Operating Systems Project: Topic 1

Modify Linux Kernel Scheduler

Liangsen Wang

224040364@link.cuhk.edu.cn

2026.1.12

Outline

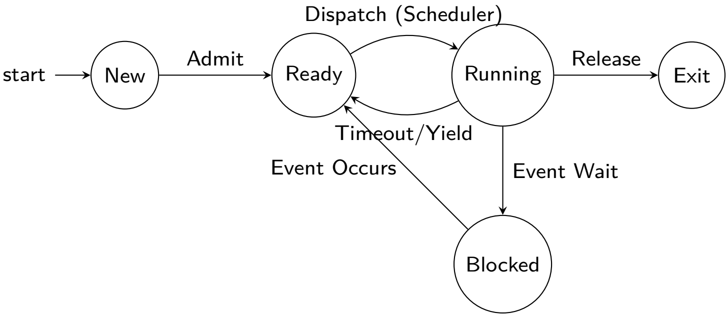
Outline

Project Overview & Motivation

- **Objective**: Understand and improve the Linux process scheduler.
- **Why this topic?**
  - The scheduler is the heart of the OS, determining responsiveness and throughput.
  - Mastering this allows you to optimize systems for specific workloads (e.g., Real-time, HPC).
- **Core Challenge**: Working with kernel/sched/ code—one of the most complex parts of the kernel.
- You can obtain the Linux source code from https://www.kernel.org/
- Linux source code (v6.18.4) - Bootlin Elixir Cross Referencer can help you to search symbols in the kernel

Outline

OS Refresher: The 5-State Process Model



- **Ready:** Processes waiting in the Runqueue.
- **Running:** Process currently executing on CPU.
- **Scheduler's Role:** Decides which process transitions from *Ready* $\rightarrow$ *Running*.

OS Refresher: Context Switch

Definition

The process of storing the state of a process so that it can be restored and resume execution later.

What is saved?

- Program Counter (PC)
- Stack Pointer (SP)
- General Purpose Registers
- Process Control Block (PCB)

Implications

- **Overhead:** CPU does no useful work during switching.
- **Frequency:** High frequency = Responsiveness but low throughput.

Linux Reality: task_struct

Defined in include/linux/sched.h, the task_struct is the PCB in Linux.

1. Process Identity & State

These fields are essential for debugging your scheduler (printk/logging).

- pid_t pid;: The Process ID.
- char comm[TASK_COMM_LEN];: The name of the program (e.g., "bash", "python").
- volatile long state; (or __state in newer kernels):
  - -1: Unrunnable
  - 0 (TASK_RUNNING): **Ready** or **Executing**.

```
struct task_struct {
    volatile long state;    // -1 unrunnable, 0 runnable, >0 stopped
    void *stack;            // Kernel stack
    pid_t pid;
    char comm[16];          // Executable name
    ...
};
```

Linux Reality: task_struct

2. The Scheduling "Hooks" (Critical for Topic 1)

Linux supports multiple scheduling policies simultaneously using **Scheduling Classes**.

- `struct sched_class *sched_class`:
  - Pointer to the function table (polymorphism).
  - E.g., `fair_sched_class`, `rt_sched_class`.
  - **Project Goal:** You might create `my_sched_class`.
- `struct sched_entity se`:
  - Used by CFS (Completely Fair Scheduler).
  - Contains `vruntime` (Virtual Runtime).

```c
struct task_struct {
    ...
    int prio;
    int static_prio;
    int normal_prio;
    unsigned int rt_priority;

    const struct sched_class *sched_class;
    struct sched_entity se;       // For CFS
    struct sched_rt_entity rt;    // For Real-time
    ...
};
```

Modular Scheduling Classes

Does Linux use one huge algorithm for all tasks? No!

Linux uses a **Modular Architecture**. Each "Scheduling Class" encapsulates a specific policy. The kernel iterates through them in a fixed priority order:

1. Stop Class (Migration/Shutdown) — Highest Priority

2. Deadline Class (SCHED_DEADLINE)

3. RT Class (SCHED_FIFO, SCHED_RR)

4. Fair Class (SCHED_NORMAL) ← *Most processes!*

5. Idle Class (swapper) — Lowest Priority

Polymorphism in C:

```c
struct sched_class {
    const struct sched_class *next;

    // Interface functions
    void (*enqueue_task) (...);
    void (*dequeue_task) (...);
    struct task_struct *(*pick_next_task) (...);
};
```

When `pick_next_task()` is called, the kernel asks the **Stop Class** first. If it returns NULL, it asks **Deadline**, and so on.

