Real-Time Scheduling and Case Studies

Week 7

SDB

Autumn 2025

Agenda: Real-Time & Modern OS Scheduling

Beyond General Purpose: Predictability and Fairness in Complex Systems

1. **Real-Time Scheduling Concepts** *(Guaranteed responsiveness)*
2. **Rate Monotonic Scheduling (RMS)** *(Static priorities for deadlines)*
3. **Earliest Deadline First (EDF)** *(Dynamic priorities for deadlines)*
4. **Linux: Completely Fair Scheduler (CFS)** *(Fairness at scale)*
5. **Windows Scheduler & Dispatching** *(Responsiveness and priority management)*
6. **Comparative Discussion & Hands-on**

Agenda: Real-Time & Modern OS Scheduling

Beyond General Purpose: Predictability and Fairness in Complex Systems

1. **Real-Time Scheduling Concepts** *(Guaranteed responsiveness)*
2. **Rate Monotonic Scheduling (RMS)** *(Static priorities for deadlines)*
3. **Earliest Deadline First (EDF)** *(Dynamic priorities for deadlines)*
4. **Linux: Completely Fair Scheduler (CFS)** *(Fairness at scale)*
5. **Windows Scheduler & Dispatching** *(Responsiveness and priority management)*
6. **Comparative Discussion & Hands-on**

Think Ahead: When Predictability is Paramount

For systems like aircraft control or medical devices, missing a deadline is not an option. How do scheduling algorithms guarantee task completion within strict time constraints? And how do general-purpose OSes balance fairness for hundreds of processes with responsiveness for interactive users?

What is Real-Time Scheduling? Meeting Deadlines I

Concept: A scheduler that guarantees tasks are completed within specific time constraints (deadlines). Crucial for systems where timeliness is as important as correctness.

Characteristics:

- **Determinism:** Predictable behavior; task execution times are known or bounded.
- **Predictability:** Ensures that tasks will meet their deadlines consistently.
- **Responsiveness:** Quick reaction to external events or internal triggers.
- **Reliability:** High availability and fault tolerance.

What is Real-Time Scheduling? Meeting Deadlines II

Types of Real-Time Systems:

- **Hard Real-Time Systems:**
  - **Definition:** Missing a deadline leads to catastrophic failure (e.g., flight control, pacemaker).
  - Requires absolute guarantees; often uses highly specialized OSes (RTOS).
- **Soft Real-Time Systems:**
  - **Definition:** Missing a deadline is undesirable but not catastrophic; performance degrades.
  - *Examples:* Multimedia streaming, online gaming, often achieved with general-purpose OSes with real-time extensions.

Rate Monotonic Scheduling (RMS): Static Priority for Real-Time I

Rule: Static-Priority Scheduling

RMS assigns **static priorities** to periodic tasks based on their frequency (rate): **Shorter period** $\Longrightarrow$ **Higher priority.** *Once a task is assigned a priority, it remains fixed throughout its execution.*

Assumptions for Basic RMS:

- Tasks are periodic and independent (no resource sharing).
- CPU burst time is constant for each task.
- Deadlines are at the end of each period.
- Context switch time is negligible.

Rate Monotonic Scheduling (RMS): Static Priority for Real-Time II

Schedulability Test (Liu & Layland Bound): For a set of n independent periodic tasks, RMS can guarantee schedulability if their total CPU utilization U is below a specific bound:

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq n(2^{1/n} - 1)$$

Where C_i is the CPU burst time of task i, and T_i is the period of task i. As $n \to \infty$, the bound approaches $\ln(2) \approx 0.693$ *(or 69.3%).*

Pros & Cons:

- **Pros:** Simple to implement, static priorities reduce runtime overhead, good for predictable, fixed workloads.
- **Cons:** CPU utilization bound is pessimistic (can fail to schedule even if total utilization $< 100\%$), not suitable for aperiodic tasks or shared resources without extensions.

