Sistemi Operativi

Corso di Laurea in Informatica

Sapienza Università di Roma

a.a. 2026/2027

Marco Raoul Marini

marini@di.uniroma1.it

Gabriele Tolomei

tolomei@di.uniroma1.it

Lecture 2.3: Virtualization III

CPU Scheduling: MLFQ, Lottery/Stride, and Linux CFS

Opening Thought

> *"The fundamental problem MLFQ tries to address is simple: how can the scheduler learn, over time, the characteristics of the jobs it is running, and use that knowledge to make better scheduling decisions?"*
> *— Remzi & Andrea Arpaci-Dusseau, OSTEP Ch. 8*

In Lecture 2.2 we showed FIFO, SJF, STCF, and RR — each optimal for exactly one metric, each requiring unrealistic assumptions. Today we build schedulers that are **adaptive** (MLFQ), **proportionally fair** (Lottery, Stride), and study the state-of-the-art Linux implementation (CFS).

Lecture Roadmap

This lecture is organized into **10 sections**:

1. **Recap & Motivation** — Why prior policies fail in practice
2. **MLFQ: Core Idea** — Five rules governing priority queues
3. **MLFQ: Traces & Edge Cases** — Gaming, boost, and I/O interaction
4. **MLFQ in Real Systems** — Solaris, FreeBSD, Linux O(1), Windows
5. **Proportional Share: Motivation** — A different goal
6. **Lottery Scheduling** — Probabilistic fairness via ticket draws
7. **Stride Scheduling** — Deterministic fairness via virtual clocks
8. **Linux CFS** — The Completely Fair Scheduler in depth
9. **Worked Examples & Exam Problems** — Full numerical traces
10. **Summary & Key Takeaways**

§ 1 — Recap & Motivation

What Lecture 2.2 Gave Us

Policy	Optimizes	Key Limitation
FIFO	Simplicity	Convoy effect
SJF	Avg turnaround (simultaneous arrivals)	Needs known runtimes
STCF	Avg turnaround (varying arrivals)	Needs known runtimes; starvation
Round Robin	Response time	Poor turnaround

Every algorithm trades one metric for another. None handles the full reality of a modern OS workload.

The Core Dilemma

Two goals pull the scheduler in opposite directions:

- **Turnaround time** is minimized by running the shortest job first (SJF/STCF).
- **Response time** is minimized by frequent context switching (RR).

Both SJF and STCF require the OS to know the exact remaining runtime of every job — which is **impossible in practice** (Halting Problem).

We need a scheduler that **learns** job characteristics from observed behavior, without prior knowledge.

The Real Workload Mix

Modern systems simultaneously run two fundamentally different classes:

Interactive Jobs

- Short bursts of CPU followed by waiting (keyboard input, network, disk).
- Need **low response time** — users notice delays above ~100 ms.
- Examples: text editors, web browsers, shells.

Batch / CPU-bound Jobs

- Long CPU computations with minimal I/O.
- Need **high throughput / good turnaround time**.
- Examples: compilers, video encoders, scientific simulations.

A good scheduler must serve both classes **simultaneously**.

The Key Insight

> ***If a job behaves like a short interactive job, treat it like one. If it behaves like a long CPU-bound job, gradually lower its priority.***

This is the fundamental intuition behind the **Multi-Level Feedback Queue (MLFQ)**:

- Use *multiple queues*, each with a different priority level.
- Move jobs *between* queues based on **observed CPU consumption**.
- No prior knowledge of job length required — the scheduler *learns*.

§ 2 — MLFQ: The Core Idea

MLFQ Structure

The **Multi-Level Feedback Queue** consists of multiple priority queues:

```
┌──────────────────────────────────────┐
│ Q0 (Highest Priority) — short RR     │ ← New jobs enter here
├──────────────────────────────────────┤
│ Q1 (High Priority)    — short RR     │
├──────────────────────────────────────┤
│ Q2 (Medium Priority)  — longer RR    │
├──────────────────────────────────────┤
│ Q3 (Lowest Priority)  — long/FIFO    │ ← Long CPU-bound jobs end here
└──────────────────────────────────────┘
```

- Each level has its own **scheduling policy** (typically RR) and **time quantum**.
- Jobs move *down* when they exhaust their allotment; *up* on a periodic boost.

The Five MLFQ Rules

All of MLFQ's behavior is governed by exactly five rules:

1. **Rule 1:** If Priority(A) > Priority(B) → A runs, B does not.
2. **Rule 2:** If Priority(A) = Priority(B) → A and B run in Round Robin.
3. **Rule 3:** When a job enters the system, it is placed at the **highest** priority queue.
4. **Rule 4:** Once a job uses up its **time allotment** at a given level (regardless of how many times it voluntarily yielded the CPU), its priority is **reduced** by one level.
5. **Rule 5 (Priority Boost):** After some period S, move **all** jobs to the topmost queue.

Rule 1: Priority Determines Who Runs

> ***If Priority(A) > Priority(B), A runs and B does not.***

- The scheduler always selects the highest-priority **non-empty** queue.
- Jobs in lower queues are completely blocked while any higher-priority job is runnable.
- This guarantees interactive jobs (which stay at high priority) get immediate CPU access.

```
Q0: [Interactive job X]   ← runs immediately
Q1: [                ]
Q2: [Batch job Y     ]      ← must wait for Q0 to empty
```

Rule 2: Same Priority → Round Robin

> ***If Priority(A) = Priority(B), A and B run in RR using the time slice of that queue level.***

- Multiple jobs at the same level share the CPU fairly within that level.
- Time slices may differ per level (shorter at top, longer at bottom).

```
Q0 (1ms slice):   A → B → A → B → ...
Q1 (5ms slice):   C → D → C → D → ...
Q2 (20ms slice):  E → ...
```

This allows fairness among jobs of the same *behavioral class*.

Rule 3: New Jobs Enter at the Top

> ***When a job enters the system, it is placed at the highest priority queue.***

- MLFQ gives every new job the **benefit of the doubt**.
- It assumes the new job might be short or interactive.
- If the assumption turns out to be wrong, the job is demoted automatically.

This implements an "*innocent until proven guilty*" philosophy:

- A newly submitted video encoder is initially treated like a keypress handler — until it reveals its CPU-hungry nature over time.

Rule 4: Demotion on Allotment Exhaustion

> ***Once a job uses up its time allotment at a given level (cumulative across all yields), its priority is reduced by one level.***

Key term — **time allotment**: the *total* CPU time a job may consume at priority level k before being demoted.

This is **cumulative**: the clock keeps running even when the job voluntarily yields the CPU (e.g., for I/O). This distinction is critical for preventing gaming.

Rule 4: Time Slice vs. Time Allotment

These two terms are often confused:

Term	Meaning
Time slice (quantum)	Maximum CPU time granted in a *single* scheduling event
Time allotment	Total CPU time permitted at *one priority level* before demotion

A job with a 1 ms time slice and a 5 ms allotment at level Q1 may be scheduled up to **5 times** at Q1 before being demoted — as long as each run uses ≤ 1 ms.

