Chapter 5: CPU Scheduling

Operating Systems

CPU–I/O burst cycle | Dispatcher and context switch | Scheduling criteria

FCFS and the convoy effect | SJF, SRTF, exponential average | Priority and aging

Round robin and the time quantum | MLQ and MLFQ | Lottery, stride, fair share

Multiprocessor scheduling | Real time: RM and EDF | Linux EEVDF, Windows

Sercan KÜLCÜ · Giresun University · Computer Engineering

© Sercan Külcü. All rights reserved. · Interactive HTML version of these slides: https://sercankulcu.github.io/files/os/slides/Chapter_05_CPU_Scheduling.html

Learning Outcomes

Concepts

Explain the CPU–I/O burst cycle, scheduling decision points, preemptive and nonpreemptive scheduling, and the dispatcher's job.

Criteria

Compute waiting, turnaround and response times from a Gantt chart; weigh throughput, CPU utilization and fairness goals.

Algorithms

Apply FCFS, SJF, SRTF, priority, RR, MLQ and MLFQ step by step to the same set of processes and compare them.

Prediction and fairness

Predict the next CPU burst with an exponential average; prevent starvation with aging, lottery, stride and fair-share scheduling.

Multiprocessor and real time

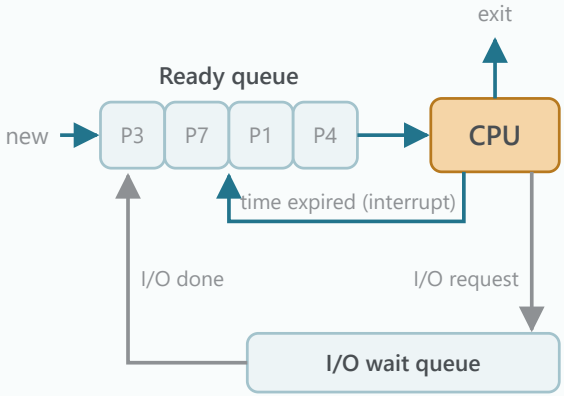
Explain load balancing and processor affinity; test schedulability with RM and EDF using utilization bounds.

Real systems

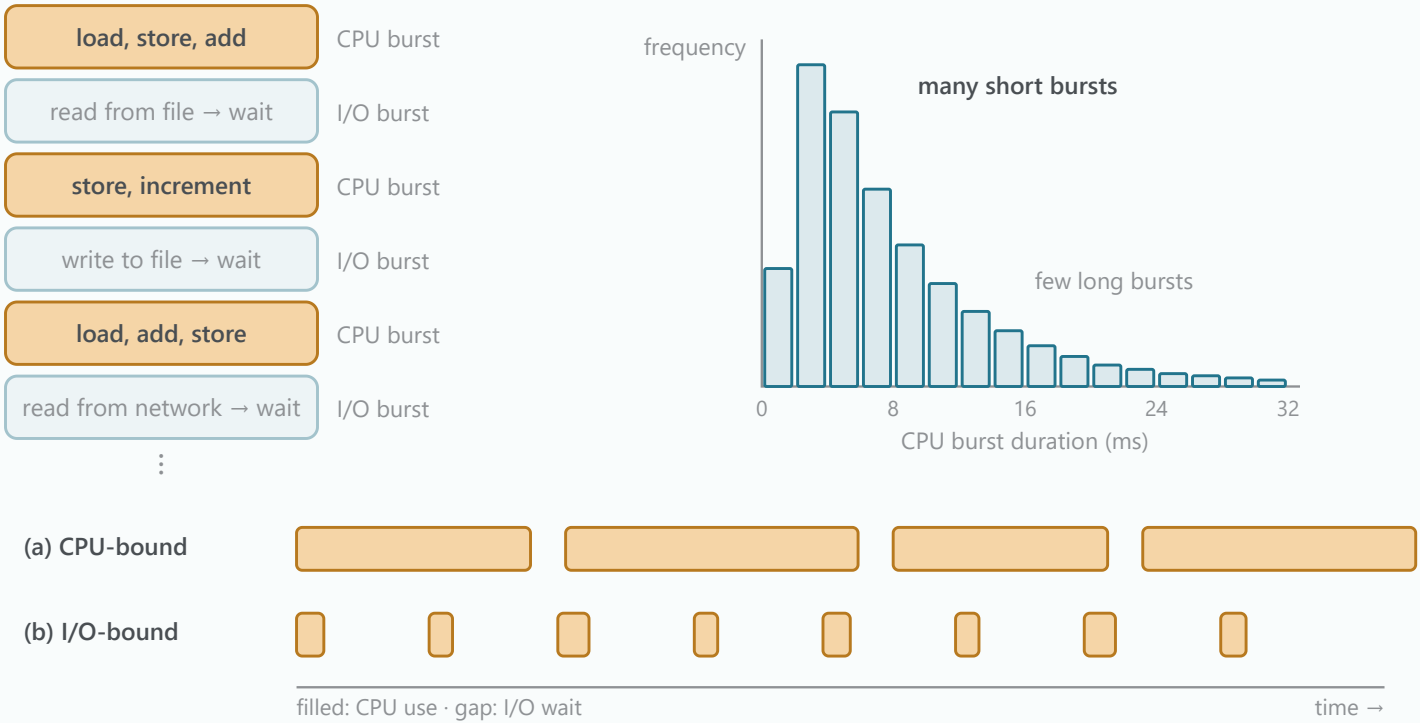
Describe Linux CFS and its successor EEVDF, Windows priority classes and dynamic priority boosts.

What Is Scheduling?

- In a multiprogrammed system many processes wait ready in memory; when the CPU becomes free, the **short-term scheduler** (CPU scheduler) decides which one runs.
- Goal: never leave the CPU idle, and improve the chosen objectives (response, throughput, fairness, deadlines).
- **Ready queue:** processes ready to run. **Wait queues:** those waiting for I/O or an event. **Running:** at most one per core.
- The ready queue need not be FIFO: it can be a priority queue, a tree or an unordered list.
- Modern kernels actually schedule **kernel threads**; "process scheduling" is used in this sense.



The CPU–I/O Burst Cycle

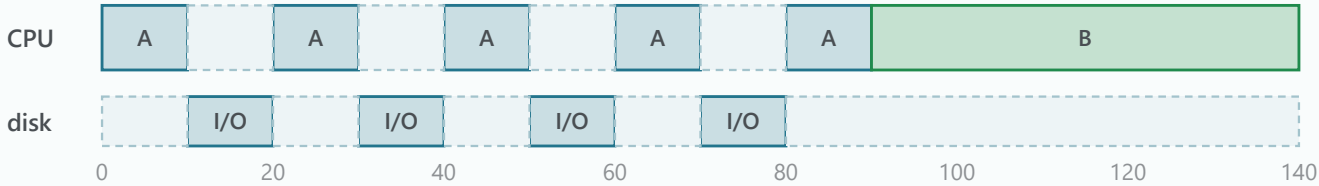


What matters is not how long the I/O wait is but **how short the CPU burst is**. An I/O-bound process that runs as soon as it is ready keeps the disk and network busy.

Overlapping with I/O: Treating Bursts as Separate Jobs

A and B each need 50 ms of CPU; A runs 10 ms and then waits 10 ms for the disk, B does no I/O.

No overlap: A's 50 ms of work counts as one job, and B starts only when A ends



A ends 90, B ends 140 → CPU busy 100 of 140 ms: 71%

If A counts as one 50 ms job, the CPU is **idle** while A is at the disk: B starts only when A ends.

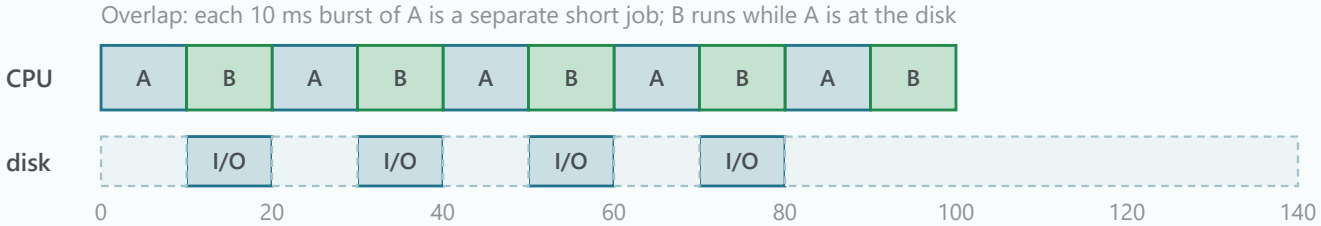
Step 1 / 2

Schedule	A done	B done	Avg. turnaround	CPU utilization
No overlap	90	140	(90 + 140) / 2 = 115 ms	100 / 140 ≈ 71%
Overlap	90	100	(90 + 100) / 2 = 95 ms	100 / 100 = 100%

A real system that does not know burst lengths learns this behavior with MLFQ.

Overlapping with I/O: Treating Bursts as Separate Jobs

A and B each need 50 ms of CPU; A runs 10 ms and then waits 10 ms for the disk, B does no I/O.



A ends 90, B ends 100 → CPU busy for all 100 ms: 100%

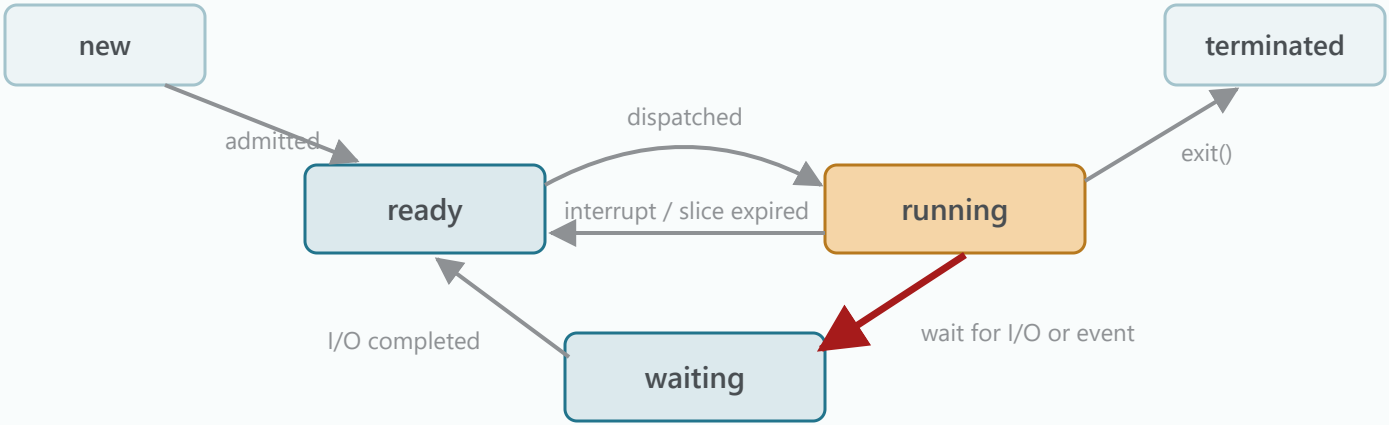
Each burst is a separate job: B runs while A is at the disk; the short returning A preempts B (STCF).

Step 2 / 2

Schedule	A done	B done	Avg. turnaround	CPU utilization
No overlap	90	140	(90 + 140) / 2 = 115 ms	100 / 140 ≈ 71%
Overlap	90	100	(90 + 100) / 2 = 95 ms	100 / 100 = 100%

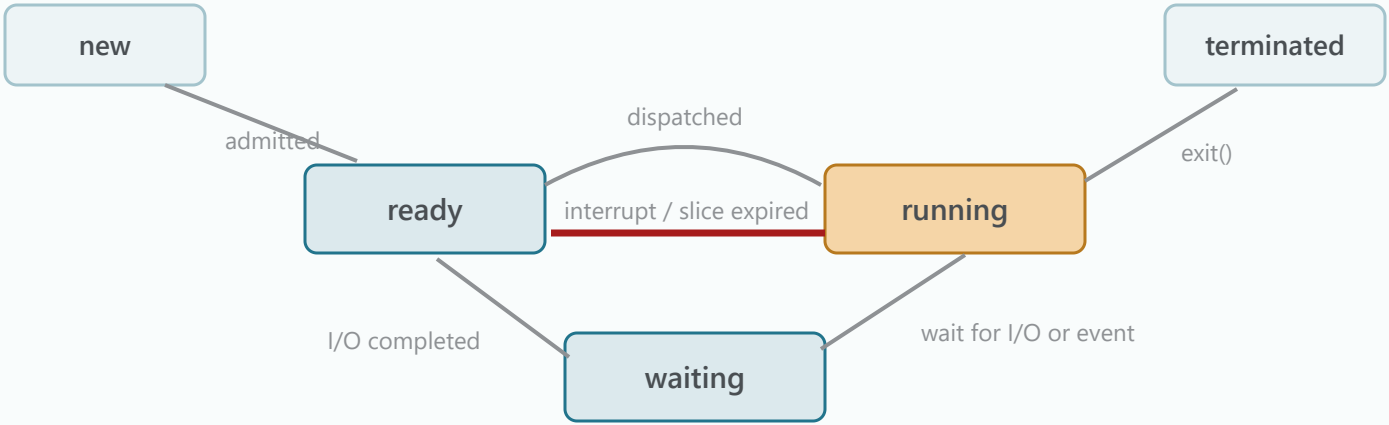
A real system that does not know burst lengths learns this behavior with MLFQ.

When Are Scheduling Decisions Made?



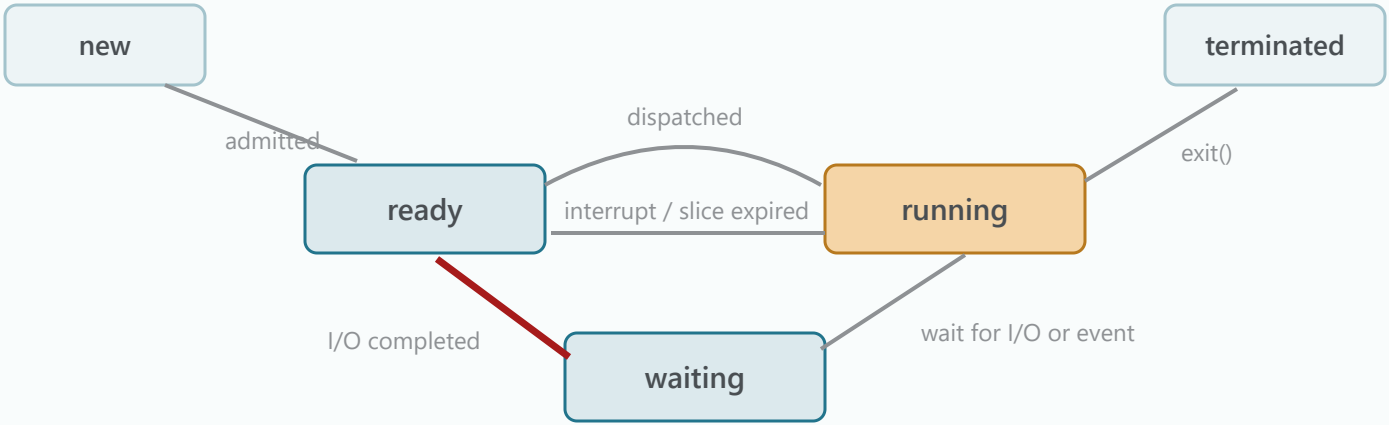
1. Running → waiting: the process requests I/O or calls `wait()`. The CPU is free; a choice is **required**.

When Are Scheduling Decisions Made?



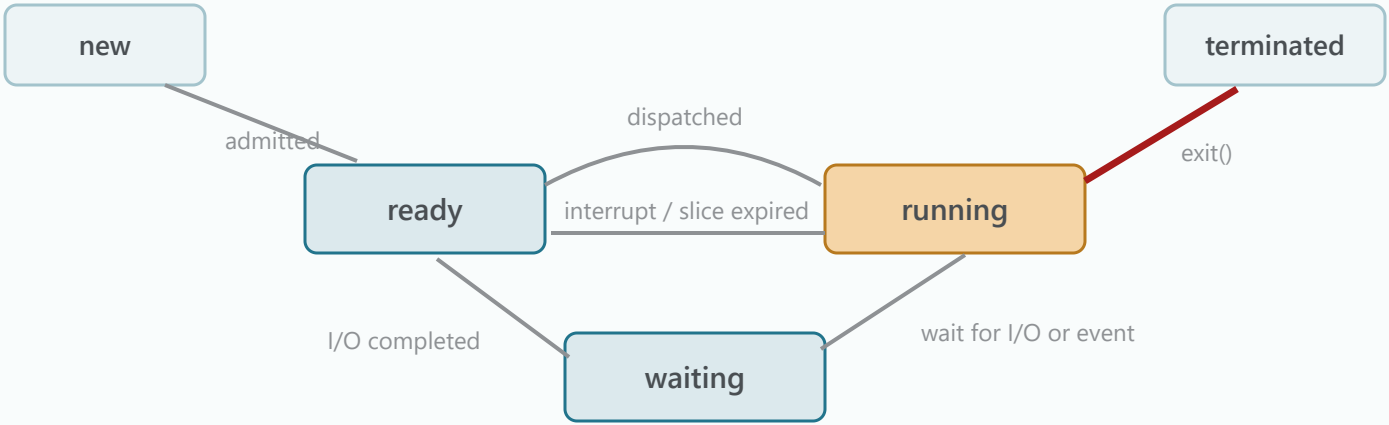
2. Running → ready: a clock interrupt arrived, the time slice expired. Preempting the running process is **optional**.

When Are Scheduling Decisions Made?



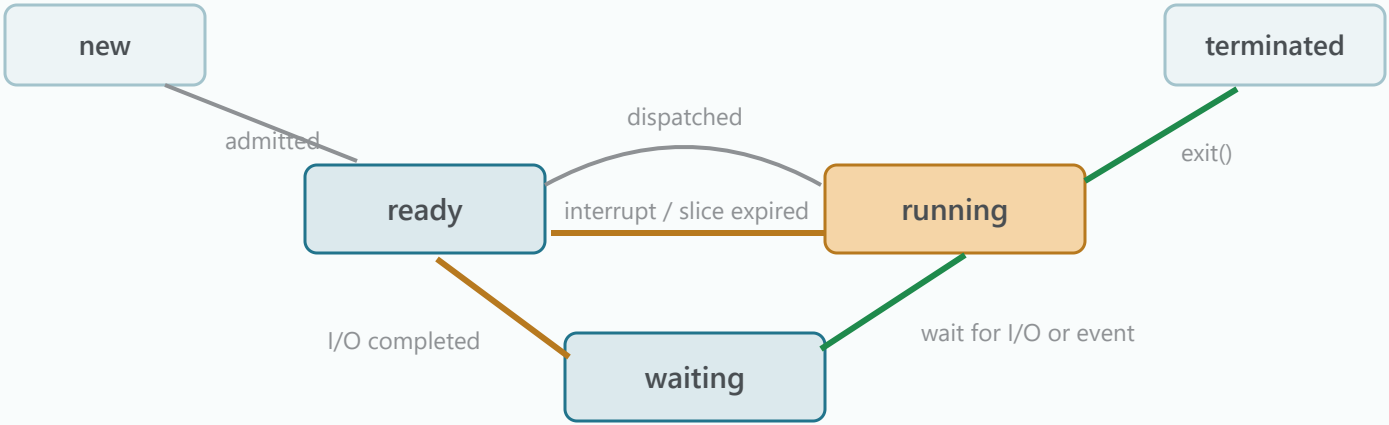
3. Waiting → ready: I/O completed; the woken process may preempt the running one if it is more important.

When Are Scheduling Decisions Made?



4. Termination: the process ends with `exit()`; the CPU is free, a choice is **required**.

When Are Scheduling Decisions Made?



