CPU Scheduling

Bocheng Jiang, Yiheng Zhang, Zhe Xue

October 19, 2025

Contents

Turnaround Time: Definition

Turnaround time measures how long a job spends in the system from arrival to completion.

$$T_{\text{turnaround}} = T_{\text{completion}} - T_{\text{arrival}}.$$

When all jobs arrive at time 0, $T_{\text{turnaround}} = T_{\text{completion}}$.

FIFO (First In, First Out)

One-sentence summary: Run jobs in the order of arrival; each job runs to completion before the next begins.

Problem — Convoy Effect: Short jobs are blocked behind a long job, leading to very high waiting times and poor average turnaround time.

SJF (Shortest Job First)

One-sentence summary: Always run the job with the shortest total runtime first.

Characteristics:

- **Non-preemptive.**
- If short jobs arrive while a long job is running, they are still blocked until the long job completes.

STCF (Shortest Time to Completion First)

One-sentence summary: Always run the job with the least remaining time; preempt currently running jobs if needed.

Characteristics:

- **Preemptive.**
- When job lengths are known, STCF is **optimal** for average turnaround time (ignoring context-switch overhead).

Response Time: Definition

Response time measures how long it takes for a job to get the CPU for the first time.

$$T_{\text{response}} = T_{\text{first run}} - T_{\text{arrival}}.$$

Lower response time improves interactivity in time-sharing systems.

Round Robin (RR)

One-sentence summary: Alternate among runnable jobs in fixed-length **time slices** (quanta).

Trade-off:

- If the time slice is too short, frequent context switches increase overhead.
- If the time slice is too long, interactivity suffers.

Characteristics:

- Excellent response time, but **poor average turnaround time** because each job is **stretched out**, delaying overall completion.

Summary & Reflections

Incorporating I/O: Overlap CPU bursts of one job with I/O of another to increase utilization.
Key broken assumption: The OS cannot **predict the future** (job lengths are unknown).
Open question: How can we achieve SJF/STCF-like performance **without** knowing job lengths, while also maintaining good response time?
Analogy: Learn from history to predict the future (e.g., branch prediction, LRU caching).

Contents

MLFQ: Motivation & Core Question

Core question: How can we schedule effectively **without knowing** job runtimes?

Dual objectives:

1. **Minimize average turnaround time** (SJF-like: prefer short jobs).
2. **Provide good interactive response** (quickly serve interactive tasks).

Challenge: The OS does not know job runtimes in advance.

Idea: Adapt priority using observed history of job behavior.

MLFQ: High-Level Approach

High-level approach:

- Maintain **multiple queues** with different priorities.
- **Learn** from a job's recent CPU vs. I/O behavior.
- **Promote** interactive/short phases; **demote** long CPU-bound phases.

Goal: Approximate SJF/STCF **and** preserve responsiveness **without** a priori knowledge.

Basic Rules

Multiple priority queues:

- **Rule 1:** If Priority(A) $>$ Priority(B), run A.
- **Rule 2:** If Priority(A) $=$ Priority(B), use **Round Robin** within that queue.

Dynamic priorities: Unlike fixed-priority schemes, MLFQ **adjusts priority** based on observed behavior.

First Attempt: Changing Priorities

Allotment: Total CPU time a job may use at a given priority level before demotion.

Rules:

- **Rule 3:** New jobs start in the **highest-priority** queue.
- **Rule 4a:** If a job **uses up** its allotment, **demote** it.
- **Rule 4b:** If a job **yields early** (e.g., for I/O), **keep** it at the same priority.

Problems with the First Attempt

Issues:

1. **Starvation:** Many interactive jobs can prevent long CPU-bound jobs from running.
2. **Phase changes:** CPU-bound jobs may later become interactive but remain stuck at low priority.
3. **Gaming:** Yield just before allotment ends to avoid demotion and hog high priority.

Second Attempt: Priority Boost

Rule 5: Every period S, **boost all jobs** to the topmost queue.
Benefits:

- Prevents starvation; all jobs **eventually** get CPU time.
- If a job becomes interactive, it will **recover** a high priority.