The allotment clock ticks *regardless* of whether the job yields voluntarily.

Rule 5: Priority Boost

> ***After some time period S, move all jobs to the topmost queue.***

This rule solves two critical problems:

1. Starvation prevention: Without boosting, a batch job at the lowest queue is never scheduled as long as interactive jobs keep arriving at Q0.

2. Changed job behavior: A batch job that completes its heavy computation and then becomes interactive (e.g., opens a GUI) can reclaim high priority after the next boost.

```
Every S time units:  ALL jobs ——————————→ Q0
```

Rule 5: Choosing the Boost Period S

The boost period S is a critical parameter — OSTEP calls it a "voodoo constant":

- **S too small:** Interactive jobs are frequently interrupted by boosted batch jobs; response time degrades.
- **S too large:** Batch jobs starve for too long before being boosted; fairness degrades.

Real system values:

- Solaris: depends on load and priority table configuration.
- Linux O(1): dynamic, typically between 100 ms and a few seconds.
- MLFQ research prototypes: 1 s is a common default.

✅ Checkpoint 1 — MLFQ Core Rules

Test your understanding:

1. State all five MLFQ rules from memory. Which rule prevents starvation of long-running jobs?
2. Explain the difference between **time slice** and **time allotment**. Why does Rule 4 use the allotment (not just the slice)?
3. Why does MLFQ place new jobs at the *highest* priority queue rather than the lowest?
4. A batch job has been at Q3 (lowest) for 10 minutes. Which rule eventually gives it CPU time, and how?
5. What is the "voodoo constant" S? What happens at either extreme of its value?

§ 3 — MLFQ: Traces & Edge Cases

Trace 1: Interactive-Only Workload

Setup: 3 queues: Q0 (allotment = 1 ms, slice = 1 ms), Q1 (allotment = 4 ms), Q2 (FIFO).
One interactive job **A**: does 0.5 ms CPU, then waits 4.5 ms for I/O. Repeating.

```
Time: 0   0.5   5   5.5   10  10.5   15  15.5 ...
Q0:   [A] [I/O] [A]
Q1:                 [I/O] [A] [I/O] [A] ...
Q2:
```

- Although A voluntarily blocks before consuming its entire time slice, its CPU usage still counts toward the **time allotment**.
- After accumulating 1 ms of CPU time at Q0, A exhausts its Q0 allotment and is demoted to Q1.
- The I/O wait improves responsiveness but **does not reset the allotment counter**.

Trace 2: Lone Batch Job

Setup: 3 queues: Q0 (allotment = 1 ms), Q1 (allotment = 4 ms), Q2 (FIFO).
One batch job **B**: needs 20 ms of CPU, no I/O.

```
t = 0–1 ms:   Q0: B runs 1 ms → allotment exhausted → demoted to Q1
t = 1–5 ms:   Q1: B runs 4 ms → allotment exhausted → demoted to Q2
t = 5–20 ms:  Q2: B runs remaining 15 ms until completion
```

- B sinks through the queues quickly.
- Any new interactive job arriving during those 20 ms would run at Q0, preempting B.
- B finishes its work as a background task. This is the **intended behavior**.

Trace 3: Interactive + Batch Concurrent

Setup: Same 3-queue structure. **A** (interactive: 0.5 ms CPU / 4.5 ms I/O). **B** (batch: 20 ms CPU). Both arrive at t = 0.

```
t=0–0.5:   Q0: A runs → blocks for I/O until t=5 (0.5 ms out of 1 ms of Q0 allotment used)
t=0.5–1.5: Q0: B runs for its entire 1 ms allotment (time slice = 1 ms, allotment = 1 ms)
           → demoted to Q1
t=1.5–5:   Q1: B continues executing (3.5 ms of its 4 ms Q1 allotment consumed)
t=5:       A completes I/O and becomes runnable.
           Since A is in Q0 and B is in Q1, B is preempted.
t=5–5.5:   Q0: A runs another 0.5 ms (Q0 allotment reaches 1 ms total)
           → blocks for I/O until t=10
           → demoted to Q1
t=5.5–6:   Q1: B resumes (Q1 allotment reaches 4 ms total)
           → demoted to Q2
...
```

- B is CPU-bound and quickly consumes its allotments, eventually reaching Q2.
- After its second 0.5 ms CPU burst (at t=5), A exhausts its 1 ms Q0 allotment and is demoted to Q1 and, eventually, to Q2.
- To keep good responsiveness of I/O-bound tasks and prevent starvation of CPU-bound tasks, boost priority occurs.

Trace 4: Priority Boost Preventing Starvation

Setup: B has been demoted to Q2. Interactive jobs keep arriving at Q0, fully occupying the CPU. Boost period S = 10 ms.

```
t=0–10:   Q0 busy with interactive jobs; B at Q2 never scheduled
t=10:     PRIORITY BOOST → B moves to Q0
t=10–11:  B gets 1 ms at Q0 (some progress made!)
t=11:     B exhausted Q0 allotment → demoted to Q1
t=11–15:  B gets 4 ms at Q1 (more progress!)
t=15:     B demoted to Q2 again
t=20:     PRIORITY BOOST → B moves to Q0 again
```

Every S ms, B is guaranteed to run. **Starvation eliminated.**

Gaming Attack: I/O Manipulation

Why using cumulative allotment? A malicious process can try to **game** a naive MLFQ implementation:

Vulnerable Rule 4 (old): "Demote only if the job uses its full time slice without yielding."

Attack: Issue a spurious I/O call just before the slice expires:

```
Slice = 10 ms
At t = 9.99 ms: write(fd, "", 0);   // fake I/O → voluntary yield
```

- The job yields at 9.99 ms, never triggering demotion.
- It re-enters the scheduler at Q0 with a **fresh** allotment.
- It monopolizes Q0 indefinitely, starving all other jobs.

Fix (current Rule 4): Use **cumulative allotment**. The clock ticks regardless of voluntary yields. Gaming stops.

Gaming Attack: Process Self-Fork

Another attempt: fork a child just before allotment expiry.

```
// Near end of allotment:
if (near_allotment_limit()) {
    if (fork() == 0) {
        // Child starts at Q0 with fresh allotment
        continue_work();
        exit(0);
    }
    // Parent exits (never gets demoted)
    exit(0);
}
```

Mitigations:

- Track CPU usage **per user (UID)**, not per process. Children inherit parent's allotment credit.
- Use an **accounting daemon** that redistributes priority based on total CPU consumed per user.

MLFQ: Variable Time Slices by Level

Refined MLFQ designs use **different time slices at each level**:

Level	Time Slice	Reasoning
Q0	1 ms	Fast context switch; interactive jobs need it
Q1	5 ms	Moderate overhead acceptable
Q2	20 ms	Batch jobs benefit from longer quanta
Q3	∞ (FIFO)	Full run-to-completion; minimal overhead

- Interactive jobs rarely need more than 1 ms per burst → stay in Q0 as much as possible.
- Batch jobs settle in Q2/Q3 with large quanta → fewer context switches.
- Context switch overhead is **amortized** more at lower priority levels.

