Finer-Grained Control

1. **Priority Scheduling** *(Prioritizing critical tasks)*
2. **Starvation & Aging** *(The challenge of fairness)*
3. **Multilevel Queue Scheduling** *(Organizing diverse workloads)*
4. **Real-World Architectures** *(How OSes actually do it)*
5. **Summary & Discussion**

Think Ahead: Balancing Power & Fairness

How can an operating system ensure that important tasks (e.g., system processes, interactive applications) get preferential treatment, while still guaranteeing that less critical tasks (e.g., background batch jobs) eventually get their turn? What are the inherent risks of such prioritization?

Priority Scheduling: Prioritizing CPU Access I

Definition

Each process is assigned a **priority number**. The CPU is always allocated to the process with the **highest priority** among those available. *(Convention: Typically, a* **smaller** *priority number means* **higher** *priority, e.g., Priority 1 > Priority 2 > Priority 3).*

Types of Priority Scheduling:

- **Non-Preemptive:** If a process starts executing, it completes its CPU burst even if a higher-priority process arrives later.
- **Preemptive:** If a new process arrives (or an existing one becomes ready) with a higher priority than the currently running process, the CPU is immediately preempted and allocated to the higher-priority process.

Priority Scheduling: Prioritizing CPU Access II

Pros & Cons:

- **Pros:**
  - Allows critical or time-sensitive processes to execute quickly.
  - Good for real-time systems where certain tasks must meet deadlines.
- **Cons:**
  - **Starvation:** Low-priority processes may wait indefinitely if there's a continuous stream of higher-priority processes.
  - Can lead to unpredictable system behavior if not managed carefully (e.g., a critical but lower-priority background task might be delayed).

Preemptive Priority Scheduling: Example & Analysis I

Processes (Arrival Time, Burst Time, Priority):

- P1: (0, 5, 3) *(Arrives at 0, needs 5 units, Priority 3)*
- P2: (1, 3, 1) *(Arrives at 1, needs 3 units, Priority 1 - HIGHEST)*
- P3: (2, 4, 2) *(Arrives at 2, needs 4 units, Priority 2)*

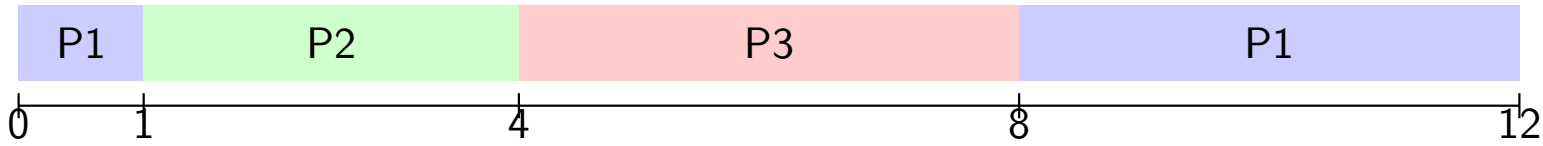
(Convention: Lower priority number = Higher priority)

Execution Trace & Gantt Chart Construction:

- **Time 0:** P1 arrives, starts executing (P1 remaining = 5).
- **Time 1:** P2 arrives (Priority 1). P2 (Prio 1) ¿ P1 (Prio 3). P2 preempts P1. P1 suspended.
- **Time 1-4:** P2 executes for its full burst (3 units). P2 completes at Time 4.
- **Time 4:** P1 (remaining 4, Prio 3) and P3 (arrived at 2, burst 4, Prio 2) are ready. P3 (Prio 2) ¿ P1 (Prio 3). P3 selected.
- **Time 4-8:** P3 executes for its full burst (4 units). P3 completes at Time 8.
- **Time 8:** Only P1 (remaining 4, Prio 3) is ready. P1 resumes.
- **Time 8-12:** P1 executes for its remaining burst (4 units). P1 completes at Time 12.