Understanding the Hierarchy: The "VIP" Line

Linux uses a modular hierarchy. When `pick_next_task` is called, it checks classes in strictly descending order:

1. **`stop` (Highest):** Kernel emergency tasks (migration, shutdown).
2. **`dl` (Deadline):** Hard real-time guarantees (`SCHED_DEADLINE`).
3. **`rt` (Real-Time):** POSIX real-time (`SCHED_FIFO`, `SCHED_RR`).
4. **`fair` (CFS):** **Normal user processes (Bash, Chrome, Python)**.
5. **`idle` (Lowest):** Runs only when CPU has nothing else to do.

Critical Implication

If a task in the `rt` class enters an infinite loop, tasks in the `fair` class (your shell, GUI) will **never** execute. The system will appear to hang.

Outline

Overview of Linux Scheduling Policies

Linux supports multiple policies defined by POSIX standards.

Real-Time (Static Priority 0-99)

- `SCHED_FIFO`: First-In, First-Out.
- `SCHED_RR`: Round Robin (FIFO + Time Slice).

Deadline (Highest Priority)

- `SCHED_DEADLINE`: Earliest Deadline First (EDF).

Normal (Dynamic Priority 100-139)

- `SCHED_NORMAL`: Standard CFS (Completely Fair Scheduler).
- `SCHED_IDLE`: For background jobs.

Real-Time: FIFO vs. Round Robin

These policies are managed by `rt_sched_class`.

SCHED_FIFO (First-In, First-Out)

- **Logic:** Run the highest priority task until it:
  1. Blocks (waits for I/O).
  2. Yields voluntarily (`sched_yield()`).
  3. Is preempted by a *higher* priority task.
- **No Time Slice:** A purely CPU-bound FIFO task can starve the entire system (infinite loop risk).

SCHED_RR (Round Robin)

- **Logic:** Same as FIFO, but with a **Time Quantum** (slice).
- If the slice expires, the task is moved to the end of the queue for its priority level.
- Ensures fairness among real-time tasks of the *same* priority.

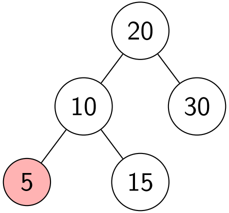
Normal: CFS (Completely Fair Scheduler)

Managed by `fair_sched_class`. Used for 99% of user tasks.

- **Goal:** Model an "Ideal Multi-Tasking CPU" on real hardware.
- **Mechanism:**
  - **Virtual Runtime (`vruntime`):** A counter that increases as the process runs.
  - **Weight:** High priority (low nice value) tasks increase `vruntime` slower (get more CPU).

Data Structure: Red-Black Tree

- Replaces the Runqueue array.
- **Key:** `vruntime`.
- **Selection:** Always pick the **leftmost** node (smallest `vruntime`).
- **O(log N)** efficiency.



(Leftmost = Next)

Outline

Topic 1: Basic Requirements (Overview)

Objective: Understand the generic Linux scheduling framework and implement a custom scheduling policy.

Mandatory Requirements (Pass/Fail Criteria):

1. **Locate & Modify:**
   - Modify `kernel/sched/fair.c` or add a new scheduling class.
   - Add printk/logging to prove your code is running.
2. **Implement a Concrete Algorithm:**
   - You must implement **AT LEAST TWO** specific algorithm (Details on next slide).
   - The scheduler must handle `enqueue`, `dequeue`, and `pick_next_task`.
3. **Performance Comparison:**
   - Run `sysbench --test=cpu` or you can choose a specific workload and implement the corresponding targeted scheduling strategy. .
   - Compare Context Switches (csw) & Latency vs. Standard CFS.
4. **GUI Visualization:** Display process status (PID, State, Priority).

Topic 1: Select Your Target Strategy

Choose AT LEAST TWO of the following strategies to implement:

Option A: Lottery Scheduler (Random)

"The more tickets you have, the higher chance to run."