MLFQ: How Many Queues?

Choosing the number of priority levels:

- **Too few (e.g., 2):** Coarse separation; interactive and batch jobs share levels.
- **Too many (e.g., 100):** Administrative complexity; tuning allotments per level becomes impractical.

Real systems:

System	Levels
Solaris Time-Sharing	60
Windows NT	32 (variable range: 1–15)
Linux O(1)	140
BSD Unix	32

Typical sweet spot: **8–32 levels**.

✅ Checkpoint 2 — MLFQ Traces & Edge Cases

Test your understanding:

1. Trace the MLFQ timeline for: A (interactive: 1 ms CPU / 9 ms I/O) and B (batch: 15 ms CPU), 3 levels with allotments 2 ms, 6 ms, ∞. Draw the timeline for t = 0 to t = 30 ms.
2. Explain why the I/O gaming attack works against the old Rule 4, and why cumulative allotment defeats it.
3. A batch job B is at Q3. The boost period is S = 50 ms. B needs 200 ms of total CPU. How many boosts occur before B completes (assuming interactive jobs use 80% of each S-period)?
4. Why do lower-priority levels use longer time slices?

§ 4 — MLFQ in Real Systems

Solaris Time-Sharing (TS) Scheduler

Solaris implements MLFQ using a **dispatch table** with 60 priority levels (0–59):

Priority	Time Quantum	Priority After Expiry
59	20 ms	→ 49
49	40 ms	→ 39
39	80 ms	→ 29
29	120 ms	→ 19
...	...	...
0	200 ms	stays at 0

- Quantum **increases** as priority **decreases** → batch jobs get longer runs.
- A job returning from sleep gets a **priority bump** → interactive detection.
- Table is stored in `/etc/dispadmin` and is administrator-configurable.

FreeBSD / BSD UNIX Scheduler

BSD uses a priority range 0–255 (lower number = higher priority):

- **0–63:** Kernel threads (highest).
- **64–127:** System processes.
- **128–255:** User processes (MLFQ-like behavior).

Priority is computed using an **exponential decay formula**:

$$P_i = \text{base} + \frac{\text{cpu_usage}_i}{2}$$

- `cpu_usage` tracks recent CPU consumption; it is **halved** every second (decay).
- A process that was CPU-heavy gains a high `cpu_usage` → high P_i → lower priority.
- A process that slept gains a low `cpu_usage` → lower P_i → higher priority.

Decay acts as an implicit, continuous priority boost — no discrete S needed.

Linux: The O(1) Scheduler (Kernel 2.4–2.6.22)

The **O(1) Scheduler** was Linux's MLFQ implementation before 2007:

- **140 priority levels** (0–139).
    - 0–99: Real-time (FIFO and RR policies).
    - 100–139: Normal user processes (MLFQ-like).
- Each level has a **separate run queue**.
- A **bitmask** (140-bit integer) marks which levels are non-empty.
- Finding the highest non-empty level: `find_first_set_bit(bitmask)` → **O(1)** thanks to special hardware instructions (e.g., x86's `BSF` / `LZCNT` or ARM's `CLZ`).

Interactive detection: Processes that sleep often (interactive) get a **priority bonus** (+5 max). Processes that use CPU heavily get a **penalty** (–5 max).

Linux O(1): Active and Expired Arrays

A clever implementation trick to avoid O(n) requeuing:

```
Two arrays of 140 queues each:
  Active   array: queues containing runnable processes
  Expired  array: queues of processes that have used their quanta

When a process exhausts its quantum:
  → Move it to the Expired array (recalculate priority)

When Active array is empty:
  → Swap Active and Expired pointers (O(1) operation!)
  → Continue scheduling from the new Active array
```

This avoids scanning all processes and achieves true O(1) scheduling decisions.

Windows NT Scheduler

Windows uses a **32-level priority system**:

```
Levels 31–16: Real-time priorities (fixed, non-adjustable by apps)
Levels 15–1:  Variable priorities (dynamic, MLFQ-like behavior)
Level  0:     Zero-page thread (system idle)
```

MLFQ-like dynamics for variable priorities:

- When a process's **time quantum expires** → priority decremented by 1.
- When a process is **awakened from I/O wait** → priority incremented (up to +2).
- The **foreground window's process** gets a priority boost (configurable: ×1, ×1.5, or ×2 quanta).

Windows MLFQ uses implicit adjustments rather than discrete queue levels.

Comparing Real Schedulers

System	Mechanism	Levels	Boost Mechanism
Solaris TS	Dispatch table	60	Sleep detection (wakeup bump)
FreeBSD	Decay formula	128 (user)	Exponential `cpu_usage` decay
Linux O(1)	Bitmask + 2 arrays	140	Sleep bonus / CPU penalty
Linux CFS	Red-black tree	Continuous	`vruntime` decay (studied next)
Windows NT	Dynamic adjustments	32	I/O wakeup + foreground boost

All systems solve the same core problem: **distinguish interactive from batch jobs without knowing runtimes in advance.**

✅ Checkpoint 3 — MLFQ in Real Systems

Test your understanding:

1. In Solaris, a process exhausts its quantum at priority 49. Where does it go? What is its new quantum?
2. How does FreeBSD approximate MLFQ without discrete queues? What data decays?
3. What is the O(1) property of the Linux O(1) Scheduler? What data structure enables it?
4. Describe the active/expired array swap trick. Why is this O(1)?
5. Windows boosts the foreground window's priority. Which MLFQ rule does this conceptually implement?

§ 5 — Proportional Share: Motivation

A Different Scheduling Goal

MLFQ optimizes **performance** (turnaround) and **response time** through adaptive priority.

A complementary goal is **proportional fairness**:

> *Give each process a **guaranteed fraction** of the CPU, proportional to its configured weight or budget.*

Use cases:

- A database server should get 60% of the CPU; a web server 30%; a monitoring daemon 10%.
- Virtual machines in a hypervisor each need a CPU guarantee.
- Real-time audio needs a guaranteed fraction to avoid stuttering.

Why MLFQ Cannot Guarantee Proportions

MLFQ is heuristic: it cannot promise "Process A always gets exactly 2× the CPU of Process B":

- A high-priority interactive job can monopolize the CPU as long as it keeps arriving.
- The boost mechanism is coarse and periodic.
- There is no knob to say "give this process exactly 33%."

We need a **different paradigm**: **Proportional Share Scheduling**.

Two Classical Approaches

We study two classical proportional-share algorithms:

1. **Lottery Scheduling** (Waldspurger & Weihl, 1994)
    - Randomized: assign tickets; draw a random winner at each scheduling event.
    - Simple to implement; handles new/departing processes gracefully.
2. **Stride Scheduling** (Waldspurger, 1995)
    - Deterministic: each process advances a virtual-time counter by its stride.
    - More precise; eliminates statistical variance of the lottery.

Both are covered in OSTEP Chapter 9.