RMS Example: Schedulability and Execution

Tasks: (Execution Time C_i, Period T_i)

- Task 1 (P1): $C_1 = 1$, $T_1 = 4$
- Task 2 (P2): $C_2 = 2$, $T_2 = 6$

RMS Example: Schedulability and Execution

Tasks: (Execution Time C_i, Period T_i)

- Task 1 (P1): $C_1 = 1$, $T_1 = 4$
- Task 2 (P2): $C_2 = 2$, $T_2 = 6$

Step 1: Calculate Utilization (U)

$$U = \frac{C_1}{T_1} + \frac{C_2}{T_2} = \frac{1}{4} + \frac{2}{6} = 0.25 + 0.333 = \mathbf{0.583}$$

RMS Example: Schedulability and Execution

Tasks: (Execution Time C_i, Period T_i)

- Task 1 (P1): $C_1 = 1$, $T_1 = 4$
- Task 2 (P2): $C_2 = 2$, $T_2 = 6$

Step 1: Calculate Utilization (U)

$$U = \frac{C_1}{T_1} + \frac{C_2}{T_2} = \frac{1}{4} + \frac{2}{6} = 0.25 + 0.333 = \mathbf{0.583}$$

Step 2: Apply RMS Schedulability Test ($n = 2$)

$$n(2^{1/n} - 1) = 2(2^{1/2} - 1) = 2(\sqrt{2} - 1) = 2(1.414 - 1) = 2(0.414) = \mathbf{0.828}$$

RMS Example: Schedulability and Execution

Tasks: (Execution Time C_i, Period T_i)

- Task 1 (P1): $C_1 = 1$, $T_1 = 4$
- Task 2 (P2): $C_2 = 2$, $T_2 = 6$

Step 1: Calculate Utilization (U)

$$U = \frac{C_1}{T_1} + \frac{C_2}{T_2} = \frac{1}{4} + \frac{2}{6} = 0.25 + 0.333 = \mathbf{0.583}$$

Step 2: Apply RMS Schedulability Test ($n = 2$)

$$n(2^{1/n} - 1) = 2(2^{1/2} - 1) = 2(\sqrt{2} - 1) = 2(1.414 - 1) = 2(0.414) = \mathbf{0.828}$$

Step 3: Compare Utilization to Bound $U = 0.583 \leq 0.828$. Since U is within the bound, the task set is **schedulable by RMS**.

RMS Example: Schedulability and Execution

Tasks: (Execution Time C_i, Period T_i)

- Task 1 (P1): $C_1 = 1$, $T_1 = 4$
- Task 2 (P2): $C_2 = 2$, $T_2 = 6$

Step 1: Calculate Utilization (U)

$$U = \frac{C_1}{T_1} + \frac{C_2}{T_2} = \frac{1}{4} + \frac{2}{6} = 0.25 + 0.333 = \mathbf{0.583}$$

Step 2: Apply RMS Schedulability Test ($n = 2$)

$$n(2^{1/n} - 1) = 2(2^{1/2} - 1) = 2(\sqrt{2} - 1) = 2(1.414 - 1) = 2(0.414) = \mathbf{0.828}$$

Step 3: Compare Utilization to Bound $U = 0.583 \leq 0.828$. Since U is within the bound, the task set is **schedulable by RMS**.

Step 4: Determine Priorities

- P1 Period = 4, P2 Period = 6.
- **P1 has higher priority** (shorter period).

RMS Example: Schedulability and Execution

Tasks: (Execution Time C_i, Period T_i)

- Task 1 (P1): $C_1 = 1$, $T_1 = 4$
- Task 2 (P2): $C_2 = 2$, $T_2 = 6$

Step 1: Calculate Utilization (U)

$$U = \frac{C_1}{T_1} + \frac{C_2}{T_2} = \frac{1}{4} + \frac{2}{6} = 0.25 + 0.333 = \mathbf{0.583}$$

Step 2: Apply RMS Schedulability Test ($n = 2$)