Choosing S: Hard to set. Too large $\Rightarrow$ starvation; too small $\Rightarrow$ hurts interactivity.
(Ousterhout: *"voodoo constants"*.)

Third Attempt: Better Accounting

Anti-gaming fix: Track **cumulative CPU time** used at the current level.
Revised Rule 4: Once a job has **used up its allotment** at a given level—**regardless of how many yields—demote** it.
Effect: A job cannot remain at high priority indefinitely by yielding frequently.

Tuning Parameters & Variants

Parameters to tune:

- Number of queues.
- Time-slice length per queue.
- Boost period S.
- Allotment per level.

Common design:

- High-priority queues: **short** quanta ($\sim$10 ms) for interactive tasks.
- Low-priority queues: **long** quanta (hundreds of ms) for CPU-bound tasks.

Real-World Examples

Solaris TS (Time-Sharing):

- ∼60 queues; time slices from ∼20 ms to a few hundred ms.
- Priority boost roughly every ∼1 s.

FreeBSD 4.3:

- Priority computed by a formula based on recent CPU usage with decay.

Adoption: Windows NT, BSD UNIX, Solaris all use MLFQ variants.

Final Rule Set (Summary)

1. Higher priority runs first.
2. Same-priority jobs use Round Robin.
3. New jobs start at the highest priority.
4. Use up allotment at a level $\Rightarrow$ demote.
5. Periodically boost all jobs to the highest priority.

Outcome: Learn from history to **approximate** SJF/STCF for short/interactive jobs while ensuring **progress** for long CPU-bound workloads.

Contents

Section Overview

- Re-recognizing Traditional Scheduling Discipline
- Scheduling Challenges: The Overhead Problem
- Dynamic or Time-Dependent Priorities

Ref: Coffman, E. G., & Kleinrock, L. (1968). Computer scheduling methods and their countermeasures. *Proceedings of the AFIPS '68 Spring Joint Computer Conference*, 11–21. DOI: 10.1145/1468075.1468078

Re-recognizing Traditional Scheduling Discipline

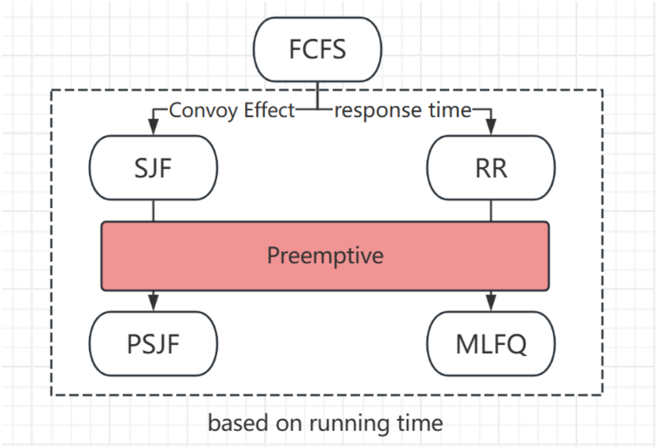
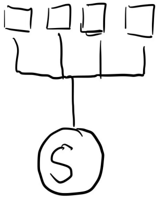
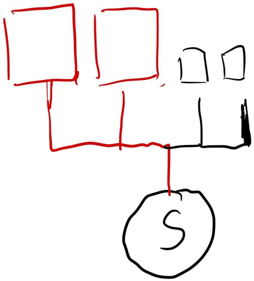


Figure: Priorities based only on running times

Re-recognizing Traditional Scheduling Discipline





RR

MLFQ

Re-recognizing Traditional Scheduling Discipline

Key Idea

It can be seen that:

- The Round Robin (RR) and MLFQ processor-sharing disciplines differ in structure in **precisely the same way** as the Shortest Job First (SJF) and Preemptive Shortest Job First (PSJF) disciplines.
- the use of running time as a means of assigning priorities is **implicit** in the RR discipline.

Scheduling Challenges: The Overhead Problem

Problem & Disciplines

The Challenge

Goal: Reduce **overhead** (context switching) and **swapping costs** in quantum-controlled service.