§ 6 — Lottery Scheduling

The Lottery Metaphor

Imagine 100 lottery tickets distributed among all processes:

```
Process A:  75 tickets  (75% of CPU)
Process B:  25 tickets  (25% of CPU)
Total:     100 tickets
```

At each scheduling event:

1. The scheduler draws a **random ticket number** (1–100).
2. Whichever process holds the winning ticket gets the CPU.
3. Over many scheduling decisions, each process receives CPU time proportional to its ticket count due to the Law of Large Numbers. Specifically, if X_i denotes the outcome of the i-th lottery draw (e.g., whether a given process wins the draw), then the observed fraction of wins converges to the expected fraction as the number of draws increases: $\bar{X}_n \to E[X_i],\ n \to \infty$.

No priority queues. No demotion. Just a random draw.

Lottery: Formal Definition

Definition: Each process holds a set of **lottery tickets**. The scheduler selects the next process by drawing a random ticket uniformly from all outstanding tickets.

$$P(\text{process } i \text{ wins}) = \frac{\text{tickets}_i}{\sum_j \text{tickets}_j}$$

- **Tickets** represent CPU entitlement — not absolute time, but *proportion*.
- The more tickets, the higher the expected CPU share.
- Over many scheduling rounds, actual CPU allocation **converges** to the target proportions.

Lottery: Simple Example

Workload: Process A has 100 tickets; Process B has 100 tickets. Total = 200.

Each wins with probability 50%. Over 100 rounds, A gets ≈ 50 quanta, B gets ≈ 50 quanta.

Changing proportions is trivial: Give A 200 tickets, keep B at 100. Now:

$$P(A) = \frac{200}{300} \approx 67\%, \quad P(B) = \frac{100}{300} \approx 33\%$$

No queue restructuring. No priority recalculation. Just change the ticket count.

Lottery: Handling New Processes

When process C (with 50 tickets) joins:

```
Before: A=100, B=100  → total 200
After:  A=100, B=100, C=50  → total 250
```

Updated shares:

- A: 100/250 = **40%**
- B: 100/250 = **40%**
- C: 50/250 = **20%**

All existing processes proportionally cede some CPU to C. No queue reorganization needed — this is a major elegance advantage over MLFQ.

Ticket Inflation and Deflation

Because ticket counts are relative, processes can be dynamically re-prioritized:

- **Inflation:** Give a process more tickets → increases its CPU share.
- **Deflation:** Remove tickets from a process → decreases its CPU share.

Important constraint: Only *trusted* code should be allowed to issue tickets. In a multi-user environment, the OS assigns fixed ticket budgets to users; processes cannot grant themselves extra tickets.

Ticket Transfer

A process can **lend** its tickets to another process:

```
Client C (owns 100 tickets) sends request to Server S.
C transfers its 100 tickets to S for the duration of the request.
S now has higher priority → serves C's request faster.
When S responds, S returns the 100 tickets to C.
```

This is **ticket transfer**: it enables *client-server priority inheritance*, preventing priority inversion (a synchronization concept revisited in Module 3).

Ticket Currency

Lottery supports a **multi-currency** model for multi-user environments:

```
User 1 budget: 100 global tickets
  └── Process A: 500 (local) → 50 global tickets
  └── Process B: 500 (local) → 50 global tickets

User 2 budget: 100 global tickets
  └── Process C: 100 (local) → 100 global tickets
```

Each user manages their own budget in their own **local currency**. The OS converts to global tickets proportionally. Users can distribute CPU among their own processes without affecting other users' shares.

Lottery: Implementation

```
int total_tickets = sum_all_tickets();
int winner = rand() % total_tickets;   // draw

int current = 0;
struct process *p;
list_for_each(p, run_queue) {
    current += p->tickets;
    if (current > winner) {
        schedule(p);
        break;
    }
}
```

- **Time complexity:** O(n) per scheduling event (single pass over the run queue).
- **Optimization:** Sort by tickets descending → winners found sooner on average.
- **Further optimization:** Ticket tree (sorted + binary search) → O(log n).

Lottery: Convergence Analysis

How quickly does lottery converge to target proportions?

Let process A have target proportion p (tickets_A / total). After n scheduling rounds:

- Expected CPU for A: $E[X_A] = np$ rounds.
- Standard deviation: $\sigma = \sqrt{np(1-p)}$.
- Absolute error: $X_A \approx np \pm \sigma$.
- Relative error: $\sigma/np = \dfrac{1-p}{\sqrt{np}}$ — decreases proportionally as $\dfrac{1}{\sqrt{n}}$.

Key insight: Lottery converges *slowly* for short jobs. Over a small number of rounds (e.g., n = 10), actual allocation may deviate significantly from the target.

Lottery Scheduling: Unfairness Over Short Intervals

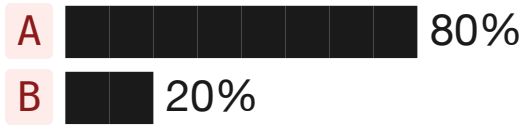
Scenario: Two processes, A and B, have equal tickets (50% each). Consider a short scheduling interval (e.g., 10 CPU quanta).

$$U = \frac{\min(C_A, C_B)}{\max(C_A, C_B)},$$

where C_A and C_B are the CPU quanta received by each process, and $U = 1$ denotes **perfect faireness**.

Possible random outcome 🎲

Process	CPU Quanta
A	8
B	2



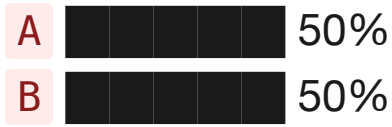
$$U = \frac{\min(8, 2)}{\max(8, 2)} = \frac{2}{8} = \frac{1}{4} = \mathbf{0.25}$$

Lottery Scheduling: Unfairness Over Short Intervals

Scenario: Two processes, A and B, have equal tickets (50% each). Consider a short scheduling interval (e.g., 10 CPU quanta).

Balanced (desired) allocation

Process	CPU Quanta
A	5
B	5



$$U = \frac{\min(5, 5)}{\max(5, 5)} = \mathbf{1.0}$$

> ***Key Takeaway:*** *Random selection can cause temporary CPU-allocation imbalances. With stride scheduling (next section), U approaches 1.0 even for short jobs.*

Lottery: Summary

Property	Value
Proportional guarantee	Probabilistic (converges as $1/\sqrt{n}$)
Implementation complexity	Very low
Adding/removing processes	O(1) — just update ticket count
Priority change	O(1)
Short-job fairness	Poor (high variance)
Starvation guarantee	No starvation (every process has > 0 tickets)
Gaming resistance	High (random; no exploitable pattern)

✅ Checkpoint 4 — Lottery Scheduling

Test your understanding:

1. Process A has 40 tickets, B has 30, C has 30. What is the probability each wins a single draw?
2. D (50 tickets) joins. Recalculate all shares.
3. Derive the relative error of lottery scheduling after n rounds. What does this imply for short-lived jobs?
4. Explain ticket transfer. In what client-server scenario is it useful?
5. Can a malicious user-space process give itself unlimited tickets? What prevents this?