A scheduler that decides only at 1 and 4 is **nonpreemptive** (cooperative); one that also decides at 2 and 3 is **preemptive**.

Preemptive and Nonpreemptive Scheduling

Nonpreemptive

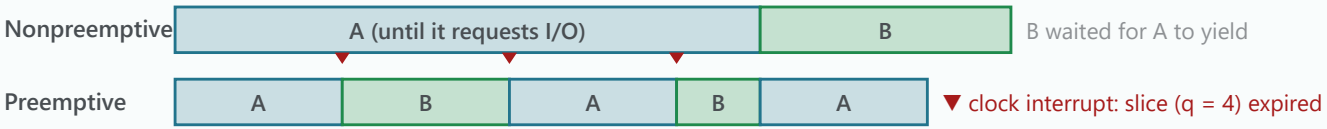
The chosen process runs until it blocks, exits or yields the CPU (`yield`). No clock interrupt needed; few switches.

Windows 3.x and classic Mac OS (≤ 9) used cooperative multitasking; one faulty program could hang the system.

Preemptive

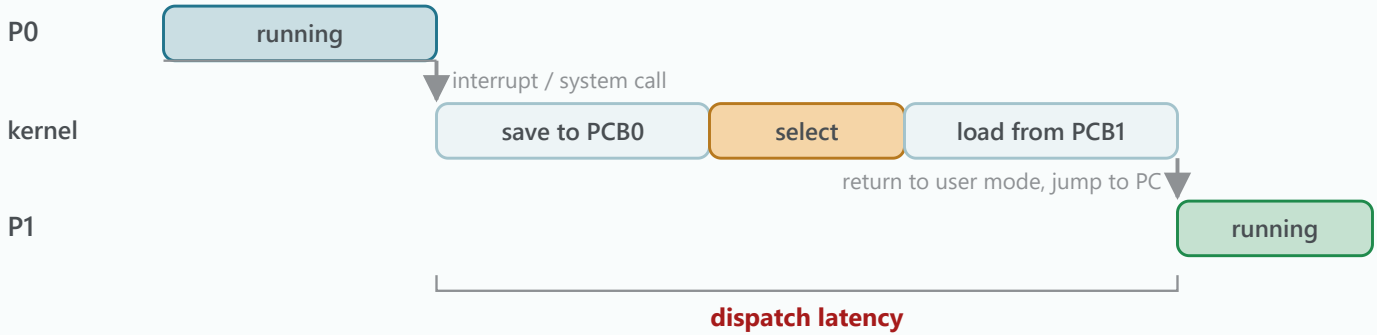
A process is suspended when its time slice expires or a higher-priority process becomes ready. Needs a clock interrupt; every modern general-purpose OS does this.

The price: **race conditions** on shared data and more context switches.



- **Kernel preemption:** since Linux 2.6 kernel code is preemptible too, but only when it holds no lock (`preempt_count = 0`).
- After 20 years, the **PREEMPT_RT** patches entered mainline in Linux 6.12 (2024): interrupt handlers run as threads; latency drops to microseconds.
- Batch jobs can use nonpreemptive or long-slice algorithms; interactive and server systems need preemption.

The Dispatcher and the Context Switch



- The **dispatcher** gives the CPU to the process the scheduler chose: it switches context, switches to user mode and jumps to the right place in the program.
- It runs on every switch, so it must be **fast**; the direct cost today is a few microseconds.
- Measuring on Linux: the `cs` column of `vmstat 1`; `voluntary_ctxt_switches` and `nonvoluntary_ctxt_switches` in `/proc/<pid>/status`.

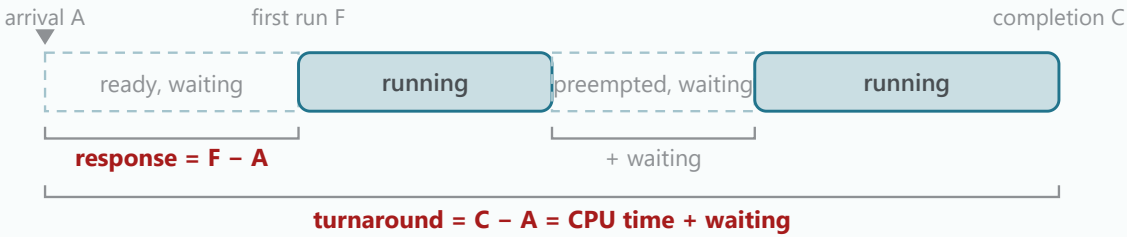
Component	On a process switch
Registers, PC, SP	saved / loaded
Memory map	page-table base changes (x86: CR3)
TLB	not flushed if PCID/ASID exist
Caches	not flushed, but left **cold**

Scheduling Goals by Environment

Environment	Main goals	Suitable approach
All systems	**Fairness** (similar processes get similar shares), **policy** enforcement (e.g. security processes first), **balance** (CPU and I/O devices busy together)	—
Batch	**Throughput** (jobs/hour), short **turnaround time**, high **CPU utilization**	Nonpreemptive or long-slice preemptive
Interactive	Short **response time**, **proportionality** (delays that match user expectations)	Preemptive; short time slice
Real-time	Meeting **deadlines** (no data loss), **predictability** (no drop in audio/video quality)	Priority / deadline based; tasks run briefly and block

Batch jobs are still common today: payroll and overnight banking runs, scientific computing and AI training (HPC queue managers such as Slurm), CI builds. Goals often conflict: frequent switches that shorten response time lower throughput.

Scheduling Criteria



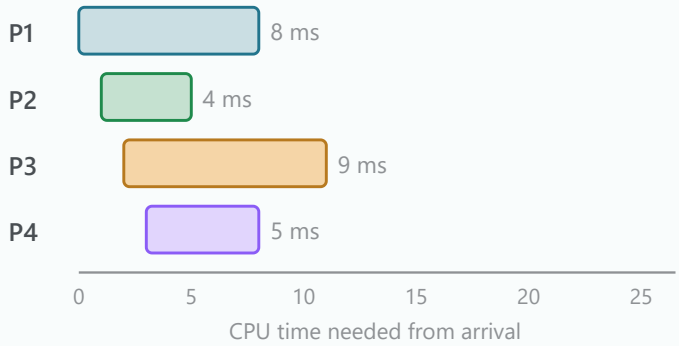
Criterion	Definition	Goal
CPU utilization	Fraction of time the CPU is busy (40–90% in real systems)	maximize
Throughput	Number of processes completed per unit time	maximize
Turnaround time	Time from submission to completion: C − A	minimize
Waiting time	Total time spent in the ready queue (without I/O: turnaround − CPU time)	minimize
Response time	Time from submission until first getting the CPU (the first response): F − A	minimize

The scheduler affects only the time spent waiting in the ready queue; CPU and I/O time belong to the program. In interactive systems reducing the **variance** of response time matters more than its average. Weighted turnaround = turnaround / CPU time.

A Common Set of Processes

Process	Arrival	CPU time	Priority
P1	0	8	2
P2	1	4	3
P3	2	9	1
P4	3	5	4

- Smaller number = **higher priority**; ties go to the earlier arrival.
- In RR a process that arrives at the same moment enters the queue **before** the one whose slice just ended.
- Times in ms; context-switch time taken as 0; no I/O.



FCFS, SJF, SRTF, priority and RR are run step by step on these four processes, so the results can be compared directly.

FCFS Step by Step

First-come, first-served: the ready queue is FIFO; a process that gets the CPU keeps it until it finishes or blocks. The simplest algorithm to implement.



t = 0

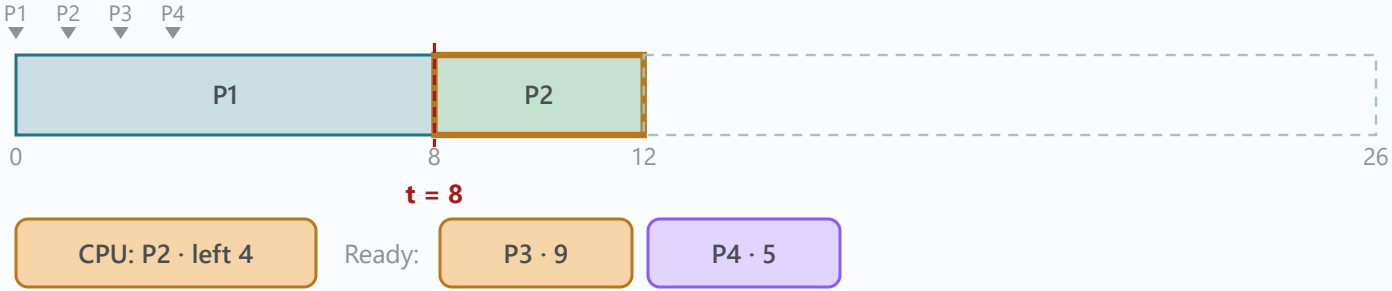
CPU: P1 · left 8

Ready: (empty)

t = 0: only **P1** is in the queue; it takes the CPU and keeps it for 8 ms (nonpreemptive).

FCFS Step by Step

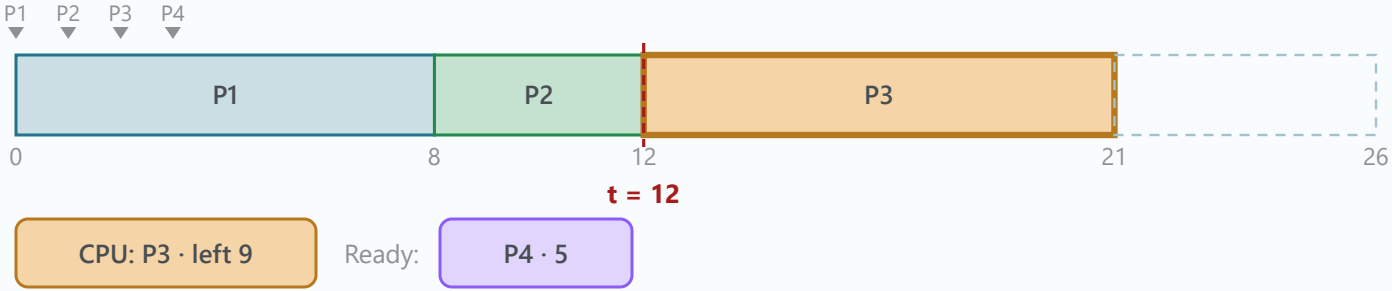
First-come, first-served: the ready queue is FIFO; a process that gets the CPU keeps it until it finishes or blocks. The simplest algorithm to implement.



t = 8: P1 is done. The queue advances in arrival order → **P2** at the head runs (it waited 7 ms).

FCFS Step by Step

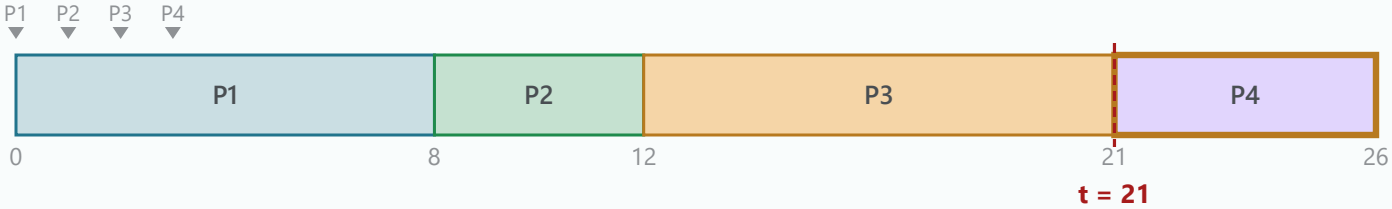
First-come, first-served: the ready queue is FIFO; a process that gets the CPU keeps it until it finishes or blocks. The simplest algorithm to implement.



t = 12: P2 is done. The queue advances in arrival order → **P3** at the head runs (it waited 10 ms).

FCFS Step by Step

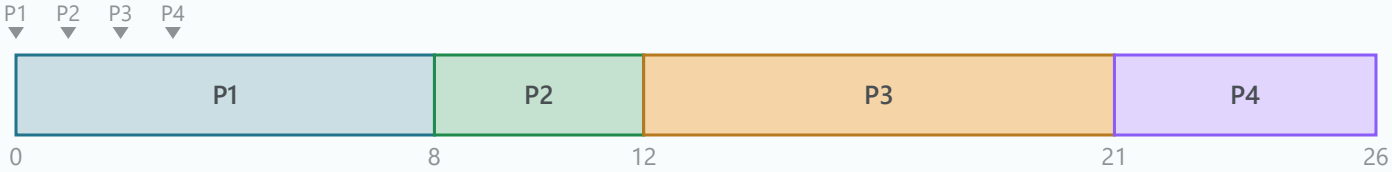
First-come, first-served: the ready queue is FIFO; a process that gets the CPU keeps it until it finishes or blocks. The simplest algorithm to implement.



t = 21: P3 is done. The queue advances in arrival order → **P4** at the head runs (it waited 18 ms).

FCFS Step by Step

First-come, first-served: the ready queue is FIFO; a process that gets the CPU keeps it until it finishes or blocks. The simplest algorithm to implement.



Process	Arrival	CPU	First run	Completion	Turnaround	Waiting	Response
P1	0	8	0	8	8	0	0
P2	1	4	8	12	11	7	7
P3	2	9	12	21	19	10	10
P4	3	5	21	26	23	18	18
Avg.					**15,25**	**8,75**	**8,75**

Average waiting **8.75**, turnaround **15.25**, response **8.75** ms · 3 context switches.

FCFS: The Convoy Effect

Order: P1 → P2 → P3 (all ready at t = 0)



Waiting: P1 = 0, P2 = 24, P3 = 27 → average (0 + 24 + 27) / 3 = 17 ms

Long job first: the short P2 and P3 wait behind P1. Average waiting **17 ms**.

Step 1 / 3

The **CPU-bound** process holds the CPU until it terminates; I/O-bound processes release it after short bursts. FCFS is nonpreemptive; there is no starvation, but it is unsuitable for interactive systems.

FCFS: The Convoy Effect

Order: P2 → P3 → P1



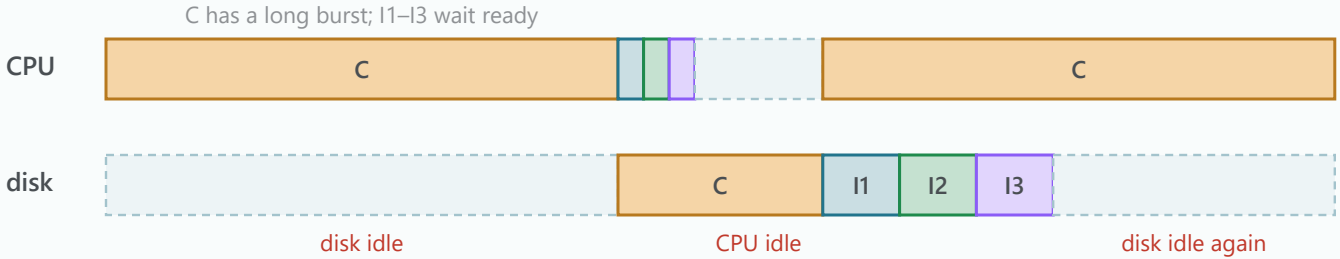
Waiting: P1 = 6, P2 = 0, P3 = 3 → average (6 + 0 + 3) / 3 = 3 ms

The same jobs in a different arrival order: average **3 ms**. Under FCFS the result is very sensitive to arrival order.

Step 2 / 3

The **CPU-bound** process holds the CPU until it terminates; I/O-bound processes release it after short bursts. FCFS is nonpreemptive; there is no starvation, but it is unsuitable for interactive systems.

FCFS: The Convoy Effect



C: CPU-bound · I1, I2, I3: I/O-bound (1 ms CPU, 3 ms disk)

Convoy effect: the CPU-bound C holds the CPU; the I/O-bound processes pile up behind it and the devices sit idle in turn.

Step 3 / 3

The **CPU-bound** process holds the CPU until it terminates; I/O-bound processes release it after short bursts. FCFS is nonpreemptive; there is no starvation, but it is unsuitable for interactive systems.

Computing FCFS Waiting Time

```
1  /* p[] sorted by arrival time */
2  void fcfs(Proc p[], int n) {
▶    int t = 0, sum = 0;
4    for (int i = 0; i < n; i++) {
5      if (t < p[i].arrival) t = p[i].arrival;
6      int wait = t - p[i].arrival;
7      t += p[i].burst;          /* completion */
8      sum += wait;
9      printf("%s: %d\n", p[i].name, wait);
10   }
11   printf("avg = %.2f\n", (double) sum / n);
12 }
```



The clock `t` and the total waiting `sum` start at zero. The processes are in arrival order.

Step 1 / 10

VARIABLES

`i = –` `t = 0` `sum = 0`

OUTPUT

Computing FCFS Waiting Time

```
1  /* p[] sorted by arrival time */
2  void fcfs(Proc p[], int n) {
3    int t = 0, sum = 0;
4    for (int i = 0; i < n; i++) {
▶      if (t < p[i].arrival) t = p[i].arrival;
▶      int wait = t - p[i].arrival;
7      t += p[i].burst;          /* completion */
8      sum += wait;
9      printf("%s: %d\n", p[i].name, wait);
10   }
11   printf("avg = %.2f\n", (double) sum / n);
12 }
```



P1 arrived at 0; the CPU was free at 0 → wait = 0 − 0 = **0**.

Step 2 / 10

VARIABLES

`i = 0` `t = 0` `arrival = 0` `wait = 0` `sum = 0`

OUTPUT

Computing FCFS Waiting Time

```
1  /* p[] sorted by arrival time */
2  void fcfs(Proc p[], int n) {
3    int t = 0, sum = 0;
4    for (int i = 0; i < n; i++) {
5      if (t < p[i].arrival) t = p[i].arrival;
6      int wait = t - p[i].arrival;
▶      t += p[i].burst;          /* completion */
▶      sum += wait;
▶      printf("%s: %d\n", p[i].name, wait);
10   }
11   printf("avg = %.2f\n", (double) sum / n);
12 }
```



P1 runs 8 ms; the clock becomes **8**; total waiting 0.

Step 3 / 10

VARIABLES

`i = 0` `t = 8` `arrival = 0` `wait = 0` `sum = 0`

OUTPUT

```
P1: 0
```

Computing FCFS Waiting Time

```
1  /* p[] sorted by arrival time */
2  void fcfs(Proc p[], int n) {
3    int t = 0, sum = 0;
4    for (int i = 0; i < n; i++) {
▶      if (t < p[i].arrival) t = p[i].arrival;
▶      int wait = t - p[i].arrival;
7      t += p[i].burst;          /* completion */
8      sum += wait;
9      printf("%s: %d\n", p[i].name, wait);
10   }
11   printf("avg = %.2f\n", (double) sum / n);
12 }
```



P2 arrived at 1; the CPU was free at 8 → wait = 8 − 1 = **7**.

Step 4 / 10

VARIABLES

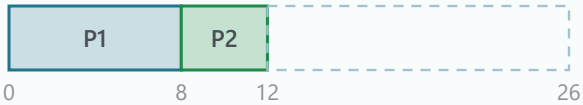
`i = 1` `t = 8` `arrival = 1` `wait = 7` `sum = 0`

OUTPUT

```
P1: 0
```

Computing FCFS Waiting Time

```
1  /* p[] sorted by arrival time */
2  void fcfs(Proc p[], int n) {
3    int t = 0, sum = 0;
4    for (int i = 0; i < n; i++) {
5      if (t < p[i].arrival) t = p[i].arrival;
6      int wait = t - p[i].arrival;
▶      t += p[i].burst;          /* completion */
▶      sum += wait;
▶      printf("%s: %d\n", p[i].name, wait);
10   }
11   printf("avg = %.2f\n", (double) sum / n);
12 }
```



P2 runs 4 ms; the clock becomes **12**; total waiting 7.