Preemptive Priority Scheduling: Example & Analysis II

Gantt Chart (Execution Order: P1 → P2 → P3 → P1):



Performance Metrics Calculation:

Process	AT	BT	Prio	CT	TAT (CT-AT)	WT (TAT-BT)	RT (First Start-AT)
P1	0	5	3	12	12 - 0 = **12**	12 - 5 = **7**	0 - 0 = **0**
P2	1	3	1	4	4 - 1 = **3**	3 - 3 = **0**	1 - 1 = **0**
P3	2	4	2	8	8 - 2 = **6**	6 - 4 = **2**	4 - 2 = **2**

Average Turnaround Time: $(12 + 3 + 6)/3 = \mathbf{7.0}$ units
Average Waiting Time: $(7 + 0 + 2)/3 = \mathbf{3.0}$ units
Average Response Time: $(0 + 0 + 2)/3 = \mathbf{0.67}$ units

Starvation and Aging: Addressing Priority's Flaw I

Starvation (Indefinite Blocking):

- Occurs when a low-priority process continuously waits for the CPU because higher-priority processes keep arriving or becoming ready.
- The low-priority process may never get to execute, even if it's runnable, essentially being "starved" of resources.
- A critical problem in systems relying heavily on fixed-priority scheduling.

Aging (The Solution):

- A technique to prevent starvation by **gradually increasing the priority of processes that wait in the Ready Queue for a long time**.
- As time passes, a waiting process's priority steadily improves, eventually reaching a point where it becomes the highest priority and gets scheduled.
- This ensures that even low-priority processes eventually get a chance to run, preventing indefinite blocking.

Starvation and Aging: Addressing Priority's Flaw II

Aging Formula (Conceptual Example):

$$\text{New Priority} = \text{Current Priority} - (\alpha \times \text{Time in Ready Queue})$$

Where α (alpha) is the aging rate (a constant determining how quickly priority increases). (Assumes lower number = higher priority. If higher number = higher priority, then it would be '+' instead of '-'.)

Multilevel Queue Scheduling: Tailoring Scheduling to Process Types I

Motivation:

- Not all processes are alike; they have different characteristics (e.g., interactive vs. batch) and different scheduling needs.
- A single scheduling algorithm often isn't optimal for all types of processes.

Concept:

- Processes are permanently partitioned into different **separate queues** based on their characteristics or type.
- Each queue typically has its **own specific scheduling algorithm** (e.g., RR for foreground, FCFS for background).
- No process ever moves between queues.

Multilevel Queue Scheduling: Tailoring Scheduling to Process Types II

Typical Queue Classifications:

- **System Processes:** OS tasks, highest priority.
- **Interactive Processes:** Foreground applications, respond quickly to user input (e.g., text editor).
- **Interactive Batch Processes:** Background, but still need some interactivity (e.g., database queries).
- **Batch Processes:** Long-running, non-interactive (e.g., compilers, scientific simulations).