§ 7 — Stride Scheduling

The Problem with Randomness

Lottery is simple and elegant — but randomness causes variance:

- For long-running jobs over many quanta, averages converge.
- For short-lived or latency-sensitive jobs, actual CPU may deviate significantly from the target.

Stride Scheduling (Waldspurger, 1995) eliminates this by using a **deterministic virtual-time** mechanism — no random number generator needed.

Stride Scheduling: Core Concept

Each process is assigned two values:

$$\text{stride}_i = \frac{W}{\text{tickets}_i}, \quad \text{pass}_i = 0 \text{ (initially)}$$

where W is a large constant (e.g., 10,000).

Scheduling rule:

1. Always run the process with the **minimum pass** value.
2. After running: $\text{pass}_i \leftarrow \text{pass}_i + \text{stride}_i$

A process with *more* tickets has a *smaller* stride → its pass advances slower → it wins more often. This is the inverse relationship that delivers proportionality.

Stride: Example Setup

Workload: W = 10,000.

- Process A: 100 tickets → stride$_A$ = 100
- Process B: 50 tickets → stride$_B$ = 200
- Process C: 250 tickets → stride$_C$ = 40

Initial pass values: A = 0, B = 0, C = 0.

Expected CPU shares: A = 25%, B = 12.5%, C = 62.5%.

Stride: Step-by-Step Trace (Part 1)

Tick	Winner	A pass	B pass	C pass
—	(start)	0	0	0
1	A (tie; A chosen first)	**100**	0	0
2	B (tie B/C; B chosen)	100	**200**	0
3	C	100	200	**40**
4	C	100	200	**80**
5	C	100	200	**120**
6	A	**200**	200	120

Stride: Step-by-Step Trace (Part 2)

Tick	Winner	A pass	B pass	C pass
6	A	200	200	120
7	C	200	200	**160**
8	C	200	200	**200**
9	A (three-way tie)	**300**	200	200
10	B	300	**400**	200
11	C	300	400	**240**
12	C	300	400	**280**

Over 12 ticks: C ran 6×, A ran 3×, B ran 1× so far. Already close to 62.5/25/12.5% target.

Stride: Convergence Guarantee

Stride scheduling provides **bounded unfairness**:

> *The deviation between any process's actual CPU allocation and its target allocation is bounded.*

Formally: $|\text{actual fraction}_i - \text{target fraction}_i| \leq \frac{C_i}{n}$, where C_i is a constant that bounds the absolute allocation error and n is total number of scheduling rounds.

As $n \to \infty$, the deviation $\to 0$. This bound $O(1/n)$ is **much tighter** than lottery's $O(1/\sqrt{n})$ convergence.

Stride: Handling New Processes

Problem: A new process joining with pass = 0 has the minimum pass value and immediately **monopolizes the CPU** until its pass catches up to the other processes.

Fix: Set the new process's initial pass to the **current minimum pass value** of all running processes:

```
void admit(Process *new_p, list_t *run_queue) {
    new_p->stride = W / new_p->tickets;
    new_p->pass   = min_pass(run_queue);  // avoid monopolization
    list_add(run_queue, new_p);
}
```

The new process starts from the current scheduling "frontier" — it gets a fair share going forward without catching up for the past.

Stride vs. Lottery: Side-by-Side

Property	Lottery	Stride
Proportional guarantee	Probabilistic (~$1/\sqrt{n}$)	Deterministic (bounded by one stride)
Short-job fairness	Poor	Good
New process handling	Trivially easy	Must initialize pass to min-pass
Global state needed	No	Yes (min-pass tracking)
Implementation complexity	Very low	Low
Randomness	Required	Not needed
Ticket changes mid-run	Easy	Adjust stride; leave pass unchanged

✅ Checkpoint 5 — Stride Scheduling

Test your understanding:

1. A has 200 tickets, B has 100. W = 10,000. Compute stride_A, stride_B.
2. Trace 8 scheduling ticks starting from $\text{pass}_A = 0$, $\text{pass}_B = 0$.
3. A new process C (100 tickets) joins at tick 5 when A.pass = 150, B.pass = 300. What should C's initial pass be?
4. Compare the fairness bound of stride vs. lottery. Which is tighter and by how much asymptotically?
5. How does stride scheduling relate to Linux CFS? (Hint: think about `vruntime` and the scheduling rule.)

§ 8 — Linux CFS

Why Linux Moved from O(1) to CFS

The O(1) Scheduler worked well for years but had two key weaknesses:

1. **Interactivity heuristic was fragile:** The bonus/penalty system for detecting interactive vs. batch was complex, hard to tune, and sometimes misbehaved under mixed workloads.
2. **Unfairness at scale:** With many processes, small rounding errors in priority adjustments accumulated, causing some processes to receive systematically more or less CPU than intended.

Ingo Molnár introduced the **Completely Fair Scheduler (CFS)** in Linux kernel **2.6.23 (October 2007)**, replacing the O(1) scheduler. CFS takes an entirely different approach.

CFS Design Philosophy

CFS does **not** use priority queues. It models an ideal "perfectly fair" CPU:

> ***Ideal multi-tasking CPU:*** *With n processes, each process runs at $\frac{1}{n}$ of CPU speed simultaneously — everyone makes progress at the same rate.*

Since we can't literally run all processes simultaneously, CFS *models* this ideal by tracking **how much CPU time each process has received** and always scheduling the process that is **furthest behind**.

This is implemented via the **virtual runtime (`vruntime`)** counter.

The `vruntime` Counter

Every process has a `vruntime` (virtual runtime) value, measured in nanoseconds.

`vruntime` advances as the process runs:

$$\texttt{vruntime} \mathrel{+}= \Delta t \cdot \frac{W_0}{w_i}$$

Where:

- Δt = real time the process ran (ns).
- W_0 = weight of a process at nice = 0 (baseline: 1024).
- w_i = weight of process i (determined by its nice value).

Key insight: A higher-priority (lower nice) process has a larger w_i → its `vruntime` advances **slower** → it appears "further behind" → it gets scheduled more often.

Nice Values and Weights

Linux maps nice values (–20 to +19) to weights via a precomputed table:

Nice	Weight	Effect
–20	88761	Highest priority — vruntime advances ~87× slower than nice 0
–10	9548	High priority
0	**1024**	**Default — baseline**
+10	110	Low priority — vruntime advances ~9× faster than nice 0
+19	15	Lowest priority — vruntime advances ~68× faster than nice 0

The table is designed so that a **one nice-level change ≈ 10% CPU difference**.

`vruntime` and Stride: The Equivalence

CFS is conceptually **equivalent to stride scheduling**:

Stride Scheduling	Linux CFS
Pass counter	`vruntime`
Stride = W / tickets	$\Delta t \cdot W_0 / w_i$ (per-tick advance)
Schedule min-pass process	Schedule min-vruntime process
Tickets → CPU share	nice weight w_i → CPU share

CFS *is* stride scheduling, implemented with a red-black tree instead of a sorted list.