Solution Categories

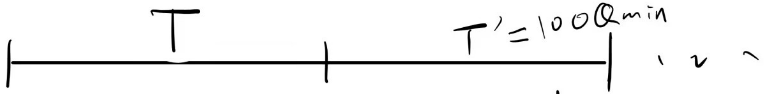
Shifting from fixed to dynamic/periodic control:

1. **Cycle-Oriented Disciplines**
2. **Input-Dependent Disciplines**

Cycle-Oriented Disciplines

- **Strategy 1: Dynamic Quantum**
  - **Rule:** Distribute Fixed Cycle Time (T) among n jobs.
  - **Formula:** $Q = \max\left(\frac{T}{n}, Q_{\min}\right)$
  - **Key:** $Q_{\min}$ prevents small quanta $\rightarrow$ **Reduces Context Switch Overhead.**
- **Strategy 2: Two-Level MLFQ or Limited RR**
  - **Queues:** Foreground (Interactive) $>$ Background (Production)
  - **Rules (Cycle-based):**
    1. Foreground gets fixed Q.
    2. Background uses **only remaining cycle time**.
    3. **Graceful Degradation:** Cycle $\rightarrow$ **Extended** if too many foreground jobs increase swapping risk.

Dynamic Quantum



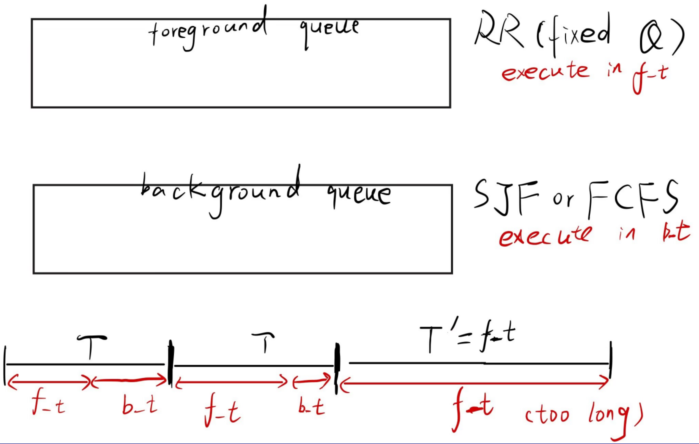
2 jobs

$\Rightarrow Q = \frac{T}{2}$

100 jobs

$\Rightarrow Q = Q_{min}$

Two-Level MLFQ or Limited RR



Input-Dependent Disciplines

Core Idea

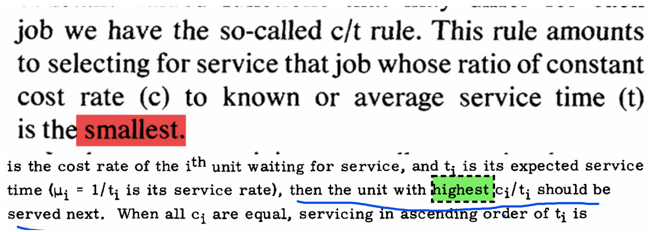
Service is adjusted based on **External Events** or **User Activity**.

- **Strategy 1: Arrival-Triggered Extension**
  - **Mechanism:** New job arrival $\rightarrow$ Job in service gets an **ADDITIONAL Quantum**.
  - **Goal:** Delays preemption; reduces switching overhead.
- **Strategy 2: Rate-Sensitive Response**
  - **Mechanism:** orders the queue of inter active user's jobs by interarrival time.
  - **Result:** those users communicating with the system at the faster rates will receive the shorter response time.

Dynamic or Time-Dependent Priorities

- **Aging Mechanism: Growing Priority through Time**
- **General Cost Accrual Discipline**
  - **Cost Accrual Rate ($C(t)$):** A function of time representing the cost rate incurred by a waiting job.
  - **The C/T Rule**: if c_i is the cost rate of the i th unit waiting for service, and t_i is its expected service time ($\mu_i = \frac{1}{t_i}$ is its service rate), then the unit with highest $\frac{c_i}{t_i}$ should be served next.