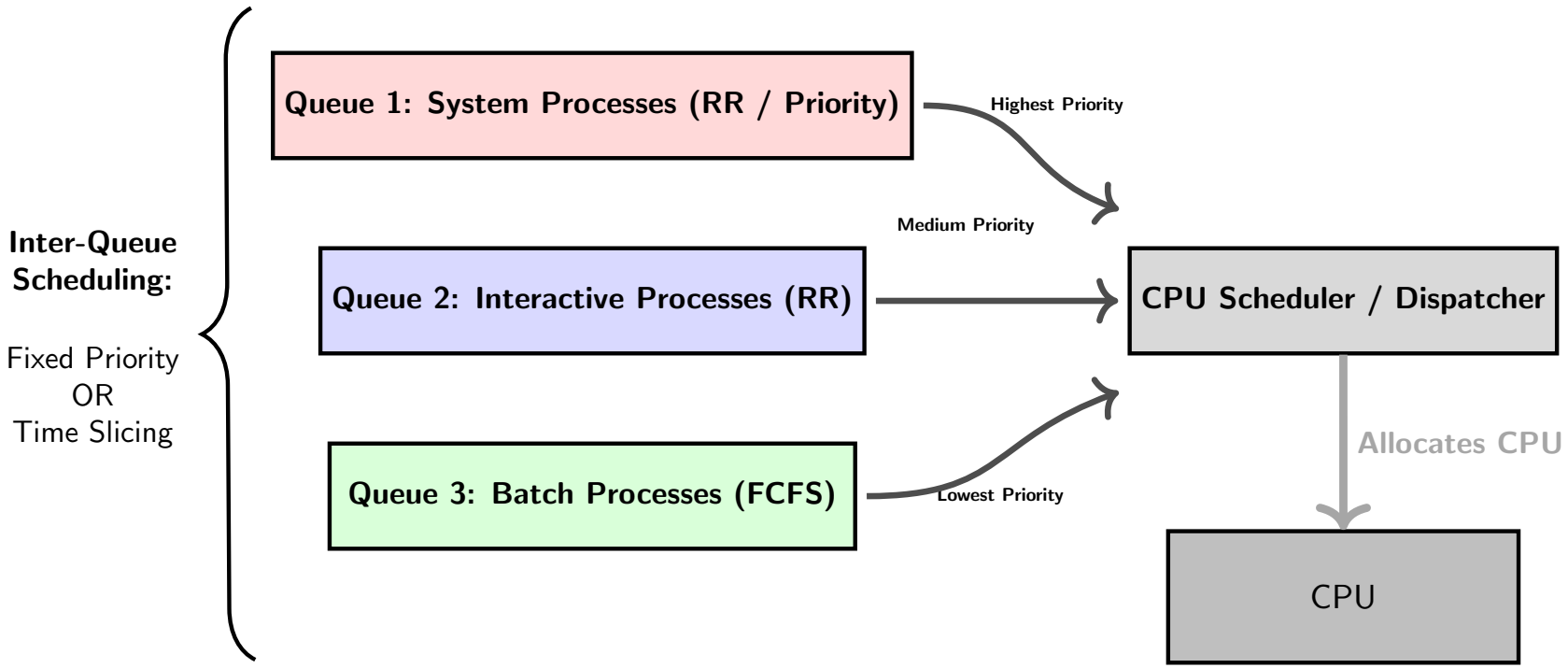
Pros & Cons:

- **Pros:** Tailored scheduling for different process types, better overall system performance.
- **Cons:** Potential for starvation of lower-priority queues (if not managed by time slicing between queues).

Multilevel Queue (MLQ) Scheduler (Illustration)

Static Queue Allocation

In a Multilevel Queue scheduler, processes are permanently assigned to a queue upon entry. They cannot move between queues. Each queue may have its own scheduling algorithm.



Multilevel Queue: Inter-Queue Scheduling Methods

How the CPU is allocated among the different queues:

- **Fixed-Priority Preemptive Scheduling:**
  - Each queue has an absolute priority over lower-priority queues.
  - A process in a higher-priority queue will always be executed before any process in a lower-priority queue.
  - **Risk:** Can lead to **starvation** for processes in low-priority queues if higher-priority queues are continuously busy.
- **Time Slicing Between Queues:**
  - Each queue receives a certain percentage of the CPU time.
  - For example, the interactive queue might get 80% of CPU time, and the batch queue gets 20%.
  - Within its allocated time slice, each queue can then schedule its processes using its own algorithm.
  - **Benefit:** Ensures that lower-priority queues still get some CPU time, preventing starvation and providing better fairness.

Multilevel Feedback Queue Scheduling: Adapting to Process Behavior I

Motivation:

- A process's behavior can change over time (e.g., an interactive job can become CPU-bound).
- The static nature of Multilevel Queue (MLQ) scheduling can lead to inefficient scheduling and starvation.

Concept:

- Processes are initially placed in a high-priority queue.
- The key difference from MLQ is the **feedback mechanism**: a process can move between queues based on its behavior.
- MLFQ aims to separate processes into CPU-bound (long-running) and I/O-bound (interactive) types automatically.