The Red-Black Tree

CFS stores all runnable processes in a **self-balancing red-black tree**, keyed by `vruntime`:

```
          [vruntime=100]
         /               \
  [vruntime=80]        [vruntime=150]
   /         \
[vr=70]   [vr=85]
```

- **Leftmost node** = process with minimum `vruntime` = **next to run**.
- Linux caches the leftmost node pointer (`rb_leftmost`) → O(1) next-process lookup.
- Insert / delete: O(log n).

Practically, with up to a few hundred runnable processes, log n ≈ 8 operations — negligible.

CFS: Selecting the Next Task

```
// Simplified CFS pick_next_task:
struct task_struct *pick_next_task_fair(struct rq *rq) {
    struct sched_entity *se;

    // Leftmost node = minimum vruntime
    se = __pick_first_entity(cfs_rq);  // O(1): cached pointer
    return task_of(se);
}
```

After running for a quantum Δt:

```
se->vruntime += calc_delta_fair(delta_exec, se);
// Re-insert se into the red-black tree at new vruntime position
// Update leftmost cache if needed
```

CFS: Scheduling Latency and Minimum Granularity

Two important CFS parameters:

Target Scheduling Latency (`sched_latency`)

- The time within which every runnable process should run at least once.
- Default: **6 ms** (adjustable via `/proc/sys/kernel/sched_latency_ns`).
- With n processes: each gets a quantum of $\frac{\text{sched_latency}}{n}$.

Minimum Granularity (`sched_min_granularity`)

- The minimum quantum regardless of how many processes are running.
- Default: **0.75 ms** (prevents excessive context switching with many processes).
- If n is large: effective latency = $n \times \text{min_granularity} > \text{sched_latency}$.

CFS: Wakeup Preemption

When a sleeping process wakes up (e.g., I/O completes), CFS must decide whether to preempt the current process:

```
// wakeup_preempt_entity: should woken 'se' preempt current 'curr'?
static bool wakeup_preempt_entity(struct sched_entity *curr,
                                  struct sched_entity *se) {
    s64 gran = sched_vslice(cfs_rq, curr);
    // Preempt if se.vruntime < curr.vruntime - gran
    return entity_before(se, curr) &&
           (curr->vruntime - se->vruntime) > gran;
}
```

- A woken process preempts the current one only if its `vruntime` is sufficiently smaller.
- The granularity threshold prevents excessive preemption (avoids "thrashing").

CFS: Group Scheduling and cgroups

Linux CFS supports **group scheduling** via cgroups (control groups):

```
System CPU
  ├── cgroup "database" (weight 512)
  │     ├── postgres (nice 0)
  │     └── pgbouncer (nice 0)
  └── cgroup "webserver" (weight 512)
        ├── nginx (nice 0)
        └── php-fpm (nice 5)
```

- Each cgroup has a weight; CPU is divided proportionally between groups first.
- Within each group, CFS divides fairly among member processes.
- This enables **container** CPU isolation (Docker, Kubernetes use cgroups).

CFS: Bandwidth Control

For hard CPU limits (not just proportional shares), CFS supports **bandwidth control**:

```
cgroup quota: 50 ms every 100 ms period
→ processes in this cgroup get at most 50% CPU
```

Implementation: a **throttle/unthrottle** mechanism.

- Each period, the cgroup gets a quota of CPU time.
- When the quota is exhausted, all processes in the group are **throttled** (removed from run queues).
- At the start of the next period, they are **unthrottled** (re-added).

Used in Kubernetes to enforce CPU limits on pods.

CFS on Multicore: Per-CPU Run Queues

Each CPU core has its own CFS run queue:

```
CPU 0 rq: [P1, P3, P5]  (sorted by vruntime)
CPU 1 rq: [P2, P4]
CPU 2 rq: [P6, P7, P8]
CPU 3 rq: [P9]           ← underloaded
```

Load balancing: A periodic load balancer checks queue lengths and migrates processes across CPUs.

Cache affinity: CFS tries to keep a process on the same CPU (warm cache), migrating only when the load imbalance is significant.

CFS: `vruntime` and New/Waking Processes

New processes: Initialized with `vruntime = min_vruntime` of the current queue. (Same idea as stride's "initialize pass to min-pass.")

Sleeping processes: While sleeping, `vruntime` does **not** advance. When they wake, they could have a very small (old) `vruntime` and monopolize the CPU.

Fix: On wakeup, set `vruntime = max(current vruntime, min_vruntime - sched_latency)`.

This ensures a sleeping process doesn't get a large unfair CPU "credit" for its sleep time — but it does get to run promptly.

✅ Checkpoint 6 — Linux CFS

Test your understanding:

1. What data structure does CFS use to store runnable processes? What is its time complexity for picking the next task?
2. Process A has nice = 0 (weight 1024), Process B has nice = 5 (weight 335). If both run for 10 ms of real time, how much does each process's `vruntime` advance?
3. What is `sched_latency`? What happens when there are too many runnable processes?
4. Explain the `vruntime` initialization for a newly woken sleeping process. Why is a correction needed?
5. How does CFS group scheduling relate to container CPU isolation in Docker/Kubernetes?

§ 9 — Worked Examples & Exam Problems

Worked Example 1: MLFQ Demotion Trace

Setup: 3 queues: Q0 (allotment = 3 ms, slice = 1 ms), Q1 (allotment = 6 ms, slice = 2 ms), Q2 (FIFO).

- Job **X**: 15 ms CPU, no I/O. Arrives at t = 0.
- Job **Y**: 4 ms CPU, no I/O. Arrives at t = 0.

Trace X's and Y's progress (assuming both enters Q0 and X goes first):

```
t=0–1:   X runs Q0 (1 out of 3 ms allotment; 14 ms left)
t=1–2:   Y runs Q0 (1 out of 3 ms allotment; 3 ms left)
t=2–3:   X runs Q0 (2 out of 3 ms allotment; 13 ms left)
t=3–4:   Y runs Q0 (2 out of 3 ms allotment; 2 ms left)
t=4–5:   X runs Q0 (3 out of 3 ms allotment; 12 ms left) → demoted to Q1
t=5–6:   Y runs Q0 (3 out of 3 ms allotment; 1 ms left) → demoted to Q1
t=6–8:   X runs Q1 (2 out of 6 ms allotment; 10 ms left)
t=8–9:   Y runs Q1 (1 out of 6 ms allotment; 0 ms left) → finishes
t=9–11:  X runs Q1 (4 out of 6 ms allotment; 8 ms left)
t=11–13: X runs Q1 (6 out of 6 ms allotment; 6 ms left) → demoted to Q2
t=13–19: X runs Q2 for the remaining 6 ms → finishes
```

Worked Example 2: MLFQ + I/O Interaction

Setup: Q0 (allotment = slice = 5 ms), Q1 (FIFO). Job **M**: 20 ms total CPU, does I/O for 10 ms after every 3 ms of CPU. Job **N**: 12 ms CPU, no I/O.