Fun Fact


job we have the so-called c/t rule. This rule amounts to selecting for service that job whose ratio of constant cost rate (c) to known or average service time (t) is the smallest.

is the cost rate of the ith unit waiting for service, and t_i is its expected service time ($\mu_i = 1/t_i$ is its service rate), then the unit with highest c_i/t_i should be served next. When all c_i are equal, servicing in ascending order of t_i is

Figure: Upper is the mistake,lower is the origin

The paper said to pick the smallest ratio.
I checked the original source paper it was referencing.
Yep, the author made a mistake!

Contents

Proportional-Share Scheduling

Goal: Share CPU resources fairly among processes based on assigned weights.

Examples:

- **Lottery Scheduling** – probabilistic fairness using random selection.
- **Stride Scheduling** – deterministic fairness using stride and pass values.

Key Idea: Each process gets a proportional share of CPU time according to its assigned value (tickets or stride).

Lottery Scheduling

Concept:

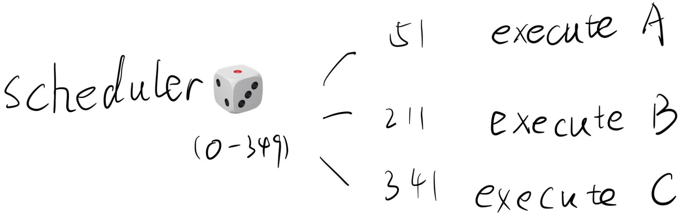
- Each process owns a number of **tickets** representing its CPU share.
- The scheduler draws a random winning ticket each time slice.
- The process holding that ticket gets scheduled.

Features:

- Simple and probabilistic fairness.
- Easy to add or remove processes dynamically.

Lottery Scheduling

	tickets	numbers
A	100	0 — 99
B	200	100 — 299
C	50	300 — 349



Lottery Scheduling

Pseudocode:

```
int counter = 0;
int winner = getrandom(0, totaltickets);
node_t *current = head;
while (current) {
    counter += current->tickets;
    if (counter > winner)
        break; // found the winner
    current = current->next;
}
// 'current' is the winner: schedule it...
```

Stride Scheduling

Concept:

- Deterministic version of lottery scheduling.
- Each process has a **stride** inversely proportional to its tickets:

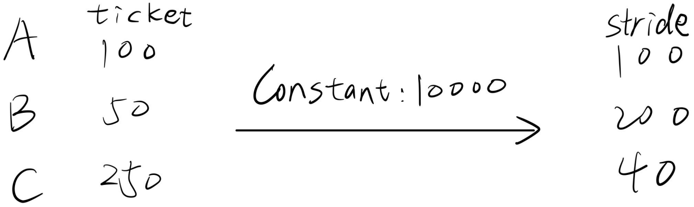
$$stride = \frac{\text{large constant}}{\text{tickets}}$$

- Each process maintains a **pass** value — lower pass = higher priority.

Characteristics:

- Deterministic fairness — exact proportional share.
- Needs global state (pass values), harder to handle new jobs.
- Suitable for predictable environments.

Stride Scheduling



Stride Scheduling

Pass(A) (stride=100)	Pass(B) (stride=200)	Pass(C) (stride=40)	Who Runs?
0	0	0	A
100	0	0	B
100	200	0	C
100	200	40	C
100	200	80	C
100	200	120	A
200	200	120	C
200	200	160	C
200	200	200	...

Stride Scheduling

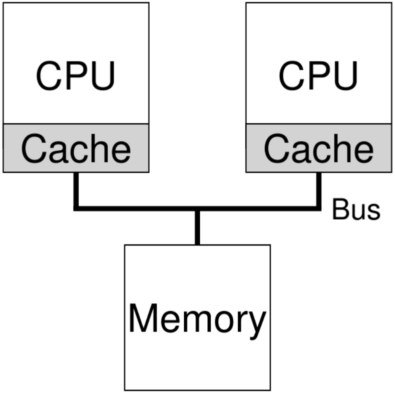
Pseudocode:

```
// pick client with min pass
curr = remove_min(queue);
schedule(curr);                    // run for quantum
curr->pass += curr->stride;        // update pass
insert(queue, curr);               // reinsert into queue
```