Multilevel Feedback Queue Scheduling: Adapting to Process Behavior II

Core Rules of MLFQ:

- **Rule 1:** If a job in a higher-priority queue is ready, it runs first.
- **Rule 2:** Jobs within the same queue are scheduled using a Round Robin algorithm.
- **Rule 3:** A new job enters the highest-priority queue.
- **Rule 4:** If a job uses its entire time quantum, it is **demoted** to the next lower-priority queue.
- **Rule 5:** If a job gives up the CPU before its quantum is finished (e.g., for I/O), its priority may remain the same or be **promoted**.

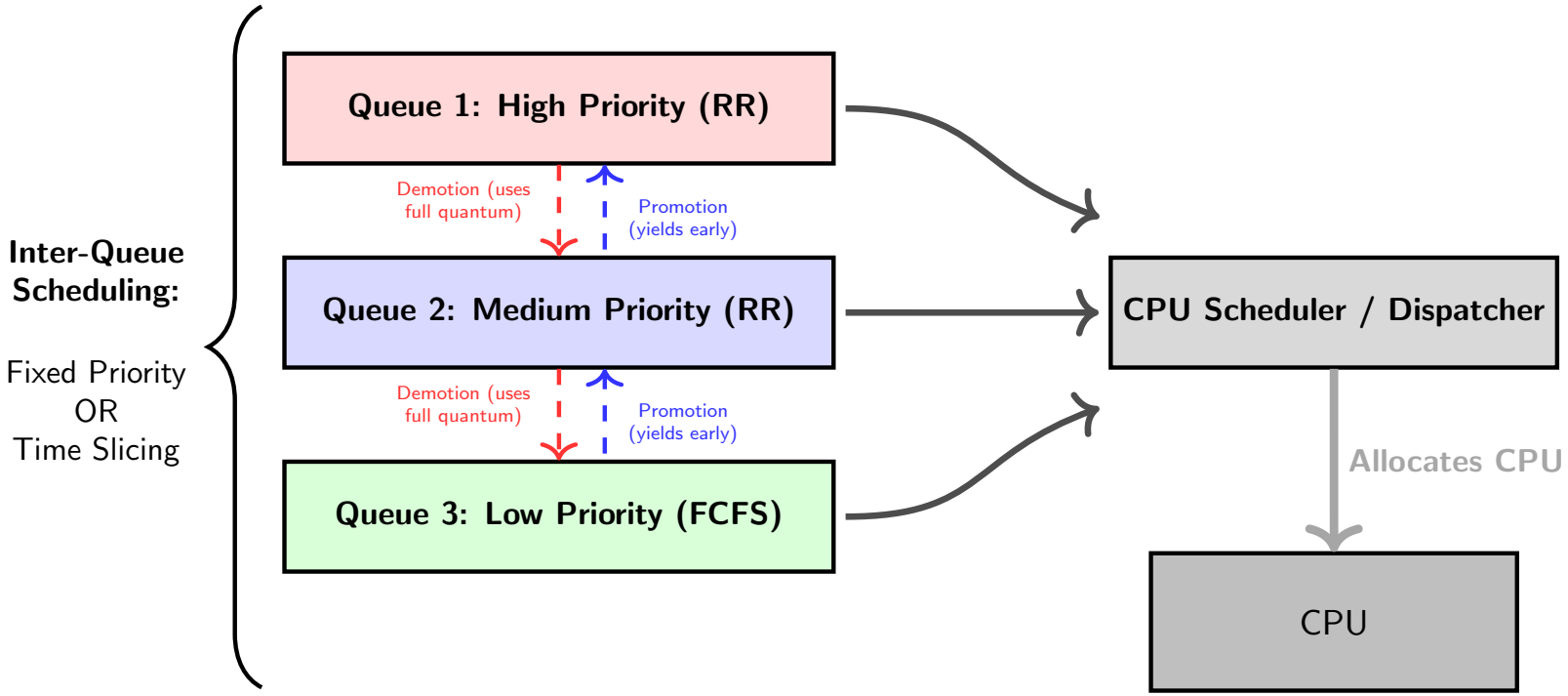
Pros & Cons:

- **Pros:** Excellent for interactive jobs, prevents starvation by demoting CPU-intensive processes, dynamically adapts.
- **Cons:** Complex to implement, can be "gamed" by processes that manipulate their behavior.