- M: each 3 ms CPU burst < 5 ms allotment
- N: uses 5 ms → demoted to Q1 immediately.

```
t=0–3:    M runs Q0 (3 out of 5 ms allotment; 17 ms left)
          M begins I/O (returns at t=13)
t=3–8:    N runs Q0 (5 out of 5 ms allotment; 7 ms left) →    demoted to Q1
t=8–13:   N runs Q1 (5 ms until t=13 when M gets back from I/O and preempts it; 2 ms left)
t=13–15:  M runs 2 ms (5 out of 5 ms allotment; 15 ms left) →    demoted to Q1
          M has 1 ms left before its next I/O
t=15–17:  N runs Q1 (0 ms left) → finishes
t=17–18:  Q1: M runs 1 ms (14 ms left)
          M begins I/O (returns at t=28)
...
```

M (interactive) preempts N (batch) when it returns from I/O.

Worked Example 3: Lottery Draw Simulation

Workload (#tickets): A=40; B=30; C=30; Total = 100. Ticket ranges: A → [1, 40], B → [41, 70], C → [71, 100].
Random draw sequence: **72, 15, 44, 91, 8, 57, 33, 76, 21, 65**.

Draw	Number	Range	Winner
1	72	71–100	C
2	15	1–40	A
3	44	41–70	B
4	91	71–100	C
5	8	1–40	A
6	57	41–70	B
7	33	1–40	A
8	76	71–100	C
9	21	1–40	A
10	65	41–70	B

A: 4 wins (40%), B: 3 wins (30%), C: 3 wins (30%).

Worked Example 4: Stride Scheduling Trace

Setup: W = 1,000. A: 500 tickets → stride = 2. B: 250 tickets → stride = 4. C: 125 tickets → stride = 8.

Initial: A.pass = 0, B.pass = 0, C.pass = 0.

Tick	Winner	A pass	B pass	C pass
1	A (tie → pick A)	**2**	0	0
2	B (tie B/C → pick B)	2	**4**	0
3	C	2	4	**8**
4	A	**4**	4	8
5	B (tie A/B → pick A actually; A=4, B=4)	**6**	4	8

(Tie-break: alphabetical order.)

After 8 ticks: A ≈ 4×, B ≈ 2×, C ≈ 1× — consistent with 500:250:125 = 4:2:1 ratio.

Worked Example 5: CFS `vruntime` Calculation

Setup: A (nice = 0, weight = 1024), B (nice = 5, weight = 335). Both runnable. $W_0 = 1024$.

Each runs for $\Delta t = 4$ ms real time:

$$\Delta \texttt{vruntime}_A = 4 \times \frac{1024}{1024} = \mathbf{4}\ \text{ms}$$

$$\Delta \texttt{vruntime}_B = 4 \times \frac{1024}{335} \approx \mathbf{12.2}\ \text{ms}$$

B's `vruntime` advances **3× faster** than A's. So A appears "further behind" relative to B after running the same real time → A gets scheduled more often. Over time: A gets ≈ 75% CPU, B gets ≈ 25% CPU.

Worked Example 6: Unfairness Metric

Two processes: A has 90 tickets, B has 10 tickets. Each needs exactly 2 quanta.

Under lottery (small sample): suppose A wins 2 draws immediately (A done at t = 2), then B wins 2 draws (B done at t = 4).

$$U = \frac{T_{\min}}{T_{\max}} = \frac{2}{4} = 0.5$$

Under stride (A.stride = 111, B.stride = 1000; W = 10,000):

- A and B interleaved in strict proportion — A finishes close to when B finishes.
- U approaches 1.0.

Stride is significantly **fairer** for short workloads.

Exam Strategy: MLFQ Problems

1. **Read all 5 rules carefully.** Note allotment per level and boost period S.
2. **Build a timeline table:** columns = time intervals; rows = jobs.
3. **Track allotment consumed per job separately** (cumulative, includes partial uses).
4. **Check for preemption on I/O return:** a job returning from I/O re-enters at its current queue level and may preempt lower-priority jobs.
5. **Apply priority boost** at every multiple of S — all jobs jump to Q0.
6. **Verify total CPU** at the end: sum of all job runtimes = total elapsed time (no idle if queues are non-empty).

Exam Strategy: Lottery & Stride Problems

Lottery:

1. Assign ticket ranges to each process (sorted, non-overlapping).
2. Map each random number to its ticket range owner.
3. Count wins per process; compare to target proportions $\text{tickets}_i / \sum$.
4. Note: few draws → high variance; state this explicitly.

Stride:

1. Compute strides: $\text{stride}_i = W / \text{tickets}_i$.
2. Initialize all passes to 0 (or min-pass for late arrivals).
3. At each tick: pick minimum-pass; run it; add stride to pass.
4. Tie-break: pick whichever process is listed first, or lowest-stride process.
5. Verify proportions after k ticks.

✅ Checkpoint 7 — Worked Examples

Self-assessment:

1. Can you trace an MLFQ timeline (including I/O, allotments, and boost) without assistance?
2. Can you map lottery draw numbers to winning processes given ticket ranges?
3. Can you perform a stride scheduling trace for 3 processes over 10 ticks?
4. Can you compute `vruntime` advancement for two CFS processes with different nice values?
5. Can you compute the unfairness metric U for any given trace?

If any answer is "no," re-read the corresponding section before the exam.

§ 10 — Summary & Key Takeaways

MLFQ: What It Achieves

MLFQ solves the fundamental scheduling dilemma **without** knowing job runtimes:

> *By observing behavior — jobs that use little CPU stay high; jobs that use their full allotment sink low — MLFQ approximates SJF/STCF for turnaround time while preserving RR-like response time for interactive jobs.*

Five rules. No oracles. Just feedback from past behavior.

Lottery & Stride: What They Achieve

Both answer a different question:

> *How do we guarantee each process or user a precise fraction of the CPU?*

- **Lottery:** Simple, elegant, probabilistic. Good for long-running processes with stable ticket counts.
- **Stride:** Deterministic, bounded error, better for short-lived processes needing precise guarantees.

Neither optimizes turnaround or response time — they optimize **fairness and proportionality**.

Linux CFS: What It Achieves

CFS unifies the insights of both worlds:

- **Proportional share** via `vruntime` weighting (like stride scheduling with nice-based weights).
- **Responsive to interactivity** via `vruntime` compensation for sleeping processes.
- **Scalable** via the O(log n) red-black tree.
- **Extensible** via group scheduling and bandwidth control for containers.

CFS is the production implementation that has served billions of Linux systems since 2007.