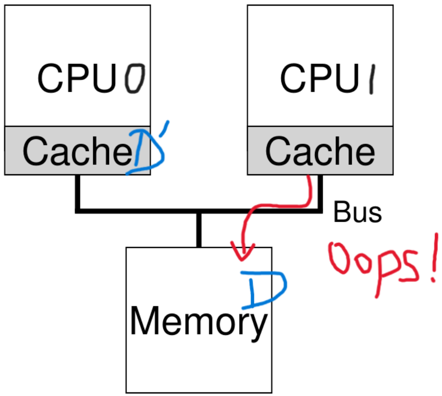
Contents

Multi-processor Scheduling

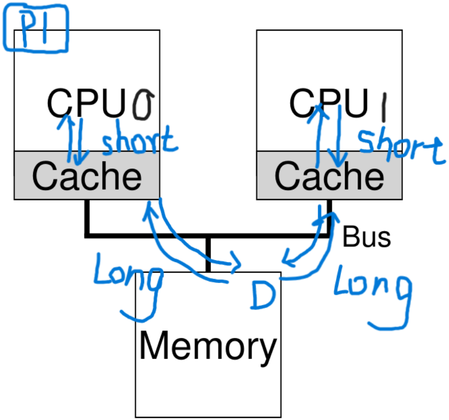
Background: Structure of Multi-processor



Cache Coherence



Cache Miss



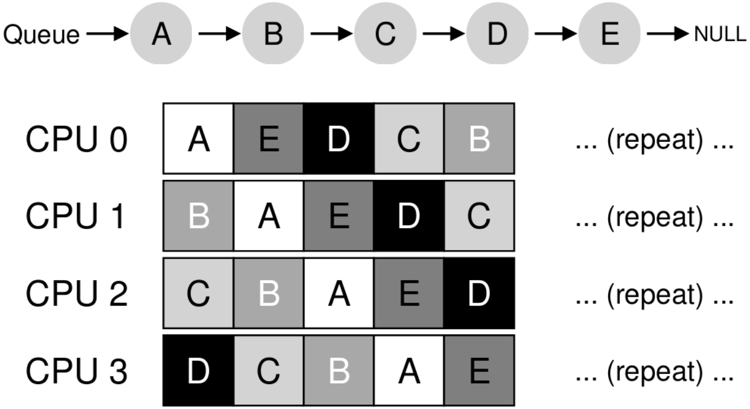
Cache Affinity

Definition: Cache affinity refers to the tendency of a process or thread to continue running on the same CPU where it previously executed.

Why it matters:

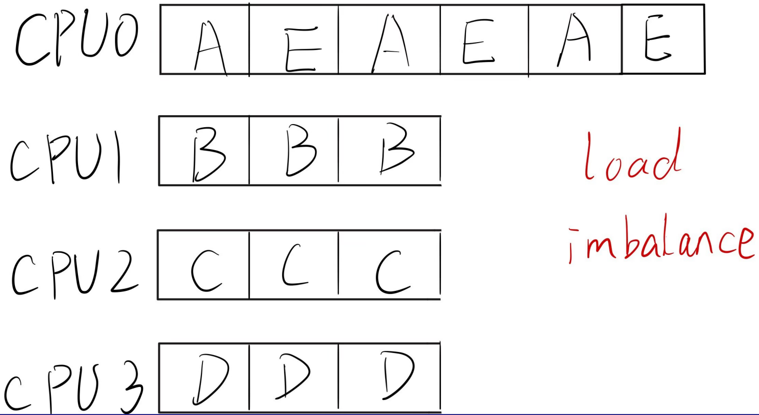
- Reuses data/instructions in CPU cache $\Rightarrow$ reduces cache misses.
- Reduces cache coherence traffic between CPUs.
- Lowers memory latency and improves CPU efficiency.
- Leads to better overall throughput and system performance.