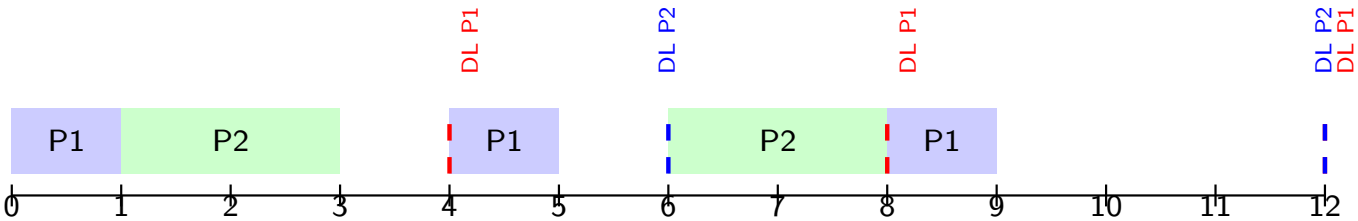
$$n(2^{1/n} - 1) = 2(2^{1/2} - 1) = 2(\sqrt{2} - 1) = 2(1.414 - 1) = 2(0.414) = \mathbf{0.828}$$

Step 3: Compare Utilization to Bound $U = 0.583 \leq 0.828$. Since U is within the bound, the task set is **schedulable by RMS**.

Step 4: Determine Priorities

- P1 Period = 4, P2 Period = 6.
- **P1 has higher priority** (shorter period).

Step 5: Conceptual Execution (Gantt Chart) *(Assume tasks arrive at $t = 0$ for their first period, then every T_i thereafter)*



All deadlines are met.

Earliest Deadline First (EDF): Dynamic Priority for Real-Time I

Concept:

- **Dynamic Priority:** Priorities are re-calculated at each scheduling point (arrival, completion, preemption).
- The process with the **earliest absolute deadline** (arrival time + relative deadline) runs first.
- If two tasks have the same deadline, FCFS is typically used as a tie-breaker.

Assumptions for Basic EDF:

- Tasks are periodic or aperiodic with known execution times and deadlines.
- Deadlines are usually equal to or less than the period.
- Context switch time is negligible.

Earliest Deadline First (EDF): Dynamic Priority for Real-Time II

Schedulability Test: For a set of n independent periodic tasks, EDF can guarantee schedulability if their total CPU utilization U is:

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1$$

EDF is considered **optimal** *because it can fully utilize the CPU (up to 100%) if the tasks are schedulable.*

Pros & Cons:

- **Pros:** Higher CPU utilization (up to 100% if schedulable), more flexible than RMS, can handle aperiodic tasks well.
- **Cons:** More complex to implement due to dynamic priorities, higher runtime overhead (recalculating priorities), unpredictable behavior if overloaded (all tasks might miss deadlines).

EDF Example: Dynamic Priorities in Action

Tasks: (Execution Time C_i, Period T_i, Deadline D_i $(=T_i)$)

- Task 1 (P1): $C_1 = 1,\ T_1 = 4$
- Task 2 (P2): $C_2 = 2,\ T_2 = 6$

EDF Example: Dynamic Priorities in Action

Tasks: (Execution Time C_i, Period T_i, Deadline D_i $(=T_i)$)

- Task 1 (P1): $C_1 = 1,\ T_1 = 4$
- Task 2 (P2): $C_2 = 2,\ T_2 = 6$

Step 1: Calculate Utilization (U)

$$U = \frac{C_1}{T_1} + \frac{C_2}{T_2} = \frac{1}{4} + \frac{2}{6} = 0.25 + 0.333 = \mathbf{0.583}$$

EDF Example: Dynamic Priorities in Action

Tasks: (Execution Time C_i, Period T_i, Deadline D_i $(=T_i)$)

- Task 1 (P1): $C_1 = 1,\ T_1 = 4$
- Task 2 (P2): $C_2 = 2,\ T_2 = 6$

Step 1: Calculate Utilization (U)

$$U = \frac{C_1}{T_1} + \frac{C_2}{T_2} = \frac{1}{4} + \frac{2}{6} = 0.25 + 0.333 = \mathbf{0.583}$$

Step 2: Apply EDF Schedulability Test $U = 0.583 \leq 1$. The task set is **schedulable by EDF**.