MLFQ: The Role of Feedback (Illustration)

Dynamic Queue Movement

MLFQ builds on the Multilevel Queue concept by allowing processes to move between queues. This "feedback" is crucial for dynamically adapting to process behavior.



MLQ vs. MLFQ: A Quick Comparison

The Key Difference

The main distinction lies in the ability of processes to change queues.

MLQ vs. MLFQ: A Quick Comparison

The Key Difference

The main distinction lies in the ability of processes to change queues.

Feature	**Multilevel Queue (MLQ)**	**Multilevel Feedback Queue (MLFQ)**
Queue Movement	**Static:** Processes are permanently assigned to a queue.	**Dynamic:** Processes can move between queues (feedback).
Flexibility	Less flexible. Cannot adapt to changing process behavior.	Highly flexible. Adapts to prioritize interactive vs. CPU-bound jobs.
Starvation	A low-priority queue may starve if high-priority queues are always busy.	Reduces starvation by demoting long-running processes, giving others a chance.

Linux and Windows Schedulers I

Linux CFS (Completely Fair Scheduler)

- **Core Philosophy:** Aims for perfect fairness by giving each process an equal share of the processor's time, rather than relying on strict priority levels.
- **Mechanism:** The scheduler uses a **virtual runtime (`vruntime`)** counter for each process. This value is a normalized measure of a process's CPU time, weighted by its priority (nice value). A lower nice value (higher priority) causes `vruntime` to advance more slowly.
- **Implementation:** All runnable processes are stored in a **red-black tree**, a self-balancing data structure. The scheduler always selects the leftmost node (the one with the lowest `vruntime`) to run next, ensuring that processes that have "fallen behind" in CPU time get their turn.
- **Key Influencing Factors:**
  - `vruntime` (**virtual runtime**): The primary factor; the process with the lowest `vruntime` is always scheduled next.
  - **Priority (nice value)**: A lower nice value (higher priority) causes `vruntime` to advance more slowly, effectively giving the process more CPU time.
  - **I/O Wait Time**: Tasks that spend a lot of time waiting for I/O have a low `vruntime` and are thus heavily prioritized when they become runnable.

Linux and Windows Schedulers II

Windows OS Scheduler

- **Core Philosophy:** A **preemptive, priority-based** approach that prioritizes responsiveness and provides predictable performance for applications.
- **Mechanism:** It uses a **multilevel queue** with 32 priority levels. The highest 16 are for real-time threads, and the lower 16 are for dynamic (variable) threads. The scheduler always runs the highest-priority thread that is ready.
- **Responsiveness:** The scheduler employs a mechanism called **priority boosting**. A thread that has been awakened from a wait state (e.g., waiting for user input or I/O) receives a temporary boost in priority to ensure it runs immediately, making the system feel highly responsive.
- **Key Influencing Factors:**
  - **Priority Level**: Processes are assigned one of 32 priority levels; higher-priority threads preempt lower-priority ones.
  - **I/O Events**: Threads waiting on I/O completion receive a temporary **priority boost** to ensure they run immediately upon receiving data, making the system feel fast.
  - **Foreground Status**: The active, user-facing application receives a higher priority boost than background applications, which is a major factor in its scheduling.