The Five MLFQ Rules — Final Summary

Rule	Description
1	Higher priority → runs first
2	Equal priority → Round Robin
3	New jobs → highest priority
4	Allotment exhausted (cumulative) → demoted one level
5	Every S time units → all jobs boosted to highest priority

Key Formulas

Lottery probability:

$$P(\text{process } i) = \frac{\text{tickets}_i}{\sum_j \text{tickets}_j}$$

Stride and pass:

$$\text{stride}_i = \frac{W}{\text{tickets}_i}, \qquad \text{pass}_i \mathrel{+}= \text{stride}_i \text{ (after each run)}$$

CFS `vruntime` advancement:

$$\Delta \texttt{vruntime}_i = \Delta t \cdot \frac{W_0}{w_i}$$

Unfairness metric:

$$U = \frac{C_{\min}}{C_{\max}} \in [0, 1], \quad U = 1 \Rightarrow \text{perfectly fair}$$

Policy Comparison Table

Goal	Best Policy
Low latency for interactive + high throughput for batch	MLFQ
Guaranteed CPU % for each process/user/VM	Lottery or Stride
Simple, low-overhead proportional share	Lottery
Precise, deterministic proportional share	Stride
Production Linux workloads	CFS (= stride + group scheduling)
Combine interactive + proportional goals	MLFQ layers + CFS within each layer

Recommended Exercises

Before Lecture 2.4, practice:

1. **MLFQ trace:** Jobs A (20 ms batch), B (3 ms interactive: 1 ms CPU / 8 ms I/O), C (8 ms batch). 3-level MLFQ, allotments 2 ms / 8 ms / ∞, boost S = 20 ms. Draw the full timeline to t = 60 ms.
2. **Lottery:** 5 processes with tickets [50, 30, 70, 10, 40]. Simulate 20 random draws and compute actual vs. target CPU shares.
3. **Stride:** A = 500 tickets, B = 250, C = 125. W = 1,000. Trace 12 ticks and verify proportional allocation.
4. **CFS:** A (nice = –5, weight = 3121), B (nice = 0, weight = 1024). Each runs for 6 ms real time. Compute `vruntime` advancement for both. Who gets scheduled next?

Key Terminology Review (1/2)

Term	Definition
MLFQ	Multi-Level Feedback Queue — adaptive priority scheduler
Time allotment	Total CPU time permitted at one priority level before demotion
Priority Boost	Resetting all jobs to Q0 after period S
Lottery Scheduling	Probabilistic proportional-share via random ticket draws
Stride Scheduling	Deterministic proportional-share via virtual-time pass counters
Pass counter	Virtual time counter per process in stride scheduling
Ticket currency	Multi-user budget system for ticket allocation
Ticket transfer	Lending CPU entitlement from one process to another

Key Terminology Review (2/2)

Term	Definition
CFS	Completely Fair Scheduler — Linux's production scheduler since 2007
`vruntime`	CFS virtual runtime; equivalent to stride's pass counter
`sched_latency`	CFS target: every process runs at least once in this window
Red-black tree	Self-balancing BST; CFS stores runnable processes keyed by `vruntime`
cgroup	Linux mechanism for grouping processes; enables CFS group scheduling
Unfairness metric U	Ratio of shorter to longer completion time; $U = 1$ = perfectly fair

📋 Lecture 2.3 Complete

Recommended Reading

- OSTEP: Chapter 8 (MLFQ) and Chapter 9 (Lottery/Stride/Linux CFS)

Contact & Q&A

- **Lecturer:** Gabriele Tolomei
- **Email:** tolomei@di.uniroma1.it

- **Office Hours:** Drop me a message, and we'll arrange a time to meet, either in person or online! 🙂

Thank you for your attention!

Appendix A: MLFQ Implementation Sketch

```c
#define NUM_QUEUES  4
#define ALLOTMENT   {2, 8, 32, INT_MAX}   // ms per level
#define QUANTUM     {1, 2, 8,  16}        // ms time slice per level

struct job {
    int pid;
    int priority;        // current queue level (0 = highest)
    int allotment_used;  // ms accumulated at current level
    int cpu_remaining;   // ms of CPU still needed
};

void mlfq_tick(struct job *j, int elapsed_ms) {
    j->allotment_used += elapsed_ms;
    j->cpu_remaining  -= elapsed_ms;
    if (j->allotment_used >= ALLOTMENT[j->priority]) {
        if (j->priority < NUM_QUEUES - 1) j->priority++;
        j->allotment_used = 0;
    }
}

void mlfq_boost(struct job jobs[], int n) {
    for (int i = 0; i < n; i++) {
        jobs[i].priority      = 0;
        jobs[i].allotment_used = 0;
    }
}
```

Appendix B: Lottery Implementation Sketch

```
int total_tickets = compute_total(run_queue);
int winner = rand() % total_tickets;   // uniform draw

int current = 0;
struct process *p;
list_for_each(p, run_queue) {
    current += p->tickets;
    if (current > winner) {
        run(p);
        return;
    }
}
```

Note: `rand() % N` has modulo bias when N is not a power of 2. Production code uses rejection sampling or a full 64-bit RNG.

Optimization: Sort run queue by descending ticket count. Winners (common processes) are found early on average → fewer iterations.

Appendix C: Stride Implementation Sketch

```
struct process *pick_min_pass(list_t *procs) {
    struct process *best = NULL;
    list_for_each(p, procs) {
        if (!best || p->pass < best->pass)
            best = p;
    }
    return best;
}

void stride_run_one(list_t *procs) {
    struct process *p = pick_min_pass(procs);
    execute_one_quantum(p);
    p->pass += p->stride;
}

void stride_admit(struct process *new_p, list_t *procs) {
    new_p->stride = LARGE_W / new_p->tickets;
    new_p->pass   = min_pass(procs);   // avoid CPU monopolization
    list_add(procs, new_p);
}
```

Can be optimized to O(log n) with a min-heap or red-black tree.

Appendix D: OSTEP Chapter References

Topic	OSTEP Reference
MLFQ structure and five rules	Ch. 8, §8.1–8.3
Gaming the scheduler	Ch. 8, §8.4
MLFQ tuning and real systems	Ch. 8, §8.5
Lottery scheduling	Ch. 9, §9.1–9.4
Stride scheduling	Ch. 9, §9.5
Linux CFS overview	Ch. 9, §9.7
O(1) scheduler background	Ch. 9 sidebar
Proportional share summary	Ch. 9, §9.8

Appendix E: Practice Problem Bank

MLFQ:

1. 2 queues (allotments 4 ms, ∞). Job A: 20 ms CPU. Job B: 3 ms CPU / 5 ms I/O repeating. Both arrive t = 0. Draw timeline to t = 40 ms.
2. 3 queues, boost S = 15 ms. Batch job (40 ms CPU). Count how many boosts occur before it completes, assuming interactive jobs consume 70% of each S-period.

Lottery:

3. A=10 tickets, B=10, C=80. What is the expected number of draws before A wins its first quantum? (Geometric distribution: $E[X] = 1/p$.)
4. A new process D (20 tickets) joins. Recalculate all shares. Which existing process lost the most proportional CPU?

Stride:

5. A changes tickets from 100 to 50 at tick 20. Stride changes immediately; pass does not. Explain why this is correct.

CFS:

6. A (nice = 0, w = 1024), B (nice = 10, w = 110). `sched_latency` = 6 ms. How many ms does each get in one scheduling period?