EDF Example: Dynamic Priorities in Action

Tasks: (Execution Time C_i, Period T_i, Deadline D_i $(=T_i)$)

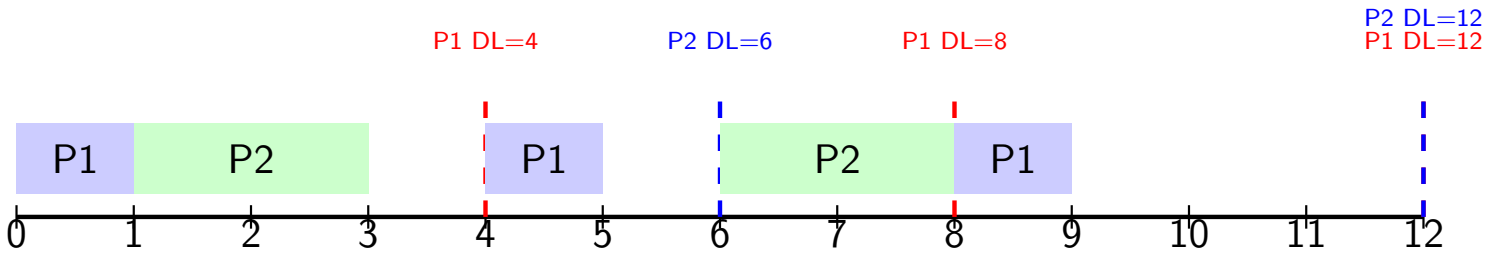
- Task 1 (P1): $C_1 = 1,\ T_1 = 4$
- Task 2 (P2): $C_2 = 2,\ T_2 = 6$

Step 1: Calculate Utilization (U)

$$U = \frac{C_1}{T_1} + \frac{C_2}{T_2} = \frac{1}{4} + \frac{2}{6} = 0.25 + 0.333 = \mathbf{0.583}$$

Step 2: Apply EDF Schedulability Test $U = 0.583 \leq 1$. The task set is **schedulable by EDF**.

Step 3: Corrected Execution (Gantt Chart) *(Tasks arrive at $t = 0$, 4, 6, 8, etc. Deadlines are T_i from arrival.)*



All deadlines are met by dynamically scheduling the task with the earliest deadline.

RMS vs EDF: A Direct Comparison

Feature	Rate Monotonic Scheduling (RMS)	Earliest Deadline First (EDF)
Priority Type	Static (fixed at design time)	Dynamic (changes at runtime)
Priority Basis	Shorter Period $\implies$ Higher Priority	Earliest Deadline $\implies$ Highest Priority
Schedulability Bound	$\leq n(2^{1/n} - 1)$ (approx. 69% for large n)	$\leq 100\%$
Implementation	Simpler (less runtime overhead)	More Complex (dynamic recalculations)
Runtime Overhead	Lower context switching frequency (for fixed tasks)	Higher, due to frequent priority re-evaluations
Overload Behavior	Predictable; highest priority tasks meet deadlines, lower ones may fail.	Unpredictable; all tasks might miss deadlines.
Flexibility	Less flexible; difficult with aperiodic tasks or varying loads.	More flexible; handles aperiodic tasks better.
Common Usage	Hard Real-Time OS (e.g., VxWorks, QNX), industrial control.	Less common in bare-metal RTOS, sometimes in RT Linux, flexible embedded systems.

Linux CFS (Completely Fair Scheduler): General Purpose Fairness I

Concept: Introduced in Linux kernel 2.6.23, CFS is the default scheduler for normal (non-real-time) processes. Its primary goal is **fairness** – ensuring that all processes receive a "fair" share of CPU time relative to their assigned weight.
Key Principles:

- **Proportional Share:** Instead of fixed time slices (like Round Robin), CFS aims for a proportional distribution of CPU time based on task "weight".
- **Virtual Runtime (vruntime):** Each runnable task has a ‘vruntime‘ value, which tracks the amount of time it would have run on an "ideal" perfectly fair CPU.
- **Red-Black Tree:** The runnable tasks are stored in a red-black tree, sorted by their ‘vruntime‘. The task with the smallest ‘vruntime‘ (meaning it's "fallen behind" the most in terms of CPU allocation) is always picked next.
- **Minimizing Latency:** While fairness is primary, CFS also tries to keep interactive task latency low.