Mobile and Desktop Schedulers I

Android (modified Linux kernel)

- **Foundation:** Android is built on the Linux kernel, so its scheduler is an extension of the **Completely Fair Scheduler (CFS)**.
- **Mobile Optimizations:** The scheduler is augmented with mobile-specific logic to manage power and user experience. It categorizes processes into groups like **foreground**, **background**, and **real-time**.
- **Key Differentiator: Foreground tasks** (the app currently visible to the user) are assigned a much higher priority than background tasks. This ensures that the active application remains smooth and responsive while conserving power by throttling background processes.
- **Key Influencing Factors:**
  - **Energy Usage**: A critical factor. The scheduler uses **Energy Aware Scheduling (EAS)** to make decisions, preferring to run tasks on more efficient cores (big.LITTLE architecture).
  - **App State**: An app's state (e.g., **foreground** vs. **background**) is the primary factor. Foreground tasks receive higher priority and resources.
  - **Power-Saving Modes**: When the device is in Doze mode or has a low battery, the scheduler severely restricts background activity and network access to extend battery life.

Mobile and Desktop Schedulers II

macOS (Darwin kernel)

- **Foundation:** The macOS scheduler is a **preemptive, multilevel feedback queue** scheduler that is part of the Darwin kernel.
- **Key Features:** It is highly tuned for desktop and laptop environments, balancing power efficiency with a smooth, interactive user interface. It assigns processes to queues based on their priority and how much CPU time they've consumed.
- **Concurrency API:** It provides a high-level API called **Grand Central Dispatch (GCD)**. GCD allows developers to define blocks of code to run concurrently without manually managing threads, abstracting away much of the underlying scheduling complexity.
- **Key Influencing Factors:**
  - **Interactivity**: User-facing applications and interactive events are given a high priority to maintain a smooth user experience.
  - **Power Efficiency**: The scheduler makes decisions to maximize battery life, for example, by consolidating tasks and scheduling them on lower-power cores when possible.
  - **Task Type**: Tasks are classified by APIs like **Grand Central Dispatch (GCD)**, allowing the scheduler to prioritize work based on whether it's user-initiated, background, or real-time.

Summary: Real-World Schedulers Comparison

A comparison of modern scheduling philosophies:

OS	Core Model	Primary Goal
Linux (CFS)	Fair-share scheduler	Aims for **perfect fairness** among all processes.
Windows	Preemptive, priority-based	Prioritizes **responsiveness** for interactive tasks.
Android	Modified CFS	Optimizes for **power and user experience** on mobile.
macOS	Multilevel feedback queue	Balances **interactivity and power efficiency** on desktop.

Discussion Prompt: Designing for Performance & Fairness

Scenario: You are designing the CPU scheduler for a new general-purpose operating system (like a desktop OS).

Consider these questions:

- **Question 1:** Should user-level I/O-heavy applications (e.g., web browser, music player) be placed in higher priority queues compared to CPU-heavy applications (e.g., video encoder, large compilation)? Why or why not?
  - *Think about user perceived responsiveness vs. overall CPU utilization.*
- **Question 2:** Is starvation ever acceptable for certain types of background jobs? Under what circumstances might it be tolerable, and when is it a critical failure?
- **Question 3:** How would you balance the benefits of strict priority for critical system tasks with the need for fair CPU allocation to all user processes? What mechanisms would you employ?

Summary: Advanced Scheduling Insights

Key Takeaways from Week 6:

- **Priority Scheduling** offers fine-grained control over CPU allocation, but its inherent risk is **starvation** for lower-priority tasks.
- **Aging** is a crucial mechanism to prevent starvation by dynamically increasing the priority of waiting processes.
- **Multilevel Queue Scheduling** organizes processes into different classes, allowing for specialized scheduling strategies per class, enhancing system efficiency.
- Modern OS schedulers (e.g., Linux CFS, Windows) are complex, hybrid systems that combine elements of priority, time-sharing, and feedback mechanisms to optimize for various goals like fairness, responsiveness, and throughput.

Summary: Advanced Scheduling Insights

Key Takeaways from Week 6:

- **Priority Scheduling** offers fine-grained control over CPU allocation, but its inherent risk is **starvation** for lower-priority tasks.
- **Aging** is a crucial mechanism to prevent starvation by dynamically increasing the priority of waiting processes.
- **Multilevel Queue Scheduling** organizes processes into different classes, allowing for specialized scheduling strategies per class, enhancing system efficiency.
- Modern OS schedulers (e.g., Linux CFS, Windows) are complex, hybrid systems that combine elements of priority, time-sharing, and feedback mechanisms to optimize for various goals like fairness, responsiveness, and throughput.