Single Queue Multiprocessor Scheduling (SQMS)



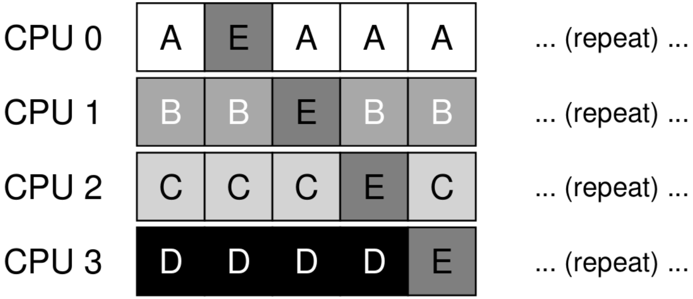
SQMS

what if we force every process can only run on one CPU.



SQMS

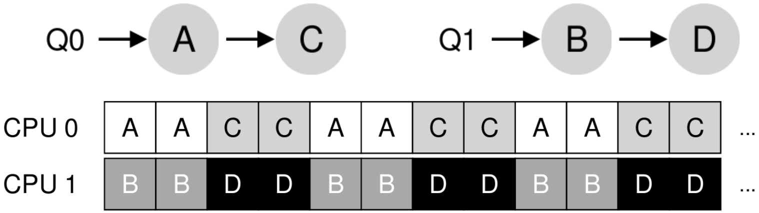
the optimal situation we want to reach.



The difference is that E employs **migration**.
But it's hard to reach!

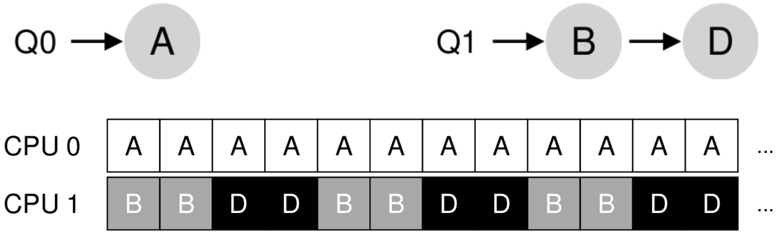
Multi-Queue Multiprocessor Scheduling (MQMS)

Ideal situation:



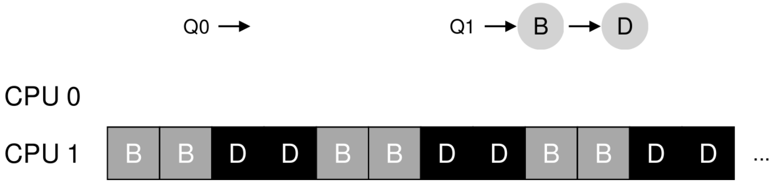
MQMS Load Imbalance

But sometime we may have **Load Imbalance**:



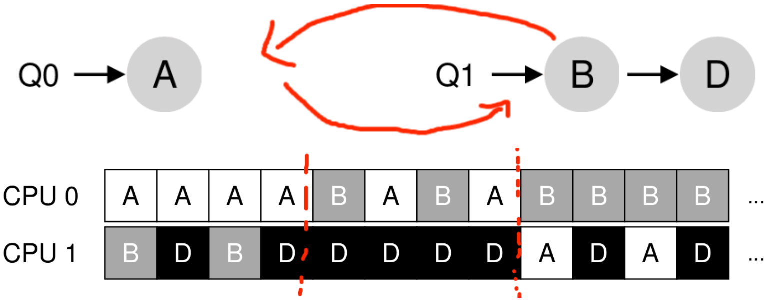
MQMS Load Imbalance

even worse:



MQMS Load Imbalance Solution

solution: **migration** or called **work stealing**.



But it's also hard and complex!

Comparison: SQMS vs MQMS

Aspect	**SQMS**	**MQMS**
Queue Structure	Single global queue	Per-CPU queue
Cache Affinity	Weak	Stronger
Load Balance	Centralized	Decentralized
Scalability	Poor	Good
Overhead Source	Lock contention	Migration checks

Trade-off: Balance between cache affinity, load balance, and scalability.

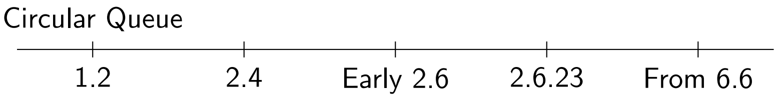
Contents

Introduction

Windows and macOS use MLFQ as their base scheduling policy.