Step 5 / 10

VARIABLES

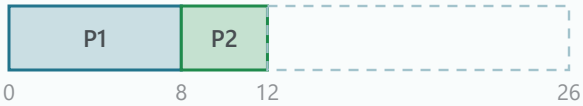
`i = 1` `t = 12` `arrival = 1` `wait = 7` `sum = 7`

OUTPUT

```
P1: 0
P2: 7
```

Computing FCFS Waiting Time

```
1  /* p[] sorted by arrival time */
2  void fcfs(Proc p[], int n) {
3    int t = 0, sum = 0;
4    for (int i = 0; i < n; i++) {
▶      if (t < p[i].arrival) t = p[i].arrival;
▶      int wait = t - p[i].arrival;
7      t += p[i].burst;          /* completion */
8      sum += wait;
9      printf("%s: %d\n", p[i].name, wait);
10   }
11   printf("avg = %.2f\n", (double) sum / n);
12 }
```



P3 arrived at 2; the CPU was free at 12 → wait = 12 − 2 = **10**.

Step 6 / 10

VARIABLES

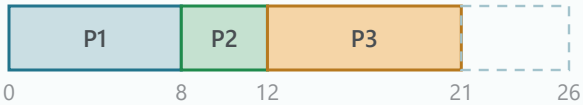
`i = 2` `t = 12` `arrival = 2` `wait = 10` `sum = 7`

OUTPUT

```
P1: 0
P2: 7
```

Computing FCFS Waiting Time

```
1  /* p[] sorted by arrival time */
2  void fcfs(Proc p[], int n) {
3    int t = 0, sum = 0;
4    for (int i = 0; i < n; i++) {
5      if (t < p[i].arrival) t = p[i].arrival;
6      int wait = t - p[i].arrival;
▶      t += p[i].burst;          /* completion */
▶      sum += wait;
▶      printf("%s: %d\n", p[i].name, wait);
10   }
11   printf("avg = %.2f\n", (double) sum / n);
12 }
```



P3 runs 9 ms; the clock becomes **21**; total waiting 17.

Step 7 / 10

VARIABLES

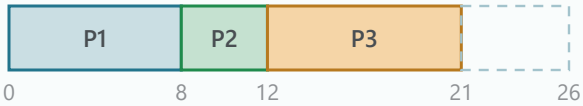
`i = 2` `t = 21` `arrival = 2` `wait = 10` `sum = 17`

OUTPUT

```
P1: 0
P2: 7
P3: 10
```

Computing FCFS Waiting Time

```
1  /* p[] sorted by arrival time */
2  void fcfs(Proc p[], int n) {
3    int t = 0, sum = 0;
4    for (int i = 0; i < n; i++) {
▶      if (t < p[i].arrival) t = p[i].arrival;
▶      int wait = t - p[i].arrival;
7      t += p[i].burst;          /* completion */
8      sum += wait;
9      printf("%s: %d\n", p[i].name, wait);
10   }
11   printf("avg = %.2f\n", (double) sum / n);
12 }
```



P4 arrived at 3; the CPU was free at 21 → wait = 21 − 3 = **18**.

Step 8 / 10

VARIABLES

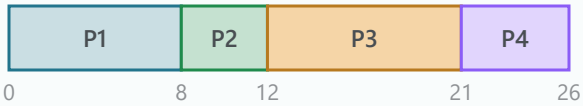
`i = 3` `t = 21` `arrival = 3` `wait = 18` `sum = 17`

OUTPUT

```
P1: 0
P2: 7
P3: 10
```

Computing FCFS Waiting Time

```
1  /* p[] sorted by arrival time */
2  void fcfs(Proc p[], int n) {
3    int t = 0, sum = 0;
4    for (int i = 0; i < n; i++) {
5      if (t < p[i].arrival) t = p[i].arrival;
6      int wait = t - p[i].arrival;
▶      t += p[i].burst;          /* completion */
▶      sum += wait;
▶      printf("%s: %d\n", p[i].name, wait);
10   }
11   printf("avg = %.2f\n", (double) sum / n);
12 }
```



P4 runs 5 ms; the clock becomes **26**; total waiting 35.

Step 9 / 10

VARIABLES

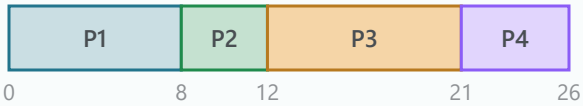
`i = 3` `t = 26` `arrival = 3` `wait = 18` `sum = 35`

OUTPUT

```
P1: 0
P2: 7
P3: 10
P4: 18
```

Computing FCFS Waiting Time

```
1  /* p[] sorted by arrival time */
2  void fcfs(Proc p[], int n) {
3    int t = 0, sum = 0;
4    for (int i = 0; i < n; i++) {
5      if (t < p[i].arrival) t = p[i].arrival;
6      int wait = t - p[i].arrival;
7      t += p[i].burst;          /* completion */
8      sum += wait;
9      printf("%s: %d\n", p[i].name, wait);
10   }
▶    printf("avg = %.2f\n", (double) sum / n);
12 }
```



Average waiting = 35 / 4 = **8.75 ms**, the same as the FCFS Gantt chart.

Step 10 / 10

VARIABLES

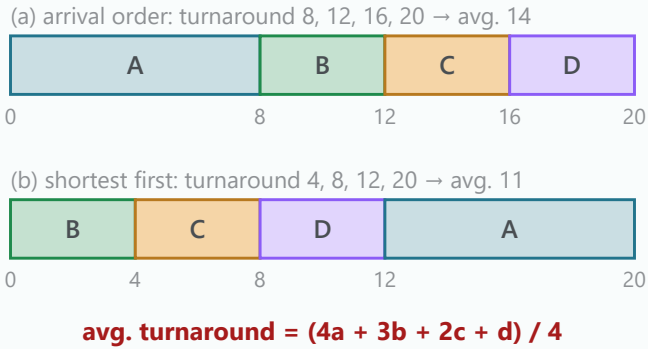
`i = 4` `t = 26` `sum = 35`

OUTPUT

```
P1: 0
P2: 7
P3: 10
P4: 18
avg = 8.75
```

SJF: Shortest Job First

- **Shortest-job-first**: among the ready processes, pick the one with the shortest **next CPU burst**; ties are broken FCFS.
- If all jobs are ready at once, it **provably minimizes the average waiting time**: the first job's length is added to everyone's wait, so the shortest should go first (in the formula on the right, a has the largest coefficient).
- If arrivals are at different times, nonpreemptive SJF is not optimal; its preemptive version, **SRTF**, is.
- The problem: the next burst length is **unknown**. Batch systems use the user's estimate (e.g. Slurm `--time`); short-term scheduling predicts it from history.
- Long jobs can **starve** if short jobs keep arriving.

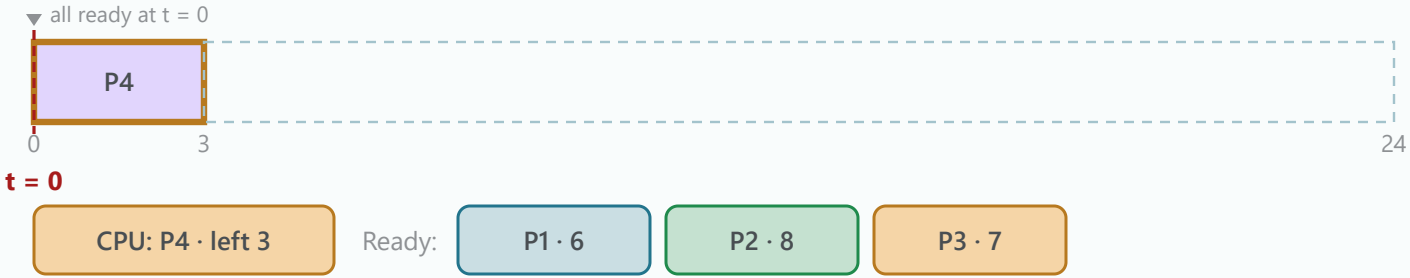


avg. turnaround = (4a + 3b + 2c + d) / 4

When all jobs are ready at once, SJF minimizes **average waiting and turnaround time**; it does not aim to raise CPU utilization, since the CPU is busy under every algorithm as long as there is work.

SJF: All Ready at Once

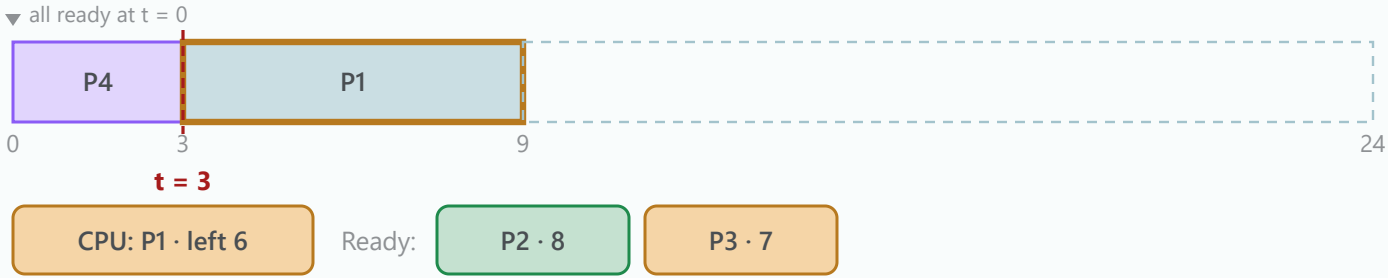
The classic example: four processes ready at **t = 0**; CPU times P1 = 6, P2 = 8, P3 = 7, P4 = 3 ms. Under FCFS the same set would have average waiting (0 + 6 + 14 + 21) / 4 = **10.25 ms**.



t = 0: all are ready; **P4**, with the shortest CPU time (3), is chosen.

SJF: All Ready at Once

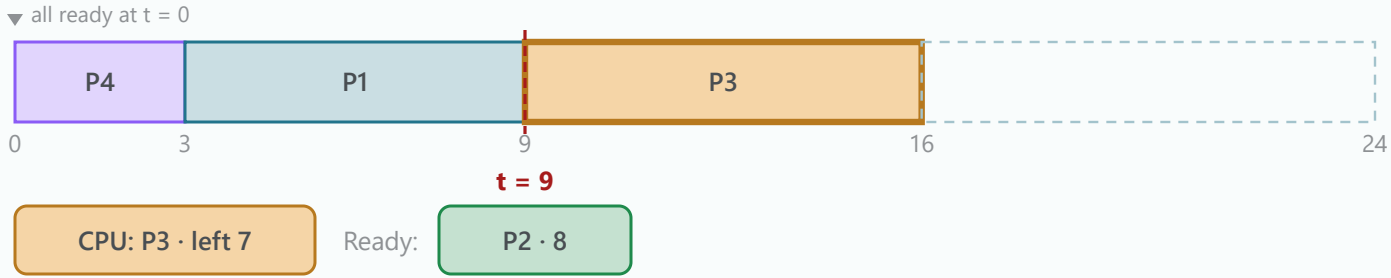
The classic example: four processes ready at **t = 0**; CPU times P1 = 6, P2 = 8, P3 = 7, P4 = 3 ms. Under FCFS the same set would have average waiting (0 + 6 + 14 + 21) / 4 = **10.25 ms**.



t = 3: P4 is done. Ready: P1 (6), P2 (8), P3 (7) → shortest **P1** (6).

SJF: All Ready at Once

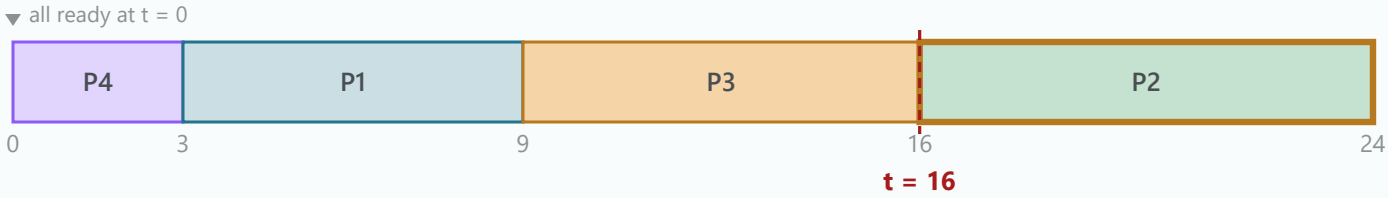
The classic example: four processes ready at **t = 0**; CPU times P1 = 6, P2 = 8, P3 = 7, P4 = 3 ms. Under FCFS the same set would have average waiting (0 + 6 + 14 + 21) / 4 = **10.25 ms**.



t = 9: P1 is done. Ready: P3 (7), P2 (8) → shortest **P3** (7).

SJF: All Ready at Once

The classic example: four processes ready at **t = 0**; CPU times P1 = 6, P2 = 8, P3 = 7, P4 = 3 ms. Under FCFS the same set would have average waiting (0 + 6 + 14 + 21) / 4 = **10.25 ms**.



CPU: P2 · left 8

Ready: (empty)

t = 16: P3 is done. Ready: P2 (8) → shortest **P2** (8).

SJF: All Ready at Once

The classic example: four processes ready at **t = 0**; CPU times P1 = 6, P2 = 8, P3 = 7, P4 = 3 ms. Under FCFS the same set would have average waiting (0 + 6 + 14 + 21) / 4 = **10.25 ms**.



Process	Arrival	CPU	First run	Completion	Turnaround	Waiting	Response
P1	0	6	3	9	9	3	3
P2	0	8	16	24	24	16	16
P3	0	7	9	16	16	9	9
P4	0	3	0	3	3	0	0
Avg.					**13**	**7**	**7**

Average waiting **7**, turnaround **13**, response **7** ms · 3 context switches.

SJF on the Common Set

Nonpreemptive SJF on the common set: at t = 0 only P1 exists and cannot be preempted; every later choice takes the **shortest** ready process.



t = 0

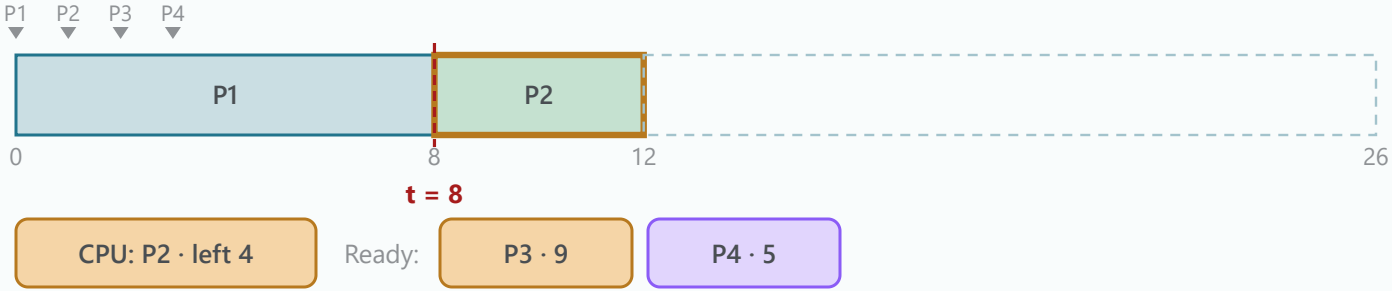
CPU: P1 · left 8

Ready: (empty)

t = 0: only **P1** is ready. SJF is nonpreemptive: short jobs arriving later cannot stop P1.

SJF on the Common Set

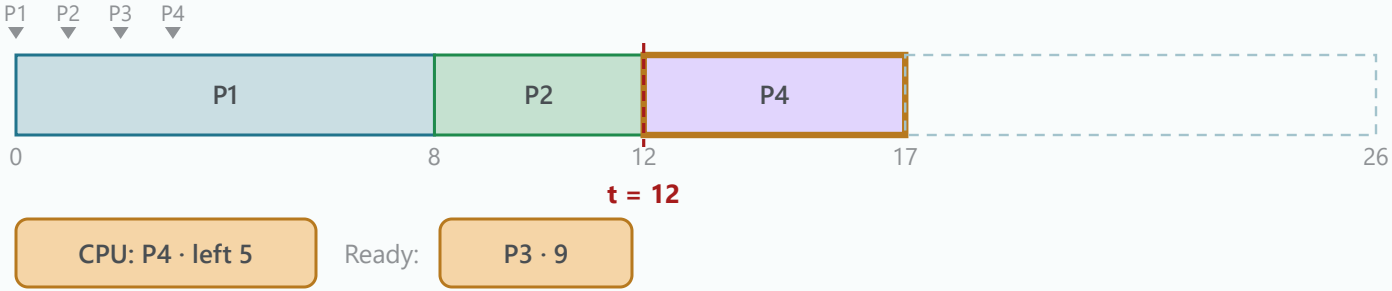
Nonpreemptive SJF on the common set: at t = 0 only P1 exists and cannot be preempted; every later choice takes the **shortest** ready process.



t = 8: P1 is done. Ready: P2 (4), P3 (9), P4 (5) → shortest **P2** (4).

SJF on the Common Set

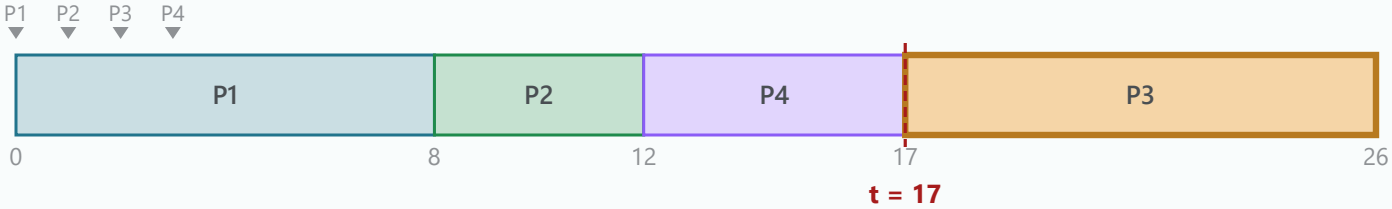
Nonpreemptive SJF on the common set: at t = 0 only P1 exists and cannot be preempted; every later choice takes the **shortest** ready process.



t = 12: P2 is done. Ready: P3 (9), P4 (5) → shortest **P4** (5).

SJF on the Common Set

Nonpreemptive SJF on the common set: at t = 0 only P1 exists and cannot be preempted; every later choice takes the **shortest** ready process.



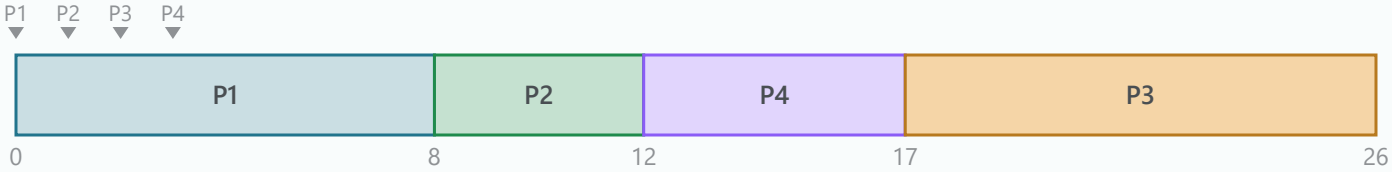
CPU: P3 · left 9

Ready: (empty)

t = 17: P4 is done. Ready: P3 (9) → shortest **P3** (9).

SJF on the Common Set

Nonpreemptive SJF on the common set: at t = 0 only P1 exists and cannot be preempted; every later choice takes the **shortest** ready process.



Process	Arrival	CPU	First run	Completion	Turnaround	Waiting	Response
P1	0	8	0	8	8	0	0
P2	1	4	8	12	11	7	7
P3	2	9	17	26	24	15	15
P4	3	5	12	17	14	9	9
Avg.					**14,25**	**7,75**	**7,75**

Average waiting **7.75**, turnaround **14.25**, response **7.75** ms · 3 context switches.