Reflection Prompt

Consider a scenario where a critical, high-priority system process suddenly enters an infinite loop, consuming 100% CPU. How would different scheduling algorithms (FCFS, RR, Priority with/without aging) behave? What would be the impact on other processes and the overall system?

Next Week Preview: Real-Time Systems and Advanced Schedulers

Looking Ahead: Deeper Dives into Specialized Scheduling

- **Real-Time Scheduling Algorithms:** Detailed look into Rate Monotonic (RM) and Earliest Deadline First (EDF) and their applicability.
- **Linux CFS Internals:** A more in-depth exploration of how the Completely Fair Scheduler works, including its red-black tree implementation and virtual runtime concept.
- **Distributed Scheduling Concepts:** Brief overview of challenges and approaches in multi-node environments.
- **Comprehensive CPU Scheduling Revision:** Consolidating all concepts learned from Week 5 & 6.

Next Week Preview: Real-Time Systems and Advanced Schedulers

Looking Ahead: Deeper Dives into Specialized Scheduling

- **Real-Time Scheduling Algorithms:** Detailed look into Rate Monotonic (RM) and Earliest Deadline First (EDF) and their applicability.
- **Linux CFS Internals:** A more in-depth exploration of how the Completely Fair Scheduler works, including its red-black tree implementation and virtual runtime concept.
- **Distributed Scheduling Concepts:** Brief overview of challenges and approaches in multi-node environments.
- **Comprehensive CPU Scheduling Revision:** Consolidating all concepts learned from Week 5 & 6.

Prep Tip for Next Session

Review your understanding of preemptive scheduling and the various scheduling criteria (especially response time and turnaround time). Think about how strict deadlines might change scheduling priorities.

Outline

Quick Quiz: Advanced Scheduling Concepts I

Test Your Conceptual Understanding:

1. **Scenario:** In a preemptive priority scheduling system where lower numbers mean higher priority, Process A (Prio 5, Burst 10) is running. Process B (Prio 2, Burst 8) arrives. What happens?
   - What is the term for this action?
   - What specific mechanism allows this to happen?

2. **True/False:** Multilevel Queue Scheduling automatically prevents starvation of low-priority queues. Justify your answer.

3. **Define:** How does "aging" specifically address the problem of starvation? Provide a simple, conceptual example.

4. **Application:** Why would a real-time operating system (RTOS) for an airplane's control system prefer a fixed-priority scheduling method over Round Robin?

Quick Quiz: Advanced Scheduling Concepts II

Think & Discuss

Formulate your answers before reviewing the solutions or discussing with peers.

Exercise: Priority Scheduling & MLQ Scenarios I

Part 1: Non-Preemptive Priority Calculation
Processes (Arrival Time, Burst Time, Priority): (Lower priority number = Higher priority)

- P1: (0, 7, 4)
- P2: (1, 3, 2)
- P3: (2, 5, 3)
- P4: (3, 2, 1)

Task: Draw the Gantt chart for **Non-Preemptive Priority** scheduling. Calculate the Completion Time, Turnaround Time, and Waiting Time for each process. Then, calculate the average Turnaround Time and average Waiting Time.

Part 2: Multilevel Queue Scenario
Imagine a system with two queues:

- **Queue 1 (High Priority):** For Interactive processes (uses Round Robin, Quantum=2)

Exercise: Priority Scheduling & MLQ Scenarios II

- **Queue 2 (Low Priority):** For Batch processes (uses FCFS)

Inter-Queue Scheduling: Queue 1 has absolute fixed priority over Queue 2.
Processes:

- Interactive P_A: Arrival=0, Burst=5
- Interactive P_B: Arrival=1, Burst=3
- Batch P_C: Arrival=0, Burst=10

Task:

1. Describe the execution flow of these processes. Which process runs first? When does it get preempted/blocked?
2. Will Batch P_C ever execute if P_A and P_B continuously generate new interactive tasks (assume they finish and re-enter Queue 1)? Explain your reasoning.
3. How could the inter-queue scheduling be changed to ensure Batch P_C eventually runs, without completely sacrificing interactive responsiveness?