Linux CFS (Completely Fair Scheduler): General Purpose Fairness II

Pros:

- **Excellent Fairness:** The scheduler tracks the virtual runtime of each process, ensuring every process gets a "fair" share of CPU time over time. No single process can dominate the CPU.
- **Highly Scalable:** It uses efficient data structures (like a red-black tree) to manage processes, allowing it to scale effectively with a large number of cores and thousands of processes without performance bottlenecks.
- **Good Responsiveness for Interactive Tasks:** Processes that spend more time waiting for user input get a higher priority when they are ready to run again. This makes the system feel quick and responsive.
- **No Starvation:** Every process is guaranteed to eventually run. As a process waits, its priority effectively increases until it is chosen by the scheduler, eliminating the risk of waiting indefinitely.

Linux CFS (Completely Fair Scheduler): General Purpose Fairness III

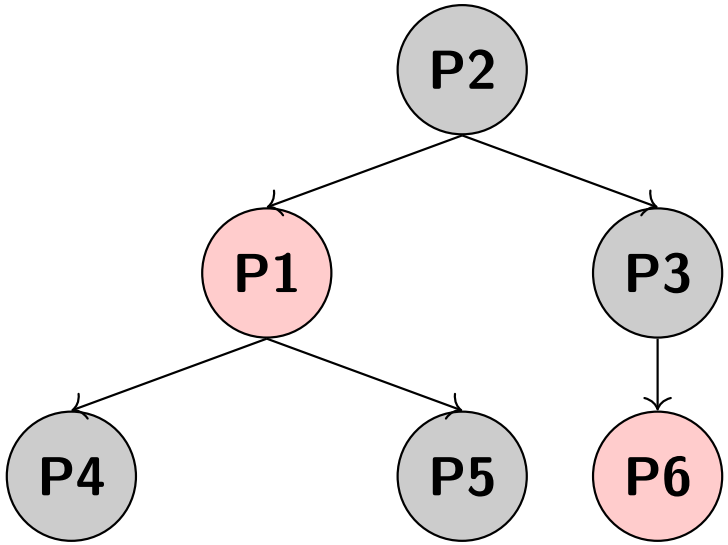
Cons:

- **Not Suitable for Hard Real-Time Tasks:** These schedulers prioritize fairness over strict deadlines. They cannot guarantee that a critical task will be executed by a specific time, making them unsuitable for mission-critical real-time systems.
- **Higher Overhead than Simpler Schedulers:** To achieve their goals, these schedulers use more complex logic and data structures. This results in a slightly higher overhead compared to very simple schedulers like First-Come, First-Served (FCFS) or Round-Robin (RR).

CFS: The Run Queue as a Red-Black Tree

Concept: The red-black tree in CFS acts as the "run queue". It's a self-balancing binary search tree that keeps tasks sorted by their 'vruntime'.

Illustration:



How it works:

- The process with the smallest 'vruntime' is always the leftmost node (e.g., P1). It gets chosen to run.
- When a process's time slice expires, its 'vruntime' is updated.
- The process is then **re-inserted** into a new position in the tree, based on its new 'vruntime' value.

Linux CFS: The vruntime Calculation I

Goal: Ensure processes get CPU time proportional to their "weight" (derived from 'nice' value). **Simplified vruntime Formula (Conceptual):**

$$\text{vruntime}_{\text{new}} = \text{vruntime}_{\text{old}} + \left(\text{actual\_CPU\_run\_time} \times \frac{\text{SCHED\_NICE\_WEIGHT\_HZ}}{\text{task\_weight}} \right)$$

Key Components:

- **'actual_CPU_run_time':** The real CPU time the task just consumed.
- **'nice_value' (User-set priority):**
  - A user-set value (from -20 to +19, default 0).
  - Lower 'nice' value (e.g., -20) implies higher user preference.

Linux CFS: The vruntime Calculation II

- **‘task_weight‘ (Derived from ‘nice_value‘):**
  - A lookup table maps ‘nice_value‘ to ‘task_weight‘.
  - Higher ‘task_weight‘ means the task deserves more CPU time (and its ‘vruntime‘ will increase slower for the same actual runtime).
  - Example: ‘nice 0‘ has a weight of 1024. ‘nice 1‘ has weight 820. ‘nice -5‘ has weight 2379.
- **‘SCHED_NICE_WEIGHT_HZ‘:** A constant for normalization (1024 in many kernels, corresponding to nice 0’s weight).

Essentially, ‘vruntime‘ aims to track normalized CPU usage. A task with a lower ‘nice‘ value (higher weight) will see its ‘vruntime‘ increase slower, meaning it stays closer to the leftmost part of the red-black tree and gets scheduled more often.

Windows Scheduling Mechanism: Priority-Based Responsiveness I

Overview: The Windows scheduler (Dispatcher) is a priority-based, preemptive, and multi-level feedback queue scheduler that operates on threads. **Key**

Features:

- **32 Priority Levels:**
  - **Real-Time (16-31):** Fixed priorities, for system-critical tasks. No dynamic changes.
  - **Dynamic (1-15):** Priorities can change based on system activity (aging, boosting).
  - **Zero (0):** Special system idle thread.

Windows Scheduling Mechanism: Priority-Based Responsiveness II

- **Dynamic Priority Adjustments:**
  - **Boosting:** Threads receive temporary priority boosts for various reasons (e.g., completing I/O, becoming foreground application, user input). This enhances responsiveness.
  - **Decaying (Aging):** After a quantum expires or a boost expires, a thread's dynamic priority is typically lowered (decayed) over time, unless it's a critical thread.
- **Quantum-Based Preemption:** Each thread runs for a fixed time quantum. When the quantum expires, the thread is preempted.
- **Dispatcher Object:** The central component that selects the highest-priority runnable thread and performs context switching.

Windows Scheduling Mechanism: Priority-Based Responsiveness III

Emphasis: Windows scheduling strongly emphasizes **responsiveness for interactive applications** (especially foreground UI threads) through its boosting and priority management mechanisms. **Pros & Cons:**

- **Pros:** Excellent responsiveness for interactive tasks, good balance between throughput and latency, robust priority management.
- **Cons:** Can be complex to predict exact behavior, potential for priority inversion (though mitigations exist), not designed for hard real-time guarantees.

Comparing Linux CFS and Windows Scheduler

Feature	Linux CFS	Windows Scheduler
Core Philosophy	Proportional Share / Fairness	Priority-based / Responsiveness
Preemption	Yes (based on ‘vruntime‘ / quantum)	Yes (based on priority / quantum)
Priority Management	Dynamic ‘vruntime‘ (from ‘nice‘ value)	32 fixed + dynamic levels (boosting/decaying)
Run Queue Structure	Red-Black Tree	Ready queues (one per priority level)
Starvation Prevention	Inherent in ‘vruntime‘ fairness model	Aging and Priority Boosting
Real-Time Support	Separate ‘SCHED_FIFO‘/‘SCHED_RR‘ policies	Dedicated Real-Time priority levels (16-31)
Typical Workload	Servers, Desktops, Embedded (fair for all types)	Desktops, Workstations (optimized for interactivity)

Hands-on Linux: Observe Scheduler Behavior I

Explore Linux Process and Scheduler States:

- Open a terminal and use these commands to observe processes and their scheduling attributes.
- Pay attention to ‘PR‘ (priority), ‘NI‘ (nice value), ‘STAT‘ (process state), ‘

```
# Show processes in a tree structure, including NI
    (nice) values
htop --tree # or ‘top‘ and look at NI column

# List processes with specific scheduling
   information
# PID: process ID, COMM: command, NI: nice value,
   PRI: kernel priority
# PCPU: CPU usage (percentage), STAT: process
   status
ps -eo pid,comm,ni,pri,pcpu,stat --sort=-pcpu
```

Hands-on Linux: Observe Scheduler Behavior II

```
# Check real-time scheduling policy for a specific
    process (replace <pid>)
# (e.g., `chrt -p <pid of your shell>`)
chrt -p <pid>

# Run a process with a custom nice value (e.g.,
   lower priority by setting nice 10)
# This process will receive less CPU time from CFS
   .
nice -n 10 dd if=/dev/zero of=/dev/null &

# Run a process with a higher priority (negative
   nice value requires sudo)
# (e.g., `sudo nice -n -10 dd if=/dev/zero of=/dev
   /null &`)
```

Hands-on Linux: Observe Scheduler Behavior III

Advanced: Inspecting Scheduler Debug Information (Requires root):

- This file provides detailed, low-level insights into the CFS scheduler's current state, including ‘vruntime‘ values.

```
sudo cat /proc/sched_debug | less
```

(Look for sections like 'runnable tasks' and their 'vruntime' values.)

Key Takeaways

- **Real-Time Scheduling** guarantees deadlines, with **Hard RT** demanding absolute predictability (e.g., RMS, EDF).
- **RMS** uses static priorities (shorter period $\implies$ higher priority) and has a clear utilization bound.
- **EDF** uses dynamic priorities (earliest deadline $\implies$ highest priority), can achieve 100% utilization, but is more complex.
- **Linux CFS** is a general-purpose, fairness-based scheduler using **virtual runtime** and a **red-black tree** for proportional CPU distribution.
- **Windows Scheduler** is a priority-based, preemptive system that prioritizes **responsiveness** through dynamic **priority boosting** and decaying.
- **The choice of scheduler depends heavily on the system's goals**: strict deadlines vs. general-purpose fairness and responsiveness.

Key Takeaways

- **Real-Time Scheduling** guarantees deadlines, with **Hard RT** demanding absolute predictability (e.g., RMS, EDF).
- **RMS** uses static priorities (shorter period $\implies$ higher priority) and has a clear utilization bound.
- **EDF** uses dynamic priorities (earliest deadline $\implies$ highest priority), can achieve 100% utilization, but is more complex.
- **Linux CFS** is a general-purpose, fairness-based scheduler using **virtual runtime** and a **red-black tree** for proportional CPU distribution.
- **Windows Scheduler** is a priority-based, preemptive system that prioritizes **responsiveness** through dynamic **priority boosting** and decaying.
- **The choice of scheduler depends heavily on the system's goals**: strict deadlines vs. general-purpose fairness and responsiveness.

Reflection Prompt: The Scheduler's Dilemma

Imagine a smartphone running both a real-time voice assistant (requires low latency) and a background photo upload (CPU-intensive). How might the concepts of RMS/EDF and CFS/Windows-like boosting be combined or modified to handle such a mixed workload effectively, without one starving the other?

Next Week Preview: Threads and Multithreading Models

From Processes to Threads: Finer-Grained Concurrency

- **Introduction to Threads:** Why threads are needed, process vs. thread.
- **User-Level vs. Kernel-Level Threads:** Understanding their differences and implementation.
- **Multithreading Models:** One-to-One, Many-to-One, Many-to-Many models.
- **Thread Libraries (‘pthreads‘):** How developers create and manage threads.
- **Thread Context Switching:** How it differs from process context switching.

Next Week Preview: Threads and Multithreading Models

From Processes to Threads: Finer-Grained Concurrency

- **Introduction to Threads:** Why threads are needed, process vs. thread.
- **User-Level vs. Kernel-Level Threads:** Understanding their differences and implementation.
- **Multithreading Models:** One-to-One, Many-to-One, Many-to-Many models.
- **Thread Libraries (‘pthreads‘):** How developers create and manage threads.
- **Thread Context Switching:** How it differs from process context switching.

Prep Tip for Next Session

Review the concept of a process Control Block (PCB) and process context. Think about what information would still be unique to a thread and what could be shared among threads within the same process.

Outline

Quick Quiz: Real-Time & Modern Scheduling I

Test Your Conceptual Understanding:

1. **Scenario:** You have two periodic tasks. Task A: (C=2, T=5), Task B: (C=1, T=3).
   - What priority would RMS assign to Task A relative to Task B?
   - Could EDF schedule these tasks if RMS cannot?

2. **True/False:** In Linux CFS, a task with a 'nice' value of +10 will typically receive more CPU time than a task with a 'nice' value of 0, assuming they are otherwise identical. Justify your answer.

3. **Define:** Explain the primary purpose of "priority boosting" in the Windows Scheduler. How does it benefit interactive user experience?

4. **Application:** For which type of system (e.g., desktop PC, missile guidance system, cloud server) would a scheduler like EDF be a better fit than CFS, and why?

Quick Quiz: Real-Time & Modern Scheduling II

Think & Discuss

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

Exercise: Real-Time Schedulability Analysis I

Part 1: RMS Schedulability Check
Tasks: (Execution Time C_i, Period T_i)

- Task A: $C_A = 2,\ T_A = 5$
- Task B: $C_B = 1,\ T_B = 4$
- Task C: $C_C = 1,\ T_C = 10$

Task:

1. Calculate the total CPU utilization (U) for this set of tasks.
2. Calculate the RMS schedulability bound for $n = 3$ tasks.
3. Based on the RMS schedulability test, can these tasks be guaranteed to be scheduled by RMS? Show your calculations.
4. What are the RMS priorities for Task A, B, and C?

Exercise: Real-Time Schedulability Analysis II

Part 2: Conceptual Comparison
Scenario: A new game console OS needs to be designed. It handles high-priority game rendering processes, background downloads, and occasional real-time voice chat.
Task:

1. Would a pure FCFS scheduler be suitable? Why or why not?
2. Would a pure EDF scheduler be a good choice for *all* tasks? Discuss potential benefits and drawbacks in this mixed environment.
3. Propose a hybrid scheduling approach (e.g., using concepts from MLQ, Priority, and fairness) that could effectively manage these diverse workloads. Justify your proposal.

Exercise: Real-Time Schedulability Analysis III

Reminder

Remember to clearly state your assumptions and show all steps for calculations.

Advanced Topics to Explore I

Delving Deeper into Real-Time and Advanced Schedulers

I. Real-Time Scheduling Extensions

- **Resource Sharing in RTOS:** Understanding solutions like Priority Inheritance Protocol (PIP) and Priority Ceiling Protocol (PCP) to prevent priority inversion.
- **Schedulability Analysis for Aperiodic Tasks:** Techniques for integrating sporadic or aperiodic tasks into periodic real-time schedules (e.g., deferrable server, sporadic server).
- **Multi-Core Real-Time Scheduling:** Challenges and approaches for guaranteeing deadlines across multiple CPU cores.

Advanced Topics to Explore II

II. Advanced Aspects of General-Purpose Schedulers

- **Linux CFS Group Scheduling (‘cgroups‘):** How CFS can apply fairness policies not just to individual tasks but to groups of tasks (e.g., ensuring a user or VM gets a certain CPU share).
- **Windows Thread Affinity & Processor Groups:** How threads can be bound to specific CPUs and how Windows handles systems with more than 64 logical processors.
- **Scheduler Interfacing (Syscalls):** Deeper look at system calls like ‘sched_setscheduler‘, ‘setpriority‘ (Linux) and ‘SetThreadPriority‘ (Windows) for manipulating scheduling parameters from user space.

Advanced Topics to Explore III

III. Emerging & Specialized Scheduling Areas

- **Energy-Aware / Power-Aware Scheduling:** Algorithms that consider CPU frequency scaling (DVFS) and core parking to optimize power consumption, crucial for mobile and data centers.
- **Virtual Machine Schedulers:** How hypervisors (like VMware ESXi, KVM) schedule virtual CPUs onto physical CPUs.
- **Quantum-Leap Schedulers (e.g., Lottery Scheduling, Stride Scheduling):** Alternative fairness-based schedulers for specific contexts.