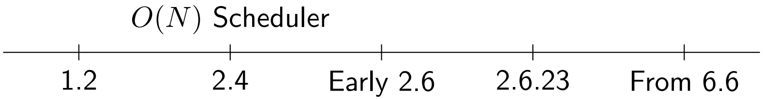
What about Linux?

History



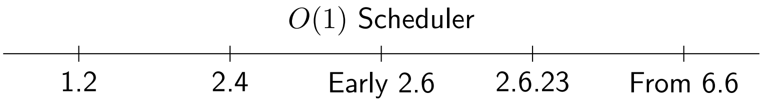
- Early Linux used minimal designs
- The 1.2 Linux used a circular queue with a round-robin policy
- Not complex, but simple and fast

History



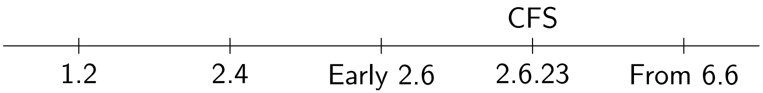
- Iterate over all the tasks
- Apply a goodness function to determine which task runs next
- Not efficient

History



- Enhanced “MLFQ” implementation
- Keep track of runnable tasks in a run queue for each priority level
- Use complex heuristic algorithms to tell whether a task is interactive

History



Linux 2.6.23 was released in 2007.

However, Completely Fair Scheduler (CFS) is still used today!

Side story: before 2.6.21, an algorithm called “Rotating Staircase Deadline Scheduler” was implemented by Con Kolivas. However, it was not merged into the mainline.

CFS: Overview

CFS basically models an "ideal, precise multi-tasking CPU" on real hardware.

In other words, one CPU can run all the N tasks in parallel with the speed $1/N$. Or we say, at any moment, the runtime of all tasks are the same.

However, one CPU core can only run a single task at once. Also, the time of context switching cannot be omitted.

CFS: Virtual Runtime

CFS introduced a variable called “virtual runtime” ($\mathrm{vruntime}$), which accumulates the CPU time consumed by a task.

When a scheduling decision occurs, CFS will pick the process with the *lowest* $\mathrm{vruntime}$ to run next.

How often should a scheduling decision occurs?

Given $\mathrm{sched\_latency}$, if there are N tasks, then the time slice for a task is $\mathrm{sched\_latency}/N$.

Also, there is a minimum value of time slice to avoid too much context switching.

By now, it just looks like round-robin with dynamic time slices.

CFS: Niceness

CFS is a proportional-share scheduler. That is, diffrent tasks have different amount of resources depending on the weight.

The weight of a task is determined by its nice value. The higher the nice value is, the lower the weight is. For a task with nice value n, the weight is $W \cdot (0.8)^n$.

In Linux, $W = 1024$, and the table is as below.

```
static const int prio_to_weight[40] = {
    /* -20 */ 88761, 71755, 56483, 46273, 36291,
    /* -15 */ 29154, 23254, 18705, 14949, 11916,
    /* -10 */  9548,  7620,  6100,  4904,  3906,
    /*  -5 */  3121,  2501,  1991,  1586,  1277,
    /*   0 */  1024,   820,   655,   526,   423,
    /*   5 */   335,   272,   215,   172,   137,
    /*  10 */   110,    87,    70,    56,    45,
    /*  15 */    36,    29,    23,    18,    15,
};
```

CFS: Niceness

$\mathrm{vruntime}$ of tasks with different weights accumulate at different rates. After each time slice,

$$\mathrm{vruntime} \leftarrow \mathrm{vruntime} + \frac{W}{\mathrm{weight}} \cdot \mathrm{runtime}$$

The length of the time slice of each task is also correspondingly changed. For a process k,

$$\mathrm{time\_slice}_k = \frac{\mathrm{weight}_k}{\sum_{i=0}^{N-1} \mathrm{weight}_i} \cdot \mathrm{sched\_latency}$$

By now, it looks like a weighted round-robin with dynamic time slices.