Exercise: Priority Scheduling & MLQ Scenarios III

Reminder

Remember to trace time step-by-step, especially when preemption or new arrivals occur.

Aging Formula Variations I

Beyond the basic linear model, here are other conceptual variations for implementing aging to prevent starvation.

Variation 1: Simple Incremental Aging

- **Formula (Conceptual):**

$$\text{New Priority} = \text{Current Priority} - \text{Constant}$$

(Applied every T time units to all processes in the ready queue.)

- **Description:** Instead of a continuous update, the kernel periodically scans the ready queue and boosts the priority of every waiting process by a fixed value. This is simpler to implement but might not be as fine-grained as the continuous formula.

Aging Formula Variations II

Variation 2: Non-Linear Aging

- **Formula (Conceptual):**

$$\text{New Priority} = \text{Current Priority} - (\alpha \times \text{Time in Ready Queue}^2)$$

- **Description:** This approach gives a much larger priority boost to a process that has been waiting for a very long time. By using a squared term (or other non-linear function), the priority of a process increases at an accelerating rate the longer it waits, ensuring it eventually runs.

Aging Formula Variations III

Variation 3: Dynamic Priority Based on Behavior

- **Formula (Conceptual):**

  New Priority = Base Priority$\pm$Adjustment for recent CPU/I/O usage

- **Description:** In more advanced schedulers, the priority is not only adjusted for waiting time but also based on the process's recent behavior. Interactive (I/O-bound) processes are rewarded with a higher priority, while CPU-bound processes are penalized. This balances responsiveness and efficiency.

Advanced Topics to Explore: Deeper Dives in CPU Scheduling I

Expanding Your Horizon Beyond This Week's Fundamentals

I. Real-Time System Scheduling (RTOS)

- **Rate Monotonic Scheduling (RMS):** A static-priority, preemptive algorithm for periodic tasks, where priority is assigned based on the inverse of the period (shorter period = higher priority).
- **Earliest Deadline First (EDF):** A dynamic-priority, preemptive algorithm where the process with the earliest absolute deadline is scheduled next. (Optimal but more complex).
- **Priority Inversion:** A critical problem in RTOS where a high-priority task gets blocked by a lower-priority task, and solutions like Priority Inheritance Protocol.

Advanced Topics to Explore: Deeper Dives in CPU Scheduling II

II. Advanced Multi-Core & Modern OS Schedulers

- **Linux Completely Fair Scheduler (CFS) Deep Dive:** Understanding its core principles (red-black tree for run queue, ‘vruntime‘ concept, group scheduling).
- **Windows Scheduler Internals:** Exploring its 32-level priority scheme, quantum adjustments, and how it handles different types of threads (e.g., foreground vs. background, I/O priority).
- **Processor Affinity & NUMA Awareness:** How schedulers try to keep processes on the same CPU or within the same NUMA node to improve cache performance and memory access.
- **Load Balancing Strategies:** Techniques used to distribute workload evenly across multiple CPU cores (e.g., push migration, pull migration).

Advanced Topics to Explore: Deeper Dives in CPU Scheduling III

III. Other Related Scheduling Concepts

- **I/O Scheduling / Disk Scheduling:** Algorithms like FCFS, SSTF (Shortest Seek Time First), SCAN, C-SCAN, LOOK, C-LOOK for optimizing disk head movement.
- **Energy-Aware Scheduling:** Techniques used in mobile and embedded systems to balance performance with power consumption.
- **Thread Scheduling vs. Process Scheduling:** Understanding the nuances when scheduling threads within processes, especially kernel-level vs. user-level threads.