Comparing FCFS and SJF

		A	B	C	D	E	Avg.
	Arrival	0	1	2	3	4	
	CPU	4	3	5	2	4	
SJF	Completion	4	9	18	6	13	
	Turnaround	4	8	16	3	9	**8**
	Weighted	1	2,67	3,2	1,5	2,25	**2,12**
FCFS	Completion	4	7	12	14	18	
	Turnaround	4	6	10	11	14	**9**
	Weighted	1	2	2	5,5	3,5	**2,8**

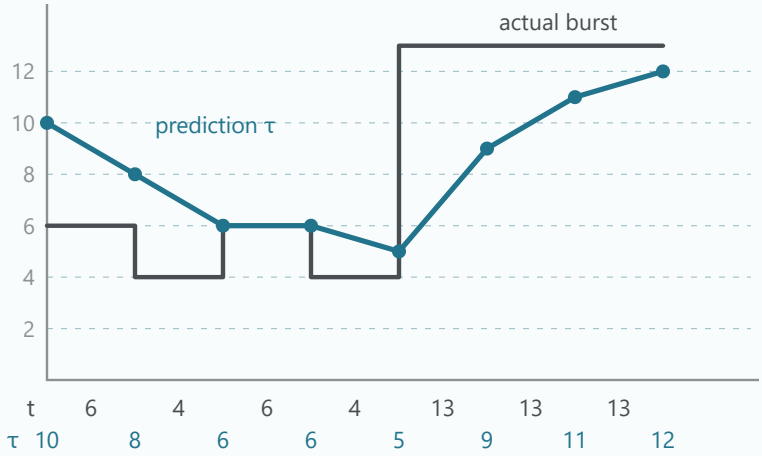
Weighted (normalized) turnaround = turnaround / CPU time; 1 is ideal. FCFS penalizes short jobs: in the set A(0, 1), B(1, 100), C(2, 1), D(3, 100), the 1 ms job C waits 99 ms behind B and its weighted turnaround is **100** (1.99 for D).

Weighted turnaround for C is 16 / 5 = **3.2**; the SJF average is 2.12.

Predicting the Next CPU Burst: Exponential Average

$$\tau_{n+1} = \alpha \cdot t_n + (1 - \alpha) \cdot \tau_n, \quad 0 \le \alpha \le 1$$

- t_n: actual length of the n-th burst; τ_n: the prediction made for it; τ_0: a constant (e.g. the system average).
- $\alpha = 0 \rightarrow$ the last measurement is ignored and the prediction never changes; $\alpha = 1 \rightarrow$ only the last burst counts. A common choice is $\alpha = ½$.
- Expanded: $\tau_{n+1} = \alpha t_n + (1-\alpha)\alpha t_{n-1} + \ldots + (1-\alpha)^{n+1}\tau_0$: older measurements weigh exponentially less.
- Tanenbaum's "aging" estimate is the same idea: T_0, $T_0/2 + T_1/2$, $T_0/4 + T_1/4 + T_2/2$, …
- Example ($\alpha = ½$, $\tau_0 = 10$): $t_0 = 6 \rightarrow \tau_1 = ½\cdot 6 + ½\cdot 10 = \mathbf{8}$.



Computing the Exponential Average

```
▶ double tau = 10.0;       /* τ0 */
▶ const double a = 0.5;   /* α */
▶ int t[] = { 6, 4, 6, 4, 13, 13, 13 };
4 for (int n = 0; n < 7; n++) {
5   printf("%.1f %d\n", tau, t[n]);
6   tau = a * t[n] + (1 - a) * tau;
7 }
8 printf("next: %.1f\n", tau);
```



Initial prediction $\tau_0 = 10$, $\alpha = 0.5$. No burst has been measured yet.

Step 1 / 9

VARIABLES

`tau = 10.0` `a = 0.5`

OUTPUT

Computing the Exponential Average

```
1 double tau = 10.0;       /* τ0 */
2 const double a = 0.5;  /* α */
3 int t[] = { 6, 4, 6, 4, 13, 13, 13 };
4 for (int n = 0; n < 7; n++) {
▶   printf("%.1f %d\n", tau, t[n]);
▶   tau = a * t[n] + (1 - a) * tau;
7 }
8 printf("next: %.1f\n", tau);
```



n = 0: actual 6, predicted 10 → τ = 0.5·6 + 0.5·10 = **8**.

Step 2 / 9

VARIABLES

`n = 0` `t[n] = 6` `tau = 8.0`

OUTPUT

```
10.0 6
```

Computing the Exponential Average

```
1 double tau = 10.0;       /* τ0 */
2 const double a = 0.5;  /* α */
3 int t[] = { 6, 4, 6, 4, 13, 13, 13 };
4 for (int n = 0; n < 7; n++) {
▶   printf("%.1f %d\n", tau, t[n]);
▶   tau = a * t[n] + (1 - a) * tau;
7 }
8 printf("next: %.1f\n", tau);
```



n = 1: actual 4, predicted 8 → τ = 0.5·4 + 0.5·8 = **6**.

Step 3 / 9

VARIABLES

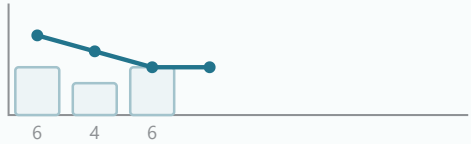
`n = 1` `t[n] = 4` `tau = 6.0`

OUTPUT

```
10.0 6
8.0 4
```

Computing the Exponential Average

```
1 double tau = 10.0;       /* τ0 */
2 const double a = 0.5;  /* α */
3 int t[] = { 6, 4, 6, 4, 13, 13, 13 };
4 for (int n = 0; n < 7; n++) {
▶   printf("%.1f %d\n", tau, t[n]);
▶   tau = a * t[n] + (1 - a) * tau;
7 }
8 printf("next: %.1f\n", tau);
```



n = 2: actual 6, predicted 6 → τ = 0.5·6 + 0.5·6 = **6**.

Step 4 / 9

VARIABLES

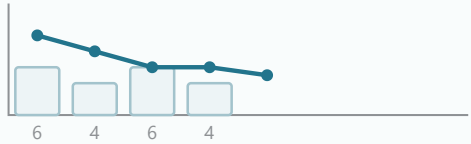
`n = 2` `t[n] = 6` `tau = 6.0`

OUTPUT

```
10.0 6
8.0 4
6.0 6
```

Computing the Exponential Average

```
1 double tau = 10.0;       /* τ0 */
2 const double a = 0.5;   /* α */
3 int t[] = { 6, 4, 6, 4, 13, 13, 13 };
4 for (int n = 0; n < 7; n++) {
▶   printf("%.1f %d\n", tau, t[n]);
▶   tau = a * t[n] + (1 - a) * tau;
7 }
8 printf("next: %.1f\n", tau);
```



n = 3: actual 4, predicted 6 → τ = 0.5·4 + 0.5·6 = **5**.

Step 5 / 9

VARIABLES

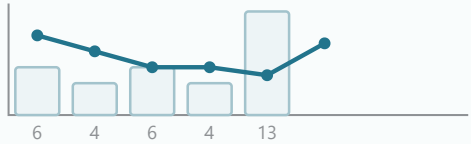
`n = 3` `t[n] = 4` `tau = 5.0`

OUTPUT

```
10.0 6
8.0 4
6.0 6
6.0 4
```

Computing the Exponential Average

```
1 double tau = 10.0;       /* τ0 */
2 const double a = 0.5;   /* α */
3 int t[] = { 6, 4, 6, 4, 13, 13, 13 };
4 for (int n = 0; n < 7; n++) {
▶   printf("%.1f %d\n", tau, t[n]);
▶   tau = a * t[n] + (1 - a) * tau;
7 }
8 printf("next: %.1f\n", tau);
```



n = 4: actual 13, predicted 5 → τ = 0.5·13 + 0.5·5 = **9**.

Step 6 / 9

VARIABLES

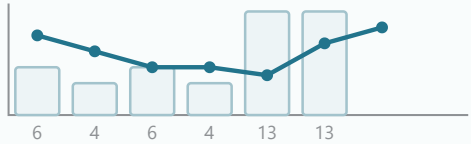
`n = 4` `t[n] = 13` `tau = 9.0`

OUTPUT

```
10.0 6
8.0 4
6.0 6
6.0 4
5.0 13
```

Computing the Exponential Average

```
1 double tau = 10.0;       /* τ0 */
2 const double a = 0.5;  /* α */
3 int t[] = { 6, 4, 6, 4, 13, 13, 13 };
4 for (int n = 0; n < 7; n++) {
▶   printf("%.1f %d\n", tau, t[n]);
▶   tau = a * t[n] + (1 - a) * tau;
7 }
8 printf("next: %.1f\n", tau);
```



n = 5: actual 13, predicted 9 → τ = 0.5·13 + 0.5·9 = **11**.

Step 7 / 9

VARIABLES

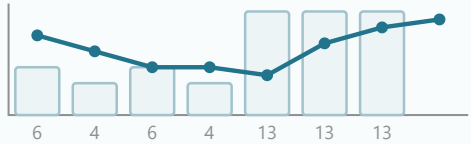
`n = 5` `t[n] = 13` `tau = 11.0`

OUTPUT

```
10.0 6
8.0 4
6.0 6
6.0 4
5.0 13
9.0 13
```

Computing the Exponential Average

```
1 double tau = 10.0;       /* τ0 */
2 const double a = 0.5;   /* α */
3 int t[] = { 6, 4, 6, 4, 13, 13, 13 };
4 for (int n = 0; n < 7; n++) {
▶   printf("%.1f %d\n", tau, t[n]);
▶   tau = a * t[n] + (1 - a) * tau;
7 }
8 printf("next: %.1f\n", tau);
```



n = 6: actual 13, predicted 11 → τ = 0.5·13 + 0.5·11 = **12**.

Step 8 / 9

VARIABLES

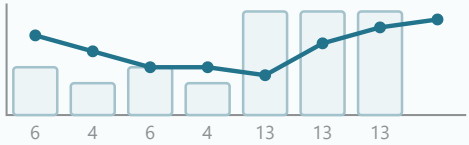
`n = 6` `t[n] = 13` `tau = 12.0`

OUTPUT

```
10.0 6
8.0 4
6.0 6
6.0 4
5.0 13
9.0 13
11.0 13
```

Computing the Exponential Average

```
1 double tau = 10.0;       /* τ0 */
2 const double a = 0.5;   /* α */
3 int t[] = { 6, 4, 6, 4, 13, 13, 13 };
4 for (int n = 0; n < 7; n++) {
5   printf("%.1f %d\n", tau, t[n]);
6   tau = a * t[n] + (1 - a) * tau;
7 }
▶ printf("next: %.1f\n", tau);
```



The bursts jumped to 13; the prediction catches up in a few steps: 5 → 9 → 11 → **12**.

Step 9 / 9

VARIABLES

`n = 7` `tau = 12.0`

OUTPUT

```
10.0 6
8.0 4
6.0 6
6.0 4
5.0 13
9.0 13
11.0 13
next: 12.0
```

SRTF Step by Step

Shortest-remaining-time-first (SRTF / SRTN): the preemptive version of SJF. When a new process arrives, its length is compared with the **remaining** time of the running one.



t = 0

CPU: P1 · left 8

Ready: (empty)

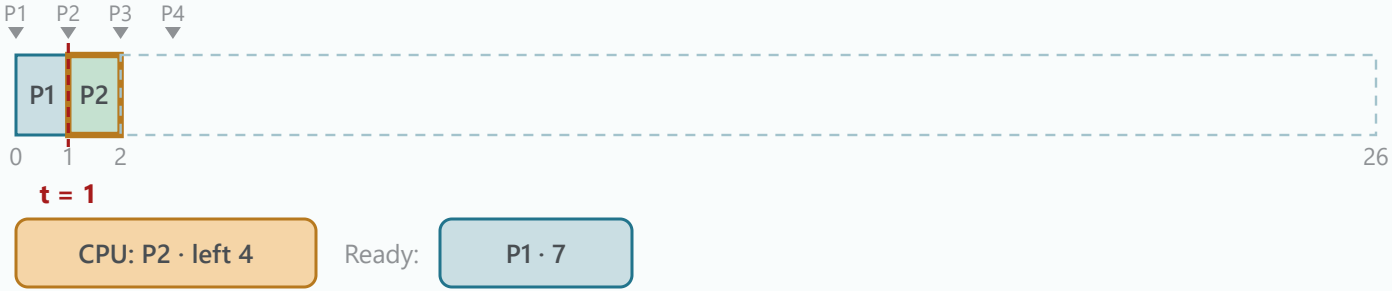
t = 0: only **P1** exists and starts running.

Step 1 / 8

P1 starts at t = 0, is preempted at 1 and resumes where it left off at 10. The average waiting time is 6.5 ms.

SRTF Step by Step

Shortest-remaining-time-first (SRTF / SRTN): the preemptive version of SJF. When a new process arrives, its length is compared with the **remaining** time of the running one.

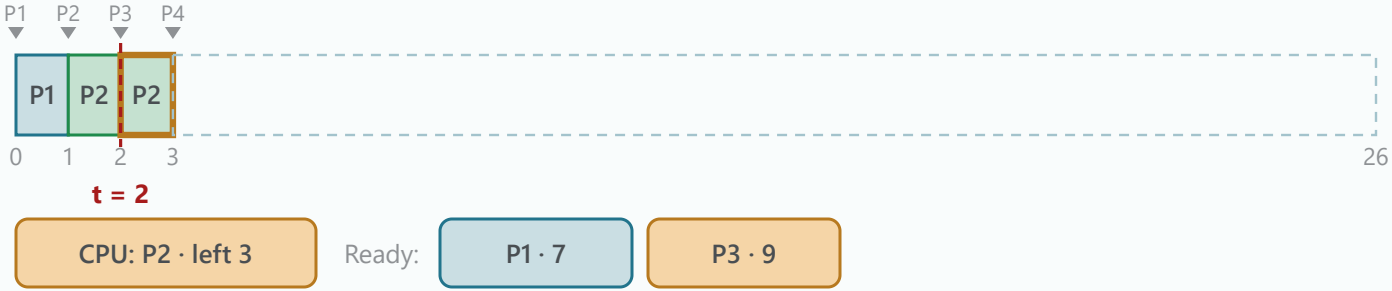


t = 1: **P2** arrives (4); the running P1 has 7 left. 4 < 7 → P1 is preempted and **P2** runs.

P1 starts at t = 0, is preempted at 1 and resumes where it left off at 10. The average waiting time is 6.5 ms.

SRTF Step by Step

Shortest-remaining-time-first (SRTF / SRTN): the preemptive version of SJF. When a new process arrives, its length is compared with the **remaining** time of the running one.



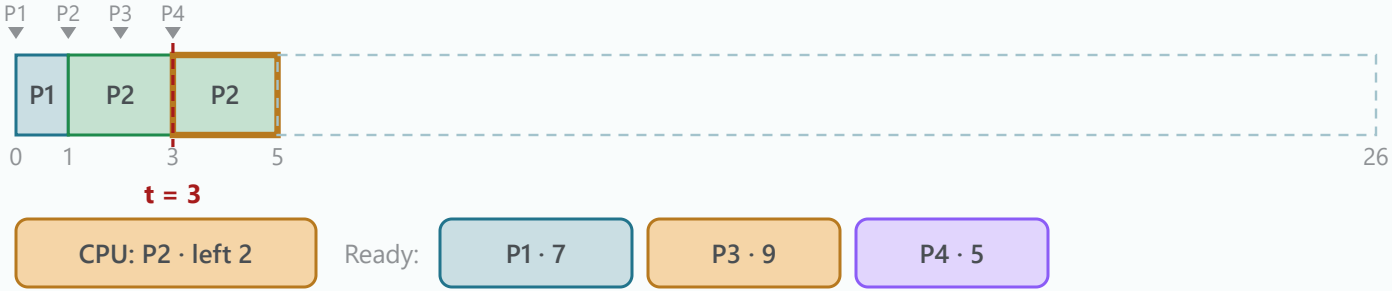
t = 2: **P3** arrives (9). The running P2 has 3 left ≤ 9 → no preemption.

Step 3 / 8

P1 starts at t = 0, is preempted at 1 and resumes where it left off at 10. The average waiting time is 6.5 ms.

SRTF Step by Step

Shortest-remaining-time-first (SRTF / SRTN): the preemptive version of SJF. When a new process arrives, its length is compared with the **remaining** time of the running one.



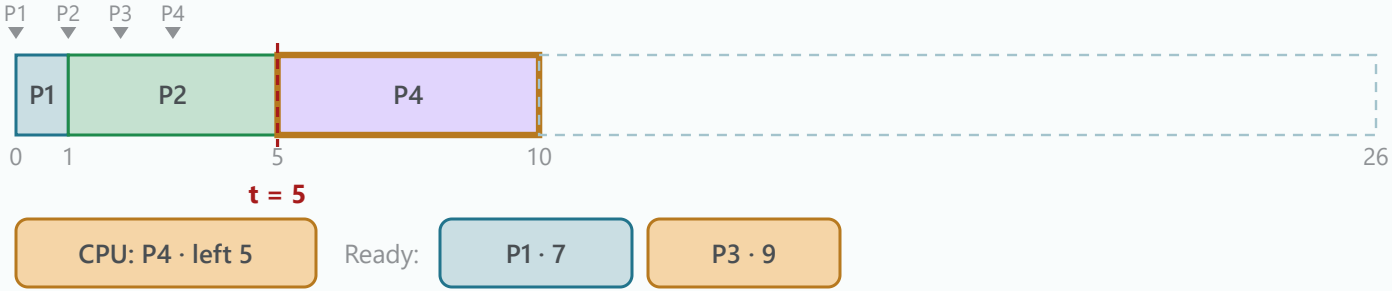
t = 3: **P4** arrives (5). The running P2 has 2 left ≤ 5 → no preemption.

Step 4 / 8

P1 starts at t = 0, is preempted at 1 and resumes where it left off at 10. The average waiting time is 6.5 ms.

SRTF Step by Step

Shortest-remaining-time-first (SRTF / SRTN): the preemptive version of SJF. When a new process arrives, its length is compared with the **remaining** time of the running one.



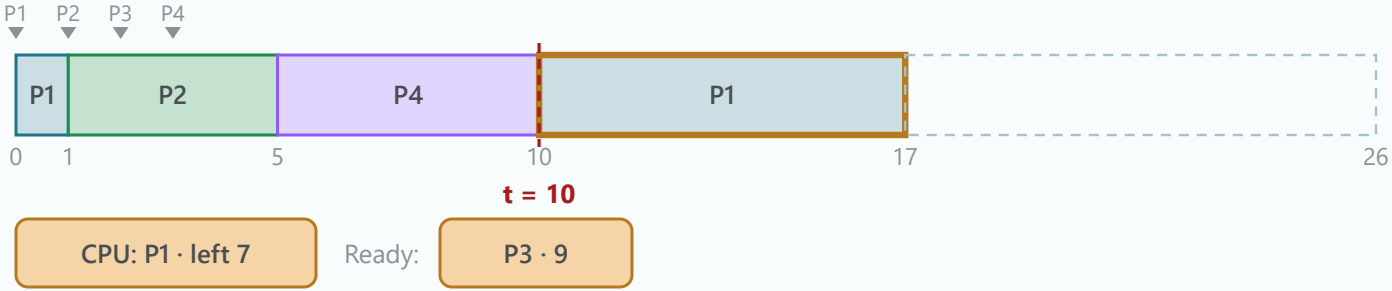
t = 5: P2 is done. Remaining times: P1 7, P3 9, P4 5 → shortest **P4** (5).

Step 5 / 8

P1 starts at t = 0, is preempted at 1 and resumes where it left off at 10. The average waiting time is 6.5 ms.

SRTF Step by Step

Shortest-remaining-time-first (SRTF / SRTN): the preemptive version of SJF. When a new process arrives, its length is compared with the **remaining** time of the running one.

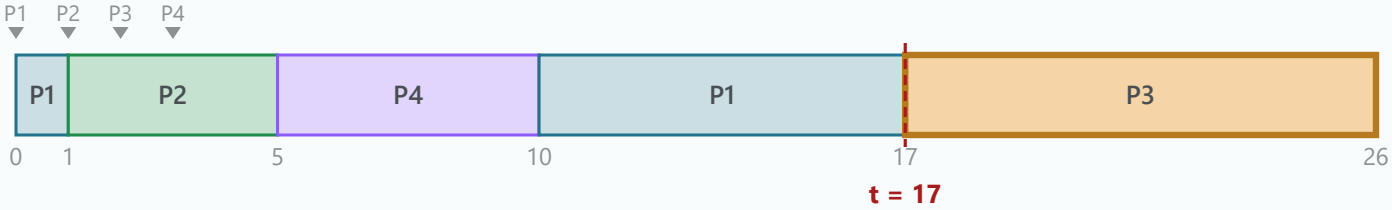


t = 10: P4 is done. Remaining times: P1 7, P3 9 → shortest **P1** (7).

P1 starts at t = 0, is preempted at 1 and resumes where it left off at 10. The average waiting time is 6.5 ms.

SRTF Step by Step

Shortest-remaining-time-first (SRTF / SRTN): the preemptive version of SJF. When a new process arrives, its length is compared with the **remaining** time of the running one.



CPU: P3 · left 9

Ready: (empty)

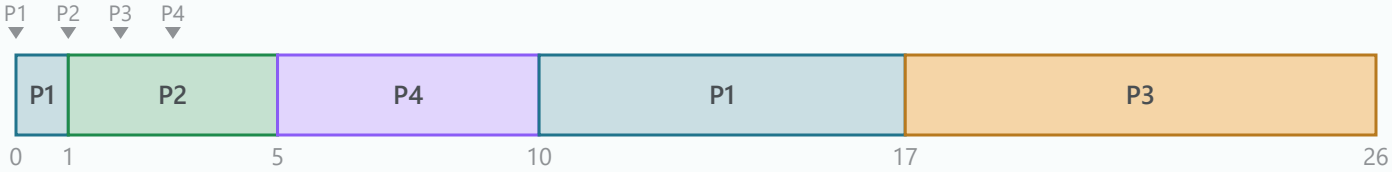
t = 17: P1 is done. Remaining times: P3 9 → shortest **P3** (9).

Step 7 / 8

P1 starts at t = 0, is preempted at 1 and resumes where it left off at 10. The average waiting time is 6.5 ms.

SRTF Step by Step

Shortest-remaining-time-first (SRTF / SRTN): the preemptive version of SJF. When a new process arrives, its length is compared with the **remaining** time of the running one.



Process	Arrival	CPU	First run	Completion	Turnaround	Waiting	Response
P1	0	8	0	17	17	9	0
P2	1	4	1	5	4	0	0
P3	2	9	17	26	24	15	15
P4	3	5	5	10	7	2	2
Avg.					**13**	**6,5**	**4,25**

Average waiting **6.5**, turnaround **13**, response **4.25** ms · 4 context switches.

Step 8 / 8

P1 starts at t = 0, is preempted at 1 and resumes where it left off at 10. The average waiting time is 6.5 ms.

Priority Scheduling

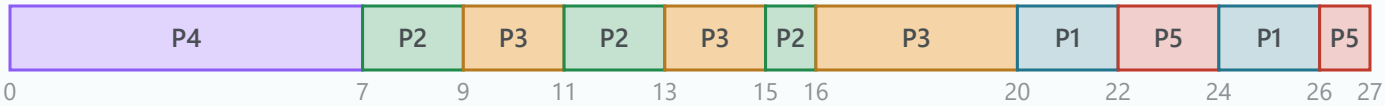
- Each process gets a **priority** number; the highest-priority ready process runs. In this course a smaller number = higher priority (Linux and Windows differ in direction).
- Priorities come from **internal** factors (memory, open files, measured I/O/CPU ratio) or **external** ones (importance, payment, policy). SJF is the special case where the priority is the predicted next burst; if all priorities are equal, the algorithm becomes FCFS.
- It can be preemptive or nonpreemptive; RR can be used among processes of the same priority (b).
- A side effect is **priority inversion**: while a high-priority job waits for a lock held by a low-priority one, medium-priority jobs cut in; the fix is priority inheritance.

(a) Nonpreemptive priority · P1 (10, pr 3), P2 (1, pr 1), P3 (2, pr 4), P4 (1, pr 5), P5 (5, pr 2), all at t = 0



Waiting: P1 6, P2 0, P3 16, P4 18, P5 1 → average 41 / 5 = 8.2 ms

(b) Priority + RR (q = 2) · P1 (4, pr 3), P2 (5, pr 2), P3 (8, pr 2), P4 (7, pr 1), P5 (3, pr 3)



Waiting: P1 22, P2 11, P3 12, P4 0, P5 24 → average 13.8 ms

Preemptive Priority on the Common Set

Preemptive priority on the common set (P1 pr 2, P2 pr 3, P3 pr 1, P4 pr 4). A newly arrived process preempts the running one if it has **higher priority**.



t = 0

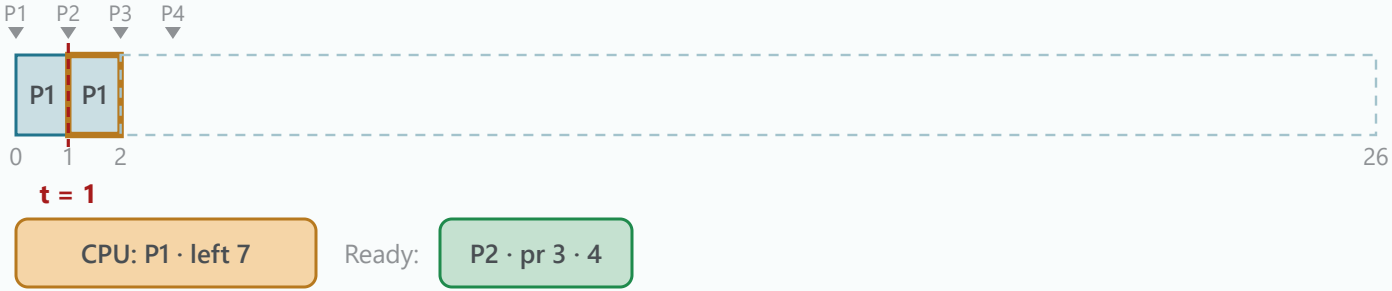
CPU: P1 · left 8

Ready: (empty)

t = 0: only **P1** exists and starts running.

Preemptive Priority on the Common Set

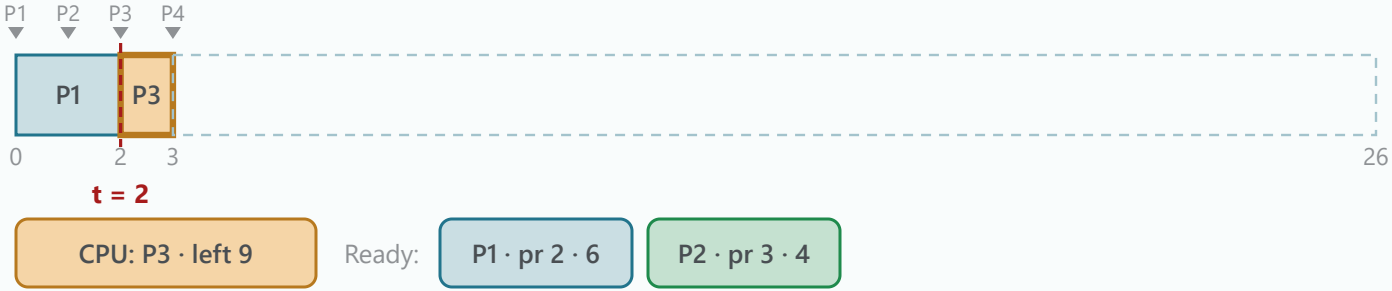
Preemptive priority on the common set (P1 pr 2, P2 pr 3, P3 pr 1, P4 pr 4). A newly arrived process preempts the running one if it has **higher priority**.



t = 1: **P2** arrives (priority 3). The running P1 has higher priority (2) → no preemption.

Preemptive Priority on the Common Set

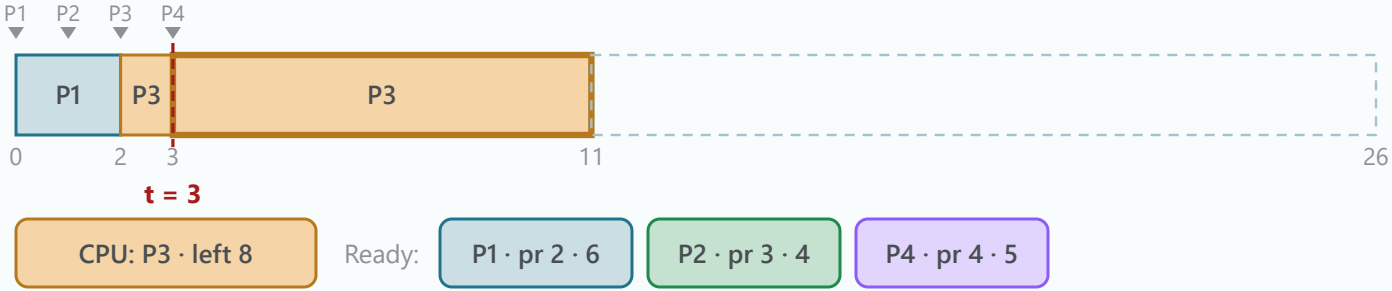
Preemptive priority on the common set (P1 pr 2, P2 pr 3, P3 pr 1, P4 pr 4). A newly arrived process preempts the running one if it has **higher priority**.



t = 2: **P3** arrives (priority 1); the running P1 has lower priority (2) → it is preempted and **P3** runs.

Preemptive Priority on the Common Set

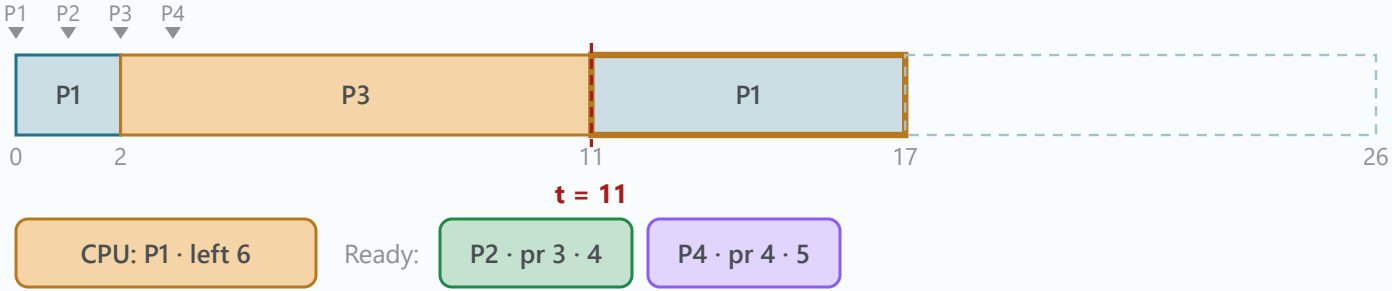
Preemptive priority on the common set (P1 pr 2, P2 pr 3, P3 pr 1, P4 pr 4). A newly arrived process preempts the running one if it has **higher priority**.



t = 3: **P4** arrives (priority 4). The running P3 has higher priority (1) → no preemption.

Preemptive Priority on the Common Set

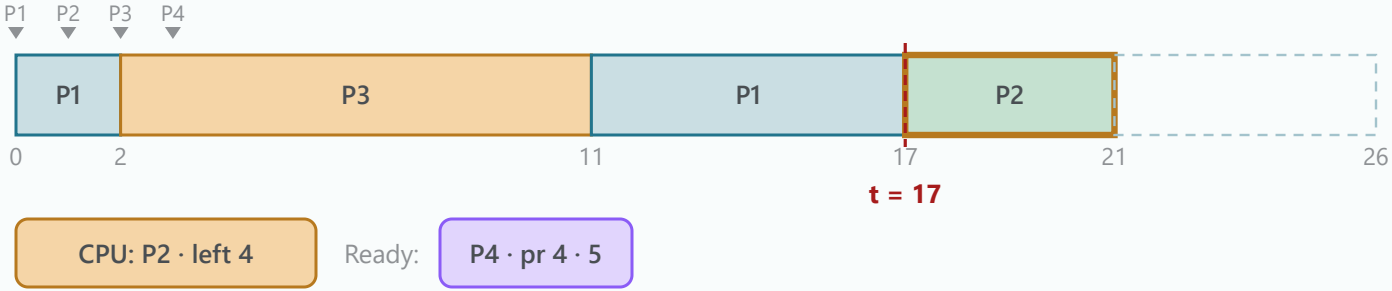
Preemptive priority on the common set (P1 pr 2, P2 pr 3, P3 pr 1, P4 pr 4). A newly arrived process preempts the running one if it has **higher priority**.



t = 11: P3 is done. Ready: P1 (pr 2), P2 (pr 3), P4 (pr 4) → highest priority **P1**.

Preemptive Priority on the Common Set

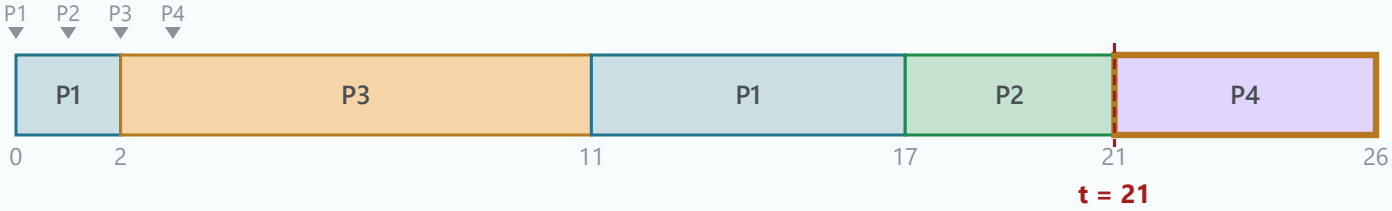
Preemptive priority on the common set (P1 pr 2, P2 pr 3, P3 pr 1, P4 pr 4). A newly arrived process preempts the running one if it has **higher priority**.



t = 17: P1 is done. Ready: P2 (pr 3), P4 (pr 4) → highest priority **P2**.

Preemptive Priority on the Common Set

Preemptive priority on the common set (P1 pr 2, P2 pr 3, P3 pr 1, P4 pr 4). A newly arrived process preempts the running one if it has **higher priority**.



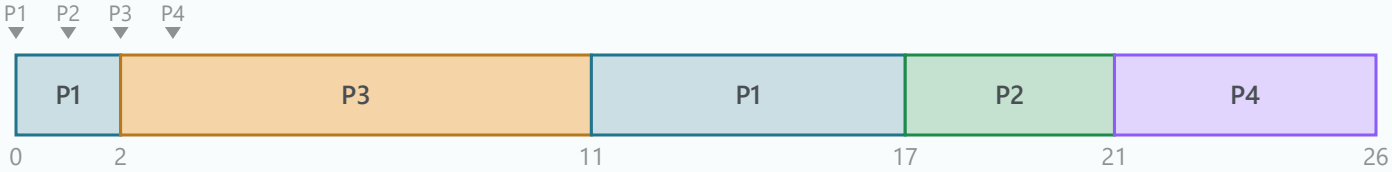
CPU: P4 · left 5

Ready: (empty)

t = 21: P2 is done. Ready: P4 (pr 4) → highest priority **P4**.

Preemptive Priority on the Common Set

Preemptive priority on the common set (P1 pr 2, P2 pr 3, P3 pr 1, P4 pr 4). A newly arrived process preempts the running one if it has **higher priority**.

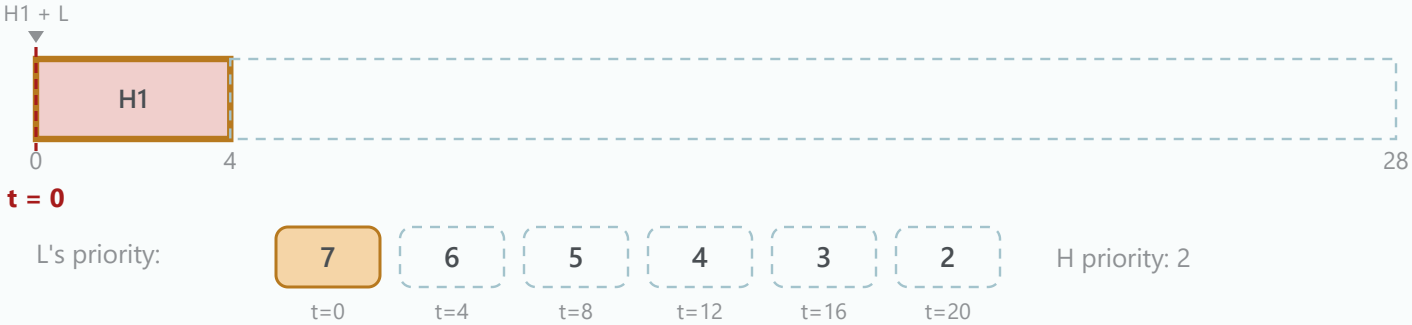


Process	Arrival	CPU	Priority	First run	Completion	Turnaround	Waiting	Response
P1	0	8	2	0	17	17	9	0
P2	1	4	3	17	21	20	16	16
P3	2	9	1	2	11	9	0	0
P4	3	5	4	21	26	23	18	18
Avg.						**17,25**	**10,75**	**8,5**

Average waiting **10.75**, turnaround **17.25**, response **8.5** ms · 4 context switches.

Starvation and Aging

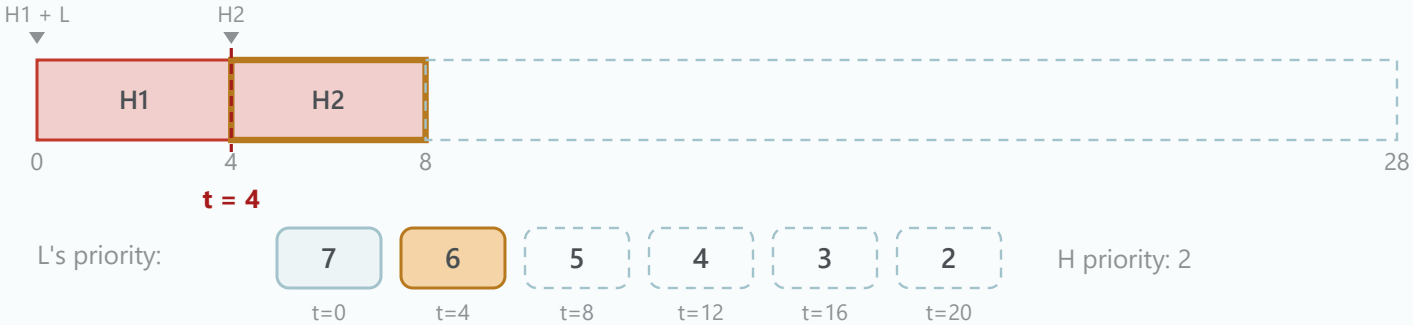
Starvation: if high-priority jobs keep arriving, a low-priority job never runs. **Aging**: its priority is raised step by step while it waits. Rule: for every 4 ms spent ready, the priority number drops by 1. L (3 ms, pr 7) arrives at t = 0; a new H (4 ms, pr 2) arrives every 4 ms. Nonpreemptive priority.



t = 0: L's priority is 7, the newly arrived H1's is 2 → **H1** runs. After L waits another 4 ms its priority becomes 6.

Starvation and Aging

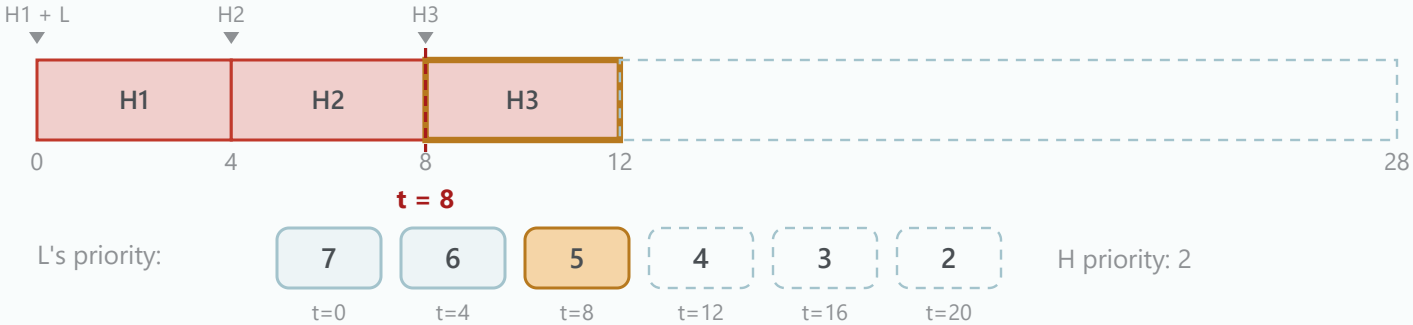
Starvation: if high-priority jobs keep arriving, a low-priority job never runs. **Aging**: its priority is raised step by step while it waits. Rule: for every 4 ms spent ready, the priority number drops by 1. L (3 ms, pr 7) arrives at t = 0; a new H (4 ms, pr 2) arrives every 4 ms. Nonpreemptive priority.



t = 4: L's priority is 6, the newly arrived H2's is 2 → **H2** runs. After L waits another 4 ms its priority becomes 5.

Starvation and Aging

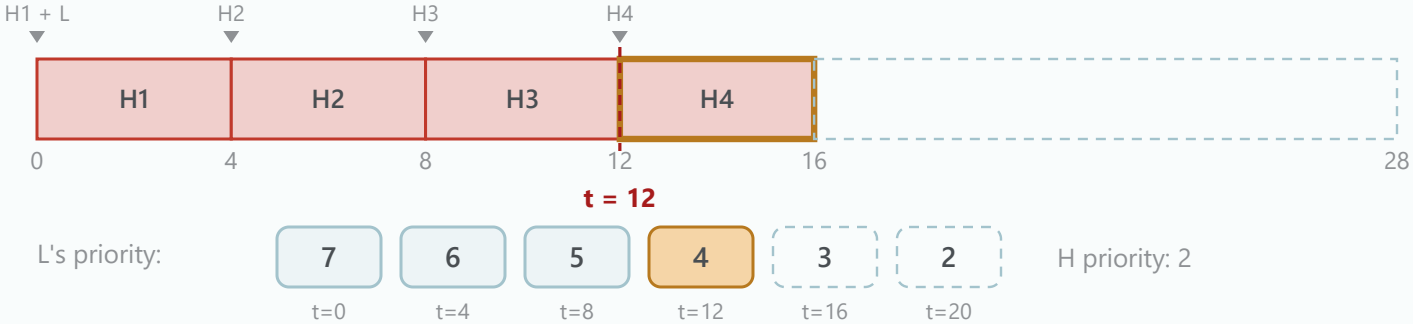
Starvation: if high-priority jobs keep arriving, a low-priority job never runs. **Aging**: its priority is raised step by step while it waits. Rule: for every 4 ms spent ready, the priority number drops by 1. L (3 ms, pr 7) arrives at t = 0; a new H (4 ms, pr 2) arrives every 4 ms. Nonpreemptive priority.



t = 8: L's priority is 5, the newly arrived H3's is 2 → **H3** runs. After L waits another 4 ms its priority becomes 4.

Starvation and Aging

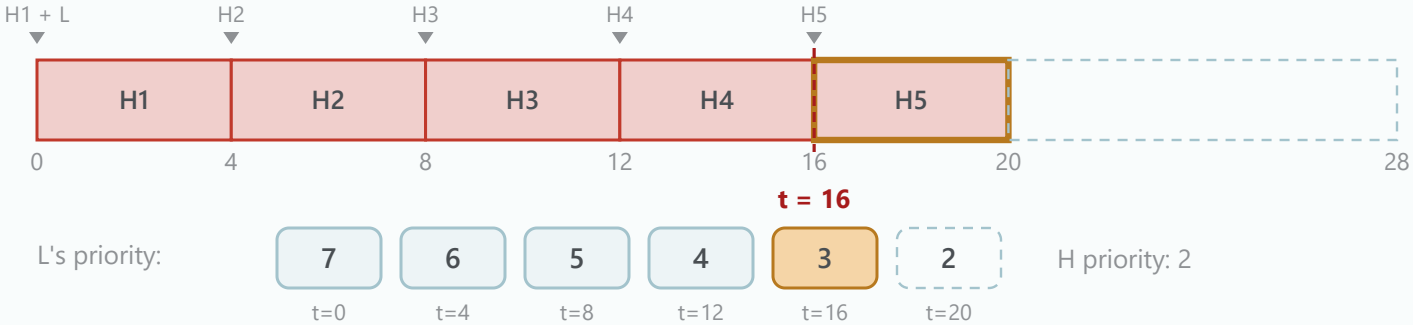
Starvation: if high-priority jobs keep arriving, a low-priority job never runs. **Aging**: its priority is raised step by step while it waits. Rule: for every 4 ms spent ready, the priority number drops by 1. L (3 ms, pr 7) arrives at t = 0; a new H (4 ms, pr 2) arrives every 4 ms. Nonpreemptive priority.



t = 12: L's priority is 4, the newly arrived H4's is 2 → **H4** runs. After L waits another 4 ms its priority becomes 3.

Starvation and Aging

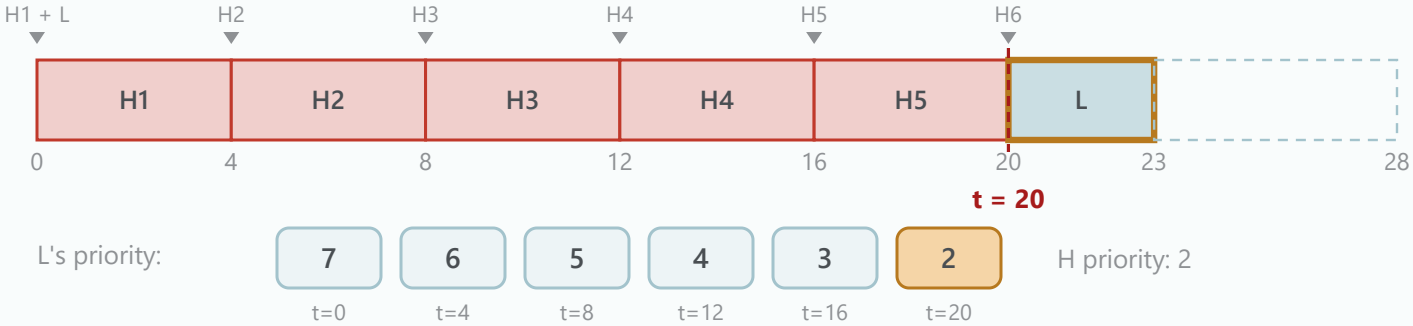
Starvation: if high-priority jobs keep arriving, a low-priority job never runs. **Aging**: its priority is raised step by step while it waits. Rule: for every 4 ms spent ready, the priority number drops by 1. L (3 ms, pr 7) arrives at t = 0; a new H (4 ms, pr 2) arrives every 4 ms. Nonpreemptive priority.



t = 16: L's priority is 3, the newly arrived H5's is 2 → **H5** runs. After L waits another 4 ms its priority becomes 2.

Starvation and Aging

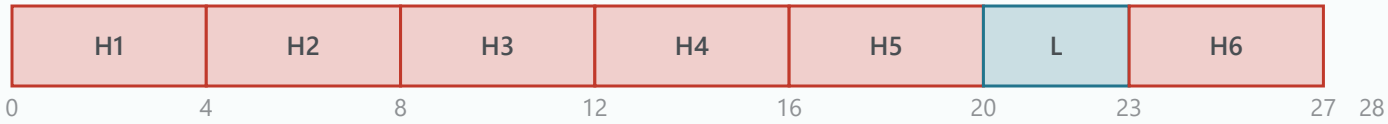
Starvation: if high-priority jobs keep arriving, a low-priority job never runs. **Aging**: its priority is raised step by step while it waits. Rule: for every 4 ms spent ready, the priority number drops by 1. L (3 ms, pr 7) arrives at t = 0; a new H (4 ms, pr 2) arrives every 4 ms. Nonpreemptive priority.



t = 20: L's priority is **2**, equal to H6's; ties go to the earlier arrival → **L** finally runs (it waited 20 ms).

Starvation and Aging

Starvation: if high-priority jobs keep arriving, a low-priority job never runs. **Aging**: its priority is raised step by step while it waits. Rule: for every 4 ms spent ready, the priority number drops by 1. L (3 ms, pr 7) arrives at t = 0; a new H (4 ms, pr 2) arrives every 4 ms. Nonpreemptive priority.

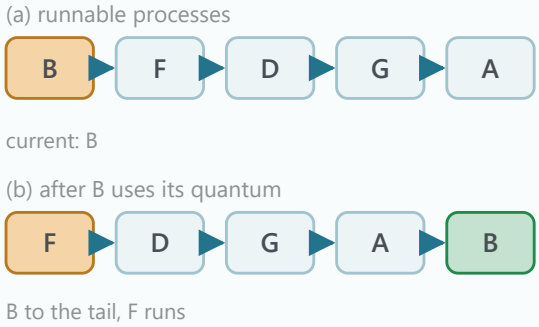


Without aging: as long as H keeps coming, L's priority stays at 7 and L never runs (starvation).

Aging turns an unbounded wait into a **bounded** one: in the worst case L waits priority gap × 4 ms = 20 ms.

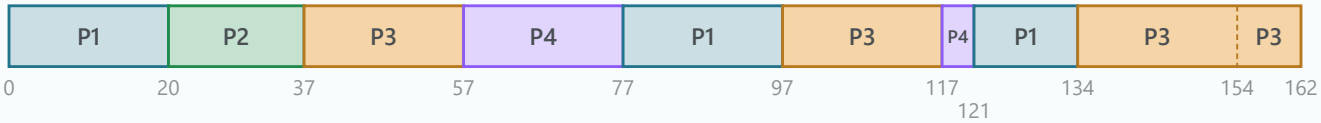
Round Robin (RR)

- **Round robin**: FCFS + preemption. Each process runs for at most one **time quantum** (q), then goes to the tail of the queue.
- With n ready processes (a fixed number) and quantum q, ignoring context switches, no process waits longer than (n − 1) · q: **response time is bounded**, and there is no starvation.
- Turnaround is usually worse than SJF, response time better.
- An I/O-bound process uses a small part of its quantum and blocks for I/O (moving to a wait queue), returning to the tail of the ready queue when the I/O completes; CPU-bound processes get a bigger share (hence refinements such as MLFQ and virtual RR).



Example: q = 20 · P1 53, P2 17, P3 68, P4 24 (all at t = 0)

thin line at 154: two consecutive quanta of P3



Waiting: P1 81, P2 20, P3 94, P4 97 → avg. 73 ms · avg. response 28.5 ms (SJF: avg. waiting 38)

RR Step by Step: q = 2

The common set, **q = 2 ms**. A small quantum: processes swap often; every process gets its first turn quickly.



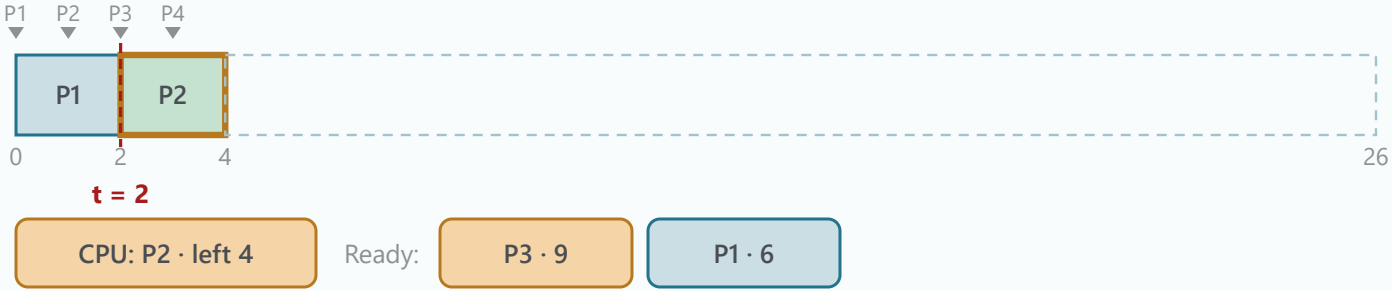
t = 0

CPU: P1 · left 8 Ready: (empty)

t = 0: only **P1** exists; it runs for at most q = 2 ms, then gives up its turn.

RR Step by Step: q = 2

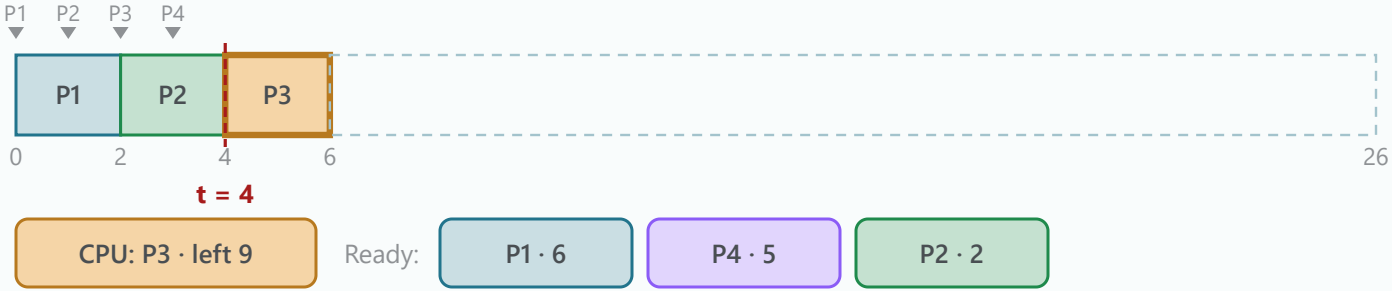
The common set, **q = 2 ms**. A small quantum: processes swap often; every process gets its first turn quickly.



t = 2: P1 used up its quantum (6 left) and goes to the tail. P2 and P3, which arrived meanwhile, entered the queue before it. Next: **P2**.

RR Step by Step: q = 2

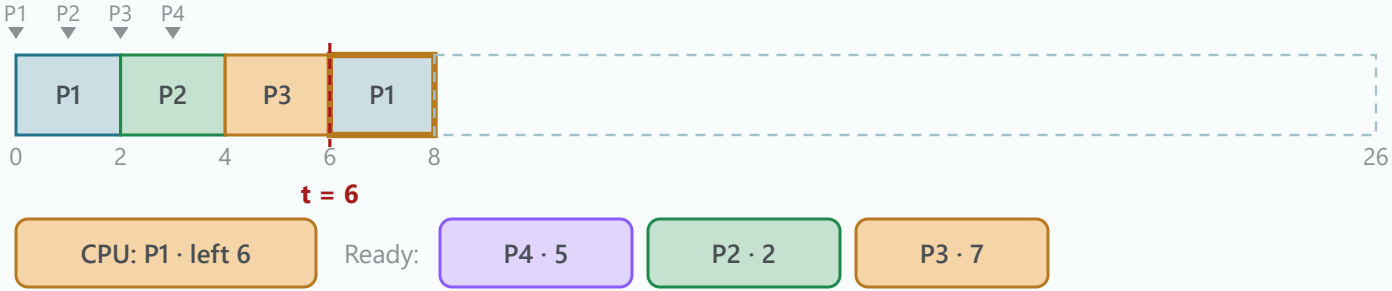
The common set, **q = 2 ms**. A small quantum: processes swap often; every process gets its first turn quickly.



t = 4: P2 used up its quantum (2 left) and goes to the tail. P4, which arrived meanwhile, entered the queue before it. Next: **P3**.

RR Step by Step: q = 2

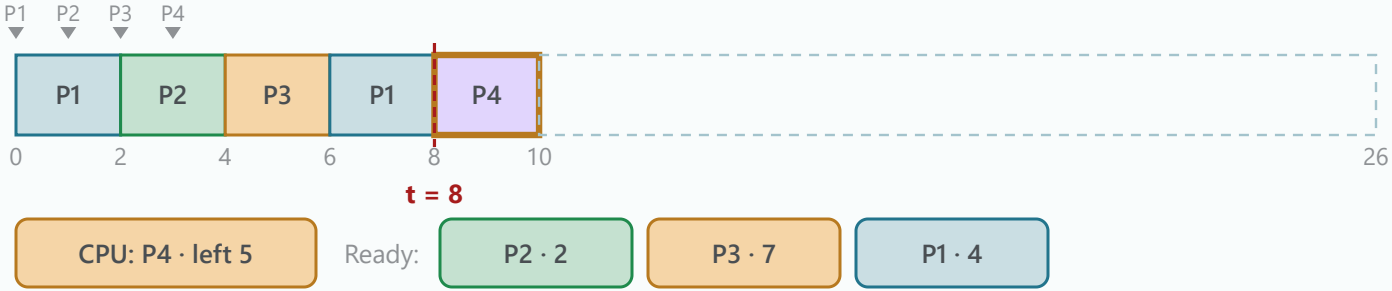
The common set, **q = 2 ms**. A small quantum: processes swap often; every process gets its first turn quickly.



t = 6: P3 used up its quantum (7 left) and goes to the tail. Next: **P1**.

RR Step by Step: q = 2

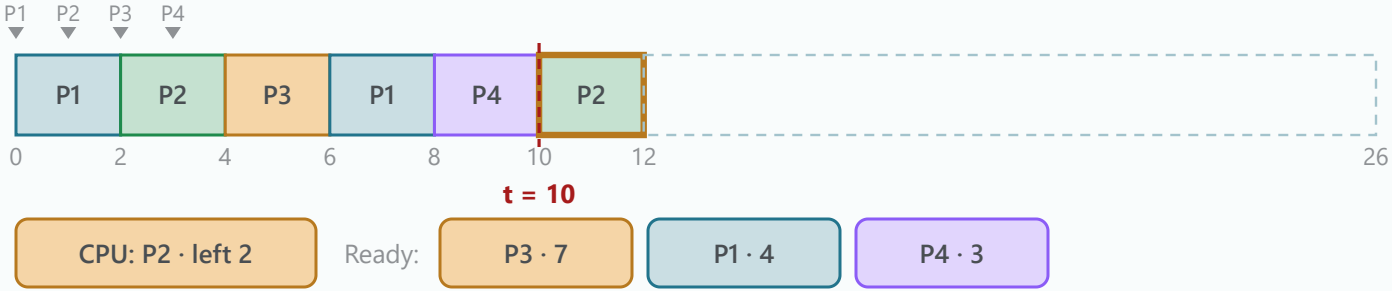
The common set, **q = 2 ms**. A small quantum: processes swap often; every process gets its first turn quickly.



t = 8: P1 used up its quantum (4 left) and goes to the tail. Next: **P4**.

RR Step by Step: q = 2

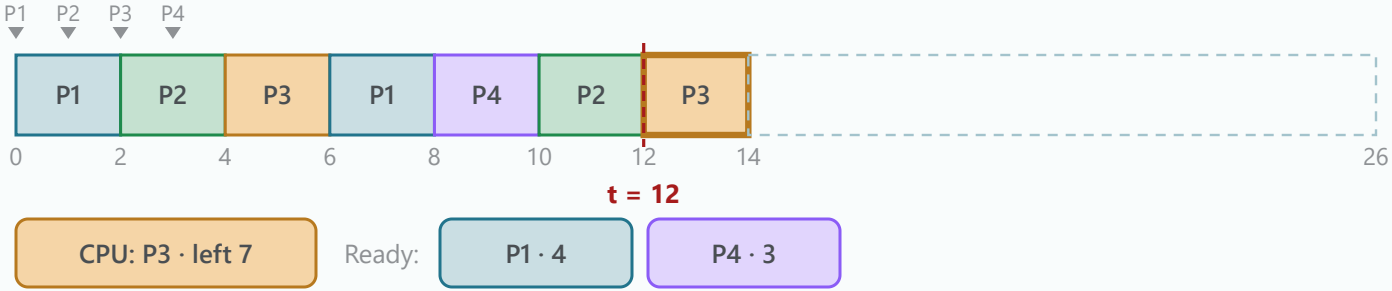
The common set, **q = 2 ms**. A small quantum: processes swap often; every process gets its first turn quickly.



t = 10: P4 used up its quantum (3 left) and goes to the tail. Next: **P2**.

RR Step by Step: q = 2

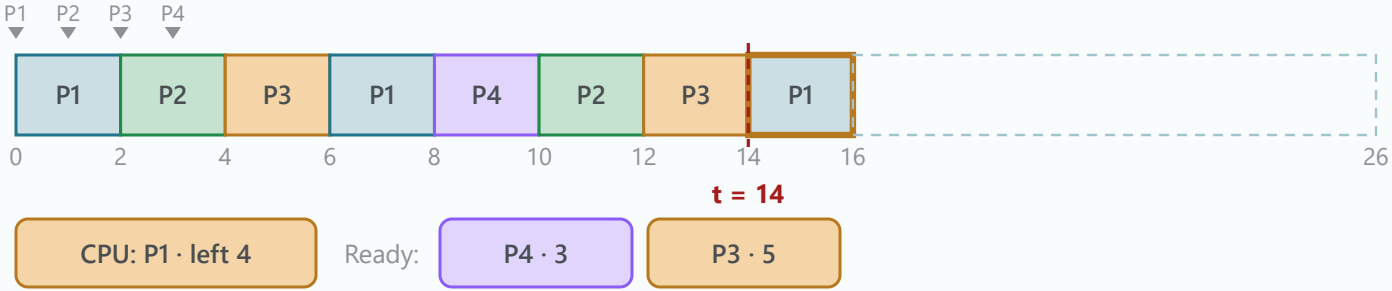
The common set, **q = 2 ms**. A small quantum: processes swap often; every process gets its first turn quickly.



t = 12: P2 finished before its quantum ran out. Next: **P3**.

RR Step by Step: q = 2

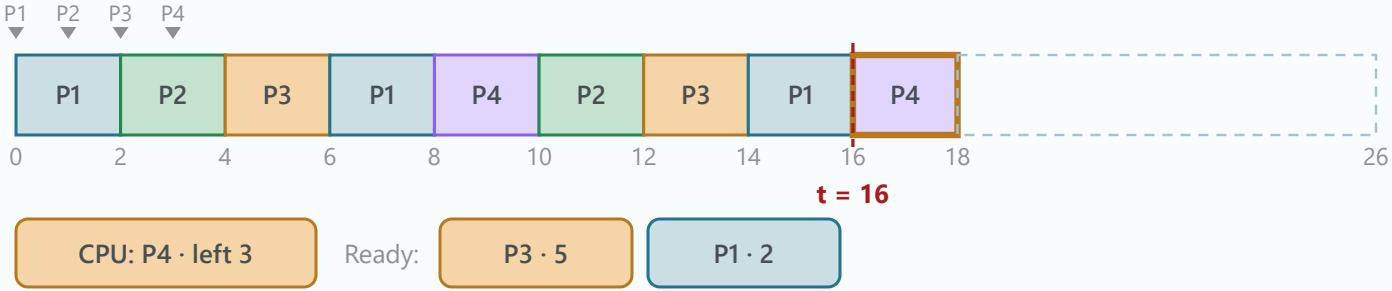
The common set, **q = 2 ms**. A small quantum: processes swap often; every process gets its first turn quickly.



t = 14: P3 used up its quantum (5 left) and goes to the tail. Next: **P1**.

RR Step by Step: q = 2

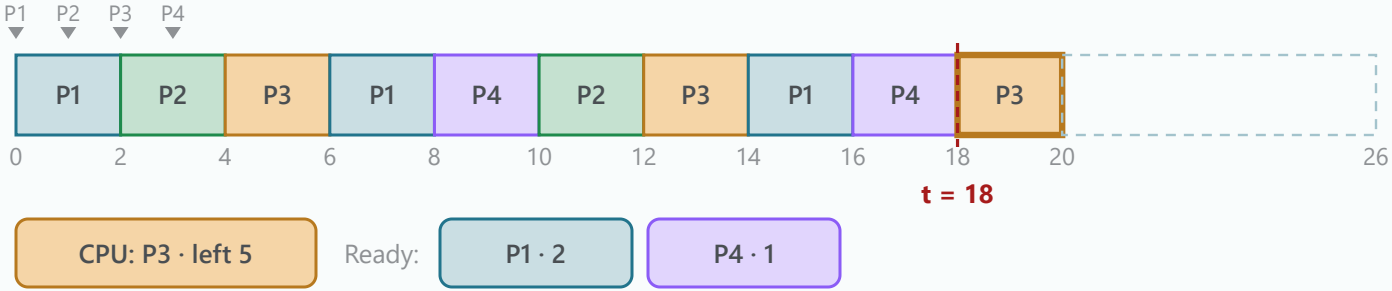
The common set, **q = 2 ms**. A small quantum: processes swap often; every process gets its first turn quickly.



t = 16: P1 used up its quantum (2 left) and goes to the tail. Next: **P4**.

RR Step by Step: q = 2

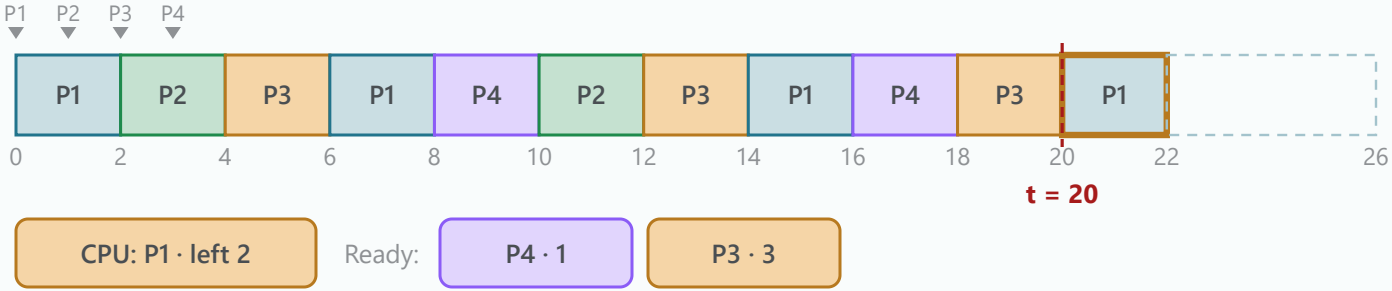
The common set, **q = 2 ms**. A small quantum: processes swap often; every process gets its first turn quickly.



t = 18: P4 used up its quantum (1 left) and goes to the tail. Next: **P3**.

RR Step by Step: q = 2

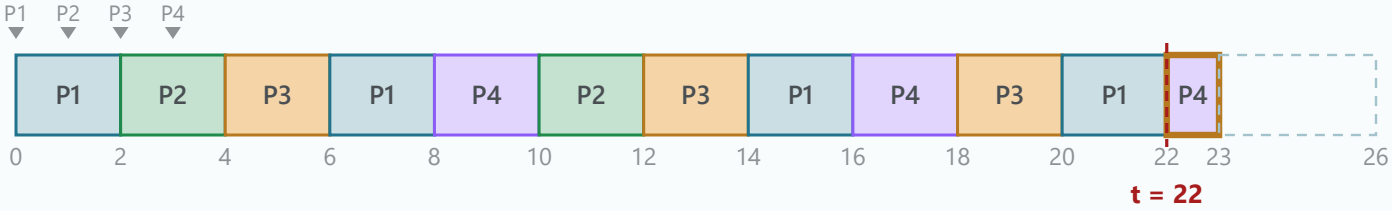
The common set, **q = 2 ms**. A small quantum: processes swap often; every process gets its first turn quickly.



t = 20: P3 used up its quantum (3 left) and goes to the tail. Next: **P1**.

RR Step by Step: q = 2

The common set, **q = 2 ms**. A small quantum: processes swap often; every process gets its first turn quickly.

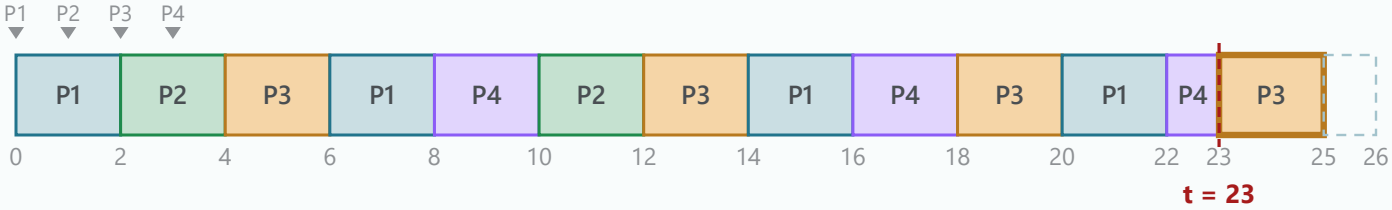


CPU: P4 · left 1 | Ready: P3 · 3

t = 22: P1 finished before its quantum ran out. Next: **P4**.

RR Step by Step: q = 2

The common set, **q = 2 ms**. A small quantum: processes swap often; every process gets its first turn quickly.



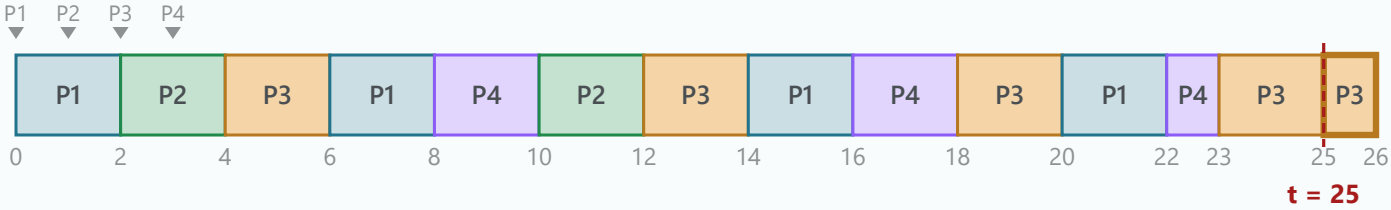
CPU: P3 · left 3

Ready: (empty)

t = 23: P4 finished before its quantum ran out. Next: **P3**.

RR Step by Step: q = 2

The common set, **q = 2 ms**. A small quantum: processes swap often; every process gets its first turn quickly.



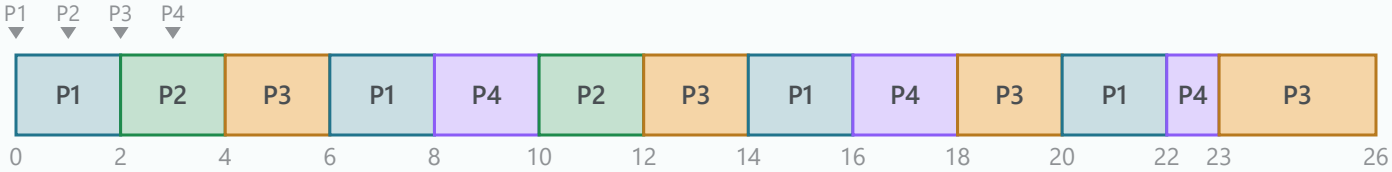
CPU: P3 · left 1

Ready: (empty)

t = 25: P3 used up its quantum (1 left), but no other process is in the queue → it gets a new quantum with no context switch.

RR Step by Step: q = 2

The common set, **q = 2 ms**. A small quantum: processes swap often; every process gets its first turn quickly.



Process	Arrival	CPU	First run	Completion	Turnaround	Waiting	Response
P1	0	8	0	22	22	14	0
P2	1	4	2	12	11	7	1
P3	2	9	4	26	24	15	2
P4	3	5	8	23	20	15	5
Avg.					**19,25**	**12,75**	**2**

Average waiting **12.75**, turnaround **19.25**, response **2** ms · 12 context switches.

RR Step by Step: q = 5

The common set, **q = 5 ms**. A larger quantum: fewer switches, but longer response time.



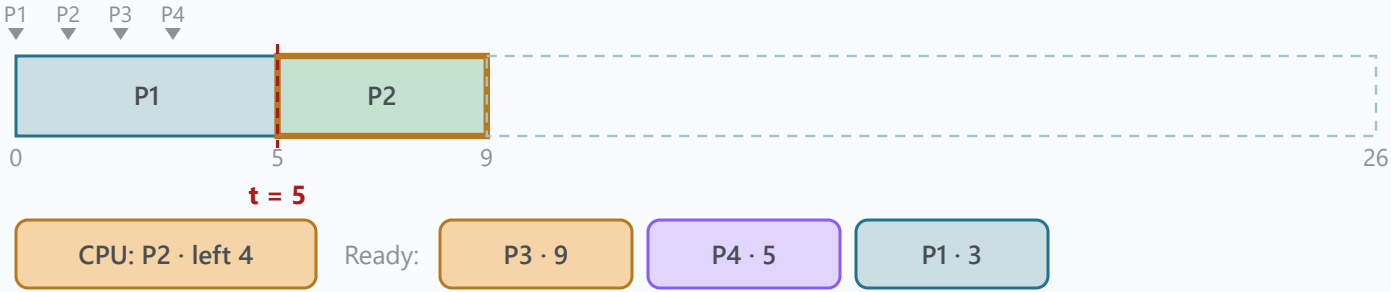
t = 0

CPU: P1 · left 8 Ready: (empty)

t = 0: only **P1** exists; it runs for at most q = 5 ms, then gives up its turn.

RR Step by Step: q = 5

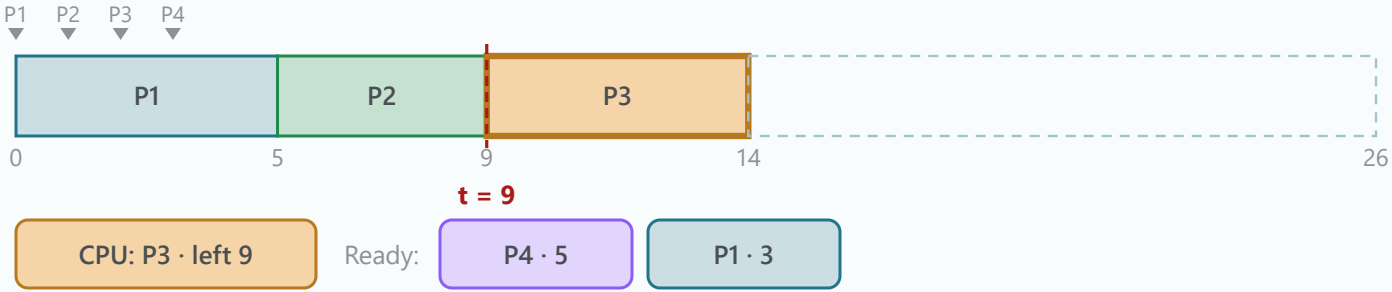
The common set, **q = 5 ms**. A larger quantum: fewer switches, but longer response time.



t = 5: P1 used up its quantum (3 left) and goes to the tail. P2, P3 and P4, which arrived meanwhile, entered the queue before it. Next: **P2**.

RR Step by Step: q = 5

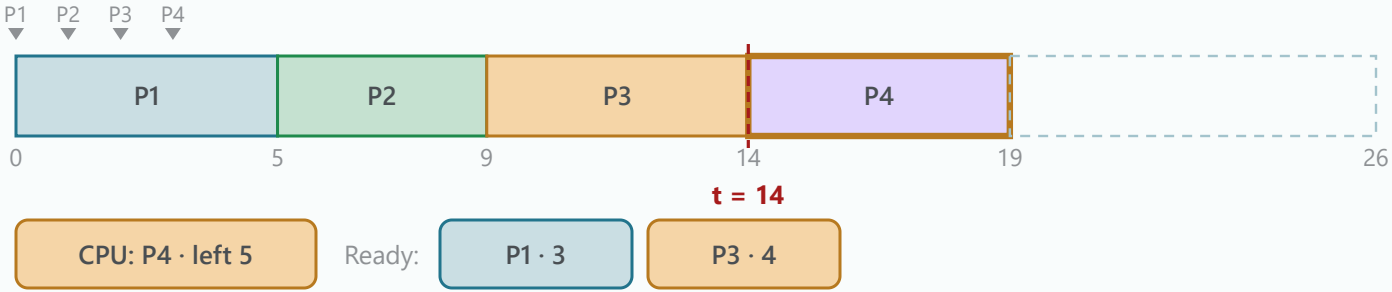
The common set, **q = 5 ms**. A larger quantum: fewer switches, but longer response time.



t = 9: P2 finished before its quantum ran out. Next: **P3**.

RR Step by Step: q = 5

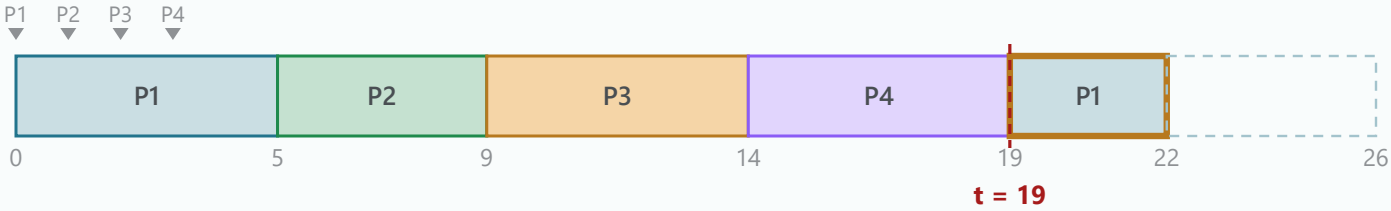
The common set, **q = 5 ms**. A larger quantum: fewer switches, but longer response time.



t = 14: P3 used up its quantum (4 left) and goes to the tail. Next: **P4**.

RR Step by Step: q = 5

The common set, **q = 5 ms**. A larger quantum: fewer switches, but longer response time.

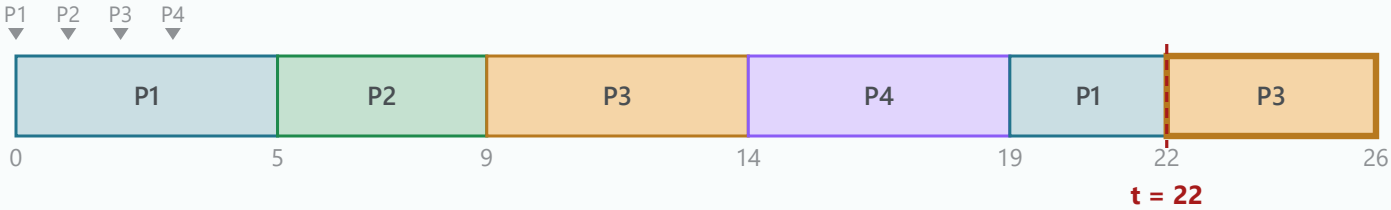


CPU: P1 · left 3 Ready: P3 · 4

t = 19: P4 finished before its quantum ran out. Next: **P1**.

RR Step by Step: q = 5

The common set, **q = 5 ms**. A larger quantum: fewer switches, but longer response time.



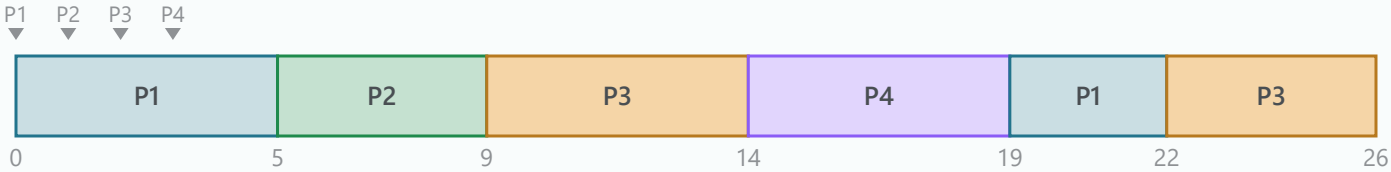
CPU: P3 · left 4

Ready: (empty)

t = 22: P1 finished before its quantum ran out. Next: **P3**.

RR Step by Step: q = 5

The common set, **q = 5 ms**. A larger quantum: fewer switches, but longer response time.



Process	Arrival	CPU	First run	Completion	Turnaround	Waiting	Response
P1	0	8	0	22	22	14	0
P2	1	4	5	9	8	4	4
P3	2	9	9	26	24	15	7
P4	3	5	14	19	16	11	11
Avg.					**17,5**	**11**	**5,5**

Average waiting **11**, turnaround **17.5**, response **5.5** ms · 5 context switches.

Round Robin: Code Trace

```
1  /* p[] sorted by arrival time */
2  void rr(Proc p[], int n, int q) {
▶    int t = 0, k = 0;     /* k: next arrival */
▶    add_arrivals(p, n, &k, t);
5    while (!empty() || k < n) {
6      if (empty()) {       /* CPU idle */
7        t = p[k].arrival;  /* next arrival */
8        add_arrivals(p, n, &k, t);
9      }
10     Proc *s = dequeue(); /* head */
11     int slice = min(q, s->left);
12     t += slice; s->left -= slice;
13     add_arrivals(p, n, &k, t);  /* first */
14     if (s->left > 0) enqueue(s); /* then */
15     else printf("%s %d\n", s->name, t);
16   }
17 }
```



CPU: idle Ready: P1·8

t = 0: only P1 has arrived; it enters the queue. The quantum is `q = 4` ms.

Step 1 / 10

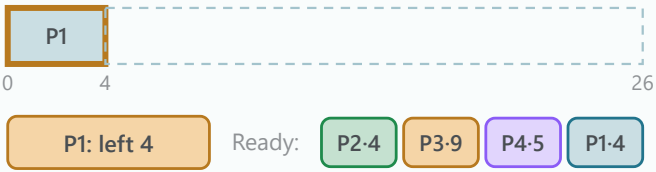
VARIABLES

`q = 4` `t = 0` `queue = [P1]`

OUTPUT

Round Robin: Code Trace

```
1  /* p[] sorted by arrival time */
2  void rr(Proc p[], int n, int q) {
3    int t = 0, k = 0;      /* k: next arrival */
4    add_arrivals(p, n, &k, t);
5    while (!empty() || k < n) {
6      if (empty()) {       /* CPU idle */
7        t = p[k].arrival;  /* next arrival */
8        add_arrivals(p, n, &k, t);
9      }
10     Proc *s = dequeue(); /* head */
▶      int slice = min(q, s->left);
▶      t += slice; s->left -= slice;
13     add_arrivals(p, n, &k, t);  /* first */
▶      if (s->left > 0) enqueue(s); /* then */
15     else printf("%s %d\n", s->name, t);
16   }
17 }
```



t = 0–4: P1 leaves the queue and runs 4 ms (4 left). Meanwhile P2, P3, P4 arrived; they enter the queue first, then P1.

Step 2 / 10

VARIABLES

`t = 4` `s = P1` `slice = 4` `s->left = 4`

`queue = [P2, P3, P4, P1]`

OUTPUT

Round Robin: Code Trace

```
1  /* p[] sorted by arrival time */
2  void rr(Proc p[], int n, int q) {
3    int t = 0, k = 0;      /* k: next arrival */
4    add_arrivals(p, n, &k, t);
5    while (!empty() || k < n) {
6      if (empty()) {       /* CPU idle */
7        t = p[k].arrival;  /* next arrival */
8        add_arrivals(p, n, &k, t);
9      }
10     Proc *s = dequeue(); /* head */
▶      int slice = min(q, s->left);
▶      t += slice; s->left -= slice;
13     add_arrivals(p, n, &k, t);  /* first */
14     if (s->left > 0) enqueue(s); /* then */
▶      else printf("%s %d\n", s->name, t);
16   }
17 }
```



t = 8: P2 ran its last 4 ms and is done; its completion time is printed.

Step 3 / 10

VARIABLES

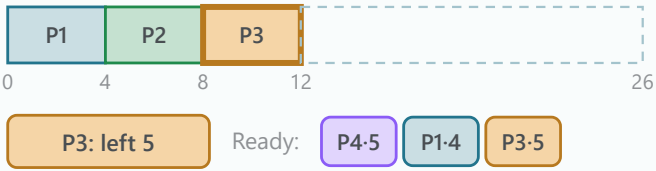
`t = 8` `s = P2` `slice = 4` `s->left = 0` `queue = [P3, P4, P1]`

OUTPUT

```
P2 8
```

Round Robin: Code Trace

```
1  /* p[] sorted by arrival time */
2  void rr(Proc p[], int n, int q) {
3    int t = 0, k = 0;      /* k: next arrival */
4    add_arrivals(p, n, &k, t);
5    while (!empty() || k < n) {
6      if (empty()) {       /* CPU idle */
7        t = p[k].arrival;  /* next arrival */
8        add_arrivals(p, n, &k, t);
9      }
10     Proc *s = dequeue(); /* head */
▶      int slice = min(q, s->left);
▶      t += slice; s->left -= slice;
13     add_arrivals(p, n, &k, t);  /* first */
▶      if (s->left > 0) enqueue(s); /* then */
15     else printf("%s %d\n", s->name, t);
16   }
17 }
```



t = 12: P3 used up its quantum (5 left) and returns to the tail.

Step 4 / 10

VARIABLES

`t = 12` `s = P3` `slice = 4` `s->left = 5`

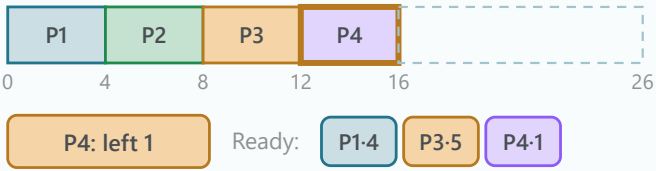
`queue = [P4, P1, P3]`

OUTPUT

```
P2 8
```

Round Robin: Code Trace

```
1  /* p[] sorted by arrival time */
2  void rr(Proc p[], int n, int q) {
3    int t = 0, k = 0;      /* k: next arrival */
4    add_arrivals(p, n, &k, t);
5    while (!empty() || k < n) {
6      if (empty()) {       /* CPU idle */
7        t = p[k].arrival;  /* next arrival */
8        add_arrivals(p, n, &k, t);
9      }
10     Proc *s = dequeue(); /* head */
▶      int slice = min(q, s->left);
▶      t += slice; s->left -= slice;
13     add_arrivals(p, n, &k, t);  /* first */
▶      if (s->left > 0) enqueue(s); /* then */
15     else printf("%s %d\n", s->name, t);
16   }
17 }
```



t = 16: P4 used up its quantum (1 left) and returns to the tail.

Step 5 / 10

VARIABLES

`t = 16` `s = P4` `slice = 4` `s->left = 1`

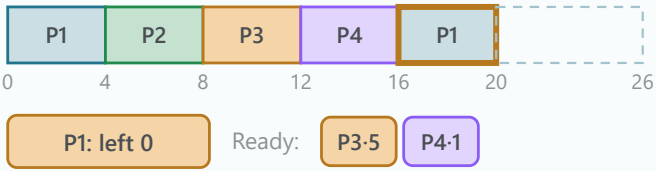
`queue = [P1, P3, P4]`

OUTPUT

```
P2 8
```

Round Robin: Code Trace

```
1  /* p[] sorted by arrival time */
2  void rr(Proc p[], int n, int q) {
3    int t = 0, k = 0;      /* k: next arrival */
4    add_arrivals(p, n, &k, t);
5    while (!empty() || k < n) {
6      if (empty()) {       /* CPU idle */
7        t = p[k].arrival;  /* next arrival */
8        add_arrivals(p, n, &k, t);
9      }
10     Proc *s = dequeue(); /* head */
▶      int slice = min(q, s->left);
▶      t += slice; s->left -= slice;
13     add_arrivals(p, n, &k, t);  /* first */
14     if (s->left > 0) enqueue(s); /* then */
▶      else printf("%s %d\n", s->name, t);
16   }
17 }
```



t = 20: P1 ran its last 4 ms and is done; its completion time is printed.

Step 6 / 10

VARIABLES

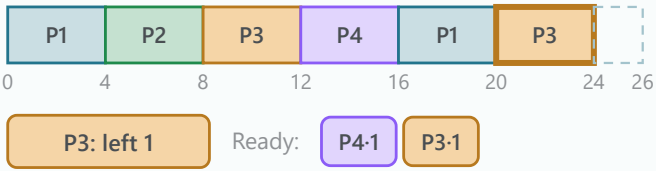
`t = 20` `s = P1` `slice = 4` `s->left = 0` `queue = [P3, P4]`

OUTPUT

```
P2 8
P1 20
```

Round Robin: Code Trace

```
1  /* p[] sorted by arrival time */
2  void rr(Proc p[], int n, int q) {
3    int t = 0, k = 0;      /* k: next arrival */
4    add_arrivals(p, n, &k, t);
5    while (!empty() || k < n) {
6      if (empty()) {       /* CPU idle */
7        t = p[k].arrival;  /* next arrival */
8        add_arrivals(p, n, &k, t);
9      }
10     Proc *s = dequeue(); /* head */
▶      int slice = min(q, s->left);
▶      t += slice; s->left -= slice;
13     add_arrivals(p, n, &k, t);  /* first */
▶      if (s->left > 0) enqueue(s); /* then */
15     else printf("%s %d\n", s->name, t);
16   }
17 }
```



t = 24: P3 used up its quantum (1 left) and returns to the tail.

Step 7 / 10

VARIABLES

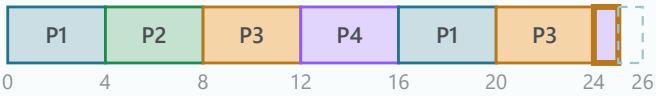
`t = 24` `s = P3` `slice = 4` `s->left = 1` `queue = [P4, P3]`

OUTPUT

```
P2 8
P1 20
```

Round Robin: Code Trace

```
1  /* p[] sorted by arrival time */
2  void rr(Proc p[], int n, int q) {
3    int t = 0, k = 0;      /* k: next arrival */
4    add_arrivals(p, n, &k, t);
5    while (!empty() || k < n) {
6      if (empty()) {       /* CPU idle */
7        t = p[k].arrival;  /* next arrival */
8        add_arrivals(p, n, &k, t);
9      }
10     Proc *s = dequeue(); /* head */
▶      int slice = min(q, s->left);
▶      t += slice; s->left -= slice;
13     add_arrivals(p, n, &k, t);  /* first */
14     if (s->left > 0) enqueue(s); /* then */
▶      else printf("%s %d\n", s->name, t);
16   }
17 }
```



P4: left 0 Ready: P3·1

t = 25: P4 finished its last 1 ms before the quantum ran out (`slice = 1`); it does not go back in the queue.

Step 8 / 10

VARIABLES

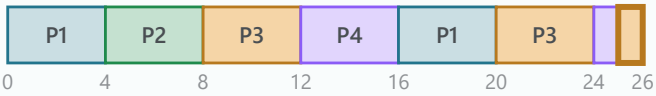
`t = 25` `s = P4` `slice = 1` `s->left = 0` `queue = [P3]`

OUTPUT

```
P2 8
P1 20
P4 25
```

Round Robin: Code Trace

```
1  /* p[] sorted by arrival time */
2  void rr(Proc p[], int n, int q) {
3    int t = 0, k = 0;      /* k: next arrival */
4    add_arrivals(p, n, &k, t);
5    while (!empty() || k < n) {
6      if (empty()) {       /* CPU idle */
7        t = p[k].arrival;  /* next arrival */
8        add_arrivals(p, n, &k, t);
9      }
10     Proc *s = dequeue(); /* head */
▶      int slice = min(q, s->left);
▶      t += slice; s->left -= slice;
13     add_arrivals(p, n, &k, t);  /* first */
14     if (s->left > 0) enqueue(s); /* then */
▶      else printf("%s %d\n", s->name, t);
16   }
17 }
```



P3: left 0 Ready: (empty)

t = 26: P3 finished its last 1 ms before the quantum ran out (`slice = 1`); it does not go back in the queue.

Step 9 / 10

VARIABLES

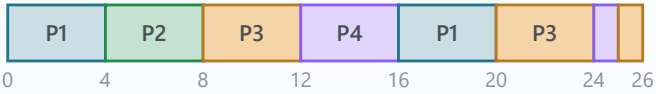
`t = 26` `s = P3` `slice = 1` `s->left = 0` `queue = []`

OUTPUT

```
P2 8
P1 20
P4 25
P3 26
```

Round Robin: Code Trace

```
1  /* p[] sorted by arrival time */
2  void rr(Proc p[], int n, int q) {
3    int t = 0, k = 0;      /* k: next arrival */
4    add_arrivals(p, n, &k, t);
▶    while (!empty() || k < n) {
6      if (empty()) {       /* CPU idle */
7        t = p[k].arrival;  /* next arrival */
8        add_arrivals(p, n, &k, t);
9      }
10     Proc *s = dequeue(); /* head */
11     int slice = min(q, s->left);
12     t += slice; s->left -= slice;
13     add_arrivals(p, n, &k, t);  /* first */
14     if (s->left > 0) enqueue(s); /* then */
15     else printf("%s %d\n", s->name, t);
16   }
17 }
```



Waiting: P1 12 · P2 3 · P3 15 · P4 17

average waiting 11.75 · turnaround 18.25 · response 4.5 ms

The queue is empty and no process is left to arrive (`k = n`), so the loop ends: 8 dispatches, **7 context switches**. New arrivals are queued before the preempted process because of the order of lines 13 and 14. Lines 6–9 never run in this example.

Step 10 / 10

VARIABLES

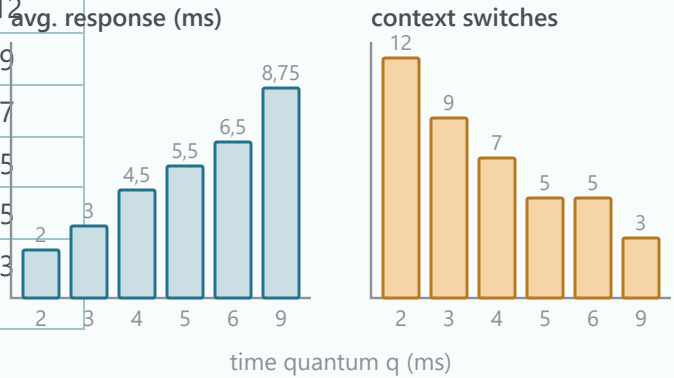
`t = 26` `queue = []`

OUTPUT

```
P2 8
P1 20
P4 25
P3 26
```

Choosing the Time Quantum: A Trade-off

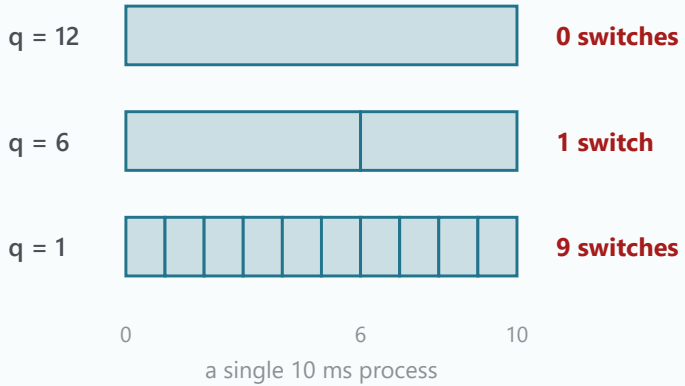
Quantum	Waiting	Turnaround	Response	Switches
q = 2	12,75	19,25	2	12
q = 3	13,5	20	3	9
q = 4	11,75	18,25	4,5	7
q = 5	11	17,5	5,5	5
q = 6	12	18,5	6,5	5
q ≥ 9 (FCFS)	8,75	15,25	8,75	3

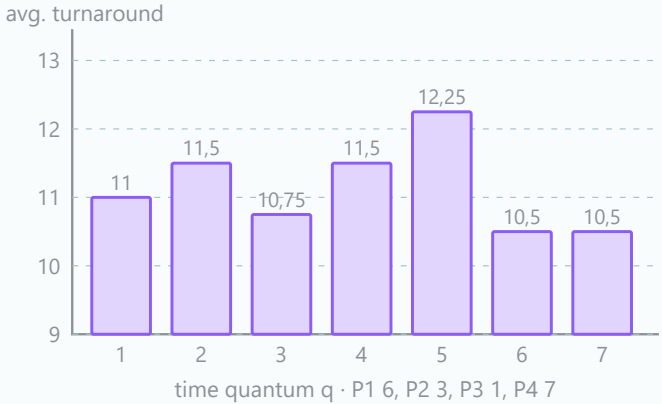


Averages in ms; switches = number of context switches.

- As q shrinks, **response** improves (2 ms at q = 2) but **context switches** increase (12 switches); on a real system each switch costs CPU time.
- Average waiting and turnaround do **not change monotonically** with q (q = 3 is worse than q = 2). Once q ≥ 9 (the longest burst), RR is exactly FCFS.

Time Quantum and Context Switches

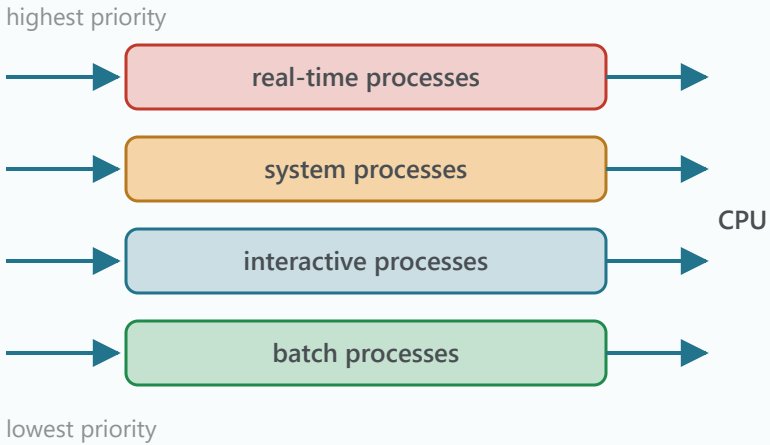




- Fraction lost = s / (q + s) (each process uses its full quantum, with a switch after each one; it differs for processes that finish early or block): with q = 4 ms and switch time s = 