CFS: Red-Black Trees

However, tasks do not always come at the same time. If we use a FIFO queue, there are some problems:

- When a new task gets into the run queue, we need to set vruntime.
    - Usually estimate a middle position.
- A task can sleep for a while, and get back to the run queue.
    - vruntime of it might be greater than the minimum value.
- A task can be moved to another run queue, e.g., run queue of another CPU core.

Hence, we need an efficient data structure.

CFS: Red-Black Trees

Red-Black Trees are a type of balanced binary search tree.

Recap: for each node x in a binary search tree, the key of each node in the left subtree is not greater than the key of x, and the key of each node in the right subtree is not less than the key of x.

In CFS, we maintain Red-Black Trees of tasks ordered by their $\mathrm{vruntime}$.

The time complexity of inserting/removing a node is $O(\log N)$. By caching the leftmost node, the time complexity of query the smallest value is $O(1)$.

Very efficient!

CFS: Red-Black Trees

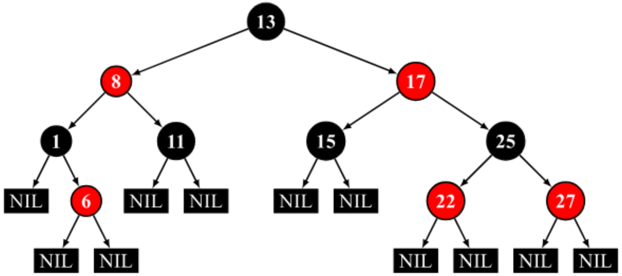


Figure: Example of a Red-Black Tree

CFS: Scheduling Classes

Though different tasks have different shares, sometimes we still prefer to run tasks with high priority immediately. So there are scheduling classes.

When a scheduling decision occurs, we firstly find a non-empty class with highest priority, then choose a task from it.

Different classes can use different policies. For example, if a class consists of tasks with deadlines, then we might prefer to choose the task with the earliest deadline.

CFS: Group Scheduling

Imagine that, User A has 100 task to run, while User B has only 1 task to run.

While it is fair to the tasks, it is unfair to the two users!

We might regard a set of tasks as a group, e.g. processes of a single user, and processes spawned by a single application.

Tasks in a group share a common $\mathrm{vruntime}$. The runtime assigned to the group is then further assigned to the tasks.

CFS: Group Scheduling

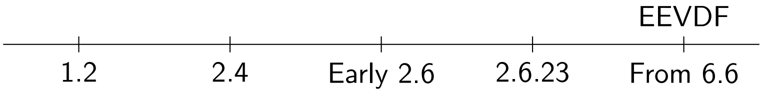
For example:

- Group A: 100
  - Process A1: 250
  - Process A2: 500
- Process B: 120

Then we first choose Group A, and then we choose Process A1 from A.

If we dive further, we can find that the group scheduling is based on “cgroup”, which is also applied in some containerization solutions, e.g., Docker.

Nowadays



From version 6.6, the Linux kernel began moving away from the earlier CFS in favor of a version of “Earliest Eligible Virtual Deadline First” (EEVDF).

EEVDF

EEVDF still uses $\mathrm{vruntime}$, but there are some differences:

- In each round, each task is assigned with a "lag", which indicates its remaining resource in this round.
- We always choose the task whose $\mathrm{lag} > 0$ with earliest virtual deadline.
- By giving latency-sensitive tasks earlier deadlines, the response time is lowered.
- Sleeping tasks remain in the run queue but marked for "deferred dequeue". When a latency-sensitive task wakes up, it can quickly get resources.

Thanks!

References

Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau.
Operating Systems: Three Easy Pieces.
Arpaci-Dusseau Books, 1.10 edition, November 2023.

Edward G. Coffman and Leonard Kleinrock.
Computer scheduling methods and their countermeasures.
In *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference*, AFIPS '68 (Spring), pages 11–21, New York, NY, USA, 1968. Association for Computing Machinery.

References

Online Resources:

- `https://github.com/torvalds/linux`
- `https://developer.ibm.com/tutorials/l-completely-fair-scheduler/`
- `https://docs.kernel.org/scheduler/sched-design-CFS.html`
- `https://docs.kernel.org/scheduler/sched-eevdf.html`