- **Mechanism:** Assign "tickets" to tasks based on their Nice value (Static Priority).
- **Logic:** Generate a random number [0, Total Tickets]. Traverse the queue and pick the winner.
- **Pros:** Easy to implement; statistically fair over time.

Option B: Weighted Round Robin (WRR)

"Higher priority gets a larger time slice."

- **Mechanism:** Replace the CFS Red-Black Tree with a simple **Linked List**.
- **Logic:** Iterate through the list. Assign time slice = Base Slice $\times$ Weight.
- **Pros:** Deterministic; simpler than CFS.

Topic 1: Select Your Target Strategy

Choose AT LEAST TWO of the following strategies to implement:

Option C: Shortest Remaining Time First (SRTF)

"Finish small jobs fast."

- **Mechanism:** Add a `burst_time` field to `task_struct` (simulate via syscall).
- **Logic:** Always pick the task with the smallest remaining burst time.
- **Pros:** Minimizes average waiting time (Theoretical Optimum).

Implementation: Where to start?

Key Files in Kernel Source:

- `kernel/sched/core.c`: Main scheduler entry point (`__schedule`).
- `kernel/sched/fair.c`: Example implementation (CFS).

Critical Function Hook (Example for Lottery):

```
// In kernel/sched/fair.c
struct task_struct *
pick_next_task_fair(struct rq *rq, struct task_struct *prev, struct rq_flags *rf)
{
    // 1. Calculate total tickets (for Lottery)
    // 2. Generate random number
    // 3. Iterate 'cfs_rq->tasks' list to find the winner
    // 4. Return the 'task_struct' of the winner

    // NOTE: You might need to disable the RB-Tree logic
    // and rely on the list_head for simple algorithms.
}

```

Requirement: GUI Visualization

You need to show what is happening inside the Kernel.

Recommended Approach:

1. **Kernel Side:** Expose data via `/proc/mysched`.
    - Iterate through 'task_struct' list.
    - Print PID, Name, State, **Tickets/Weight**.
2. **User Side:** Python/C++ GUI.
    - Read `/proc/mysched` every 100ms.
    - **Visualization:** Draw a Pie Chart (Ticket distribution) or Gantt Chart.

Tip

Visualizing the "Tickets" or "Time Slices" specifically helps verify your algorithm works!

Outline

Grading Criteria (Topic 1 Specific)

Component	Focus
Basic Implementation (25%)	Logic correctness, Code structure (The "Must-Haves").
Advanced Options (10%)	**Innovation & Complexity** (See next slides).
Stability (5%)	**NO KERNEL PANICS**.
Presentation (20%)	Live Demo + Q&A.
Report (20%)	Analysis of results (Why did performance change?).

Note: Advanced Options are "Open-Ended".

Advanced Options: Freedom to Explore

> *"Advanced Options are just some suggestions; any topic-related implementation you wish to carry out is welcome."*

You are encouraged to go beyond the list. Choose a path that interests you:

Path A: System Features

Add practical features to your scheduler:

- **Dynamic Switching:** Switch policies at runtime via `sysctl` without rebooting.
- **NUMA-Awareness(Multi core):** Prefer CPUs on the same memory node to reduce latency.

Path B: Research & Optimization

- Reproduce a paper.
- Optimize for a specific workload.

Research-Oriented Examples (For Path B)

Example 1: Paper Reproduction

- Read a classic or recent paper from **SOSP, OSDI, EuroSys or other system conference**.
- Implement a simplified version of their algorithm.

Example 2: Workload-Aware Scheduling

- Analyze a specific application you care about (e.g., Redis, Video Encoding, Gaming).
- Design a policy tailored for it.
- *e.g., "A Scheduler that prioritizes tasks holding a mutex lock to reduce lock contention duration."*

Next Steps

1. Download Linux Kernel Source (Recommend v5.x or v6.x).
2. Set up a Virtual Machine or simulator (QEMU/KVM/VBOX) for development.
   - **Warning:** Do not develop on your host OS directly.
3. Read `kernel/sched/sched.h` and other related code to understand `struct sched_class` and the implementation methods of Linux scheduling.
4. Realize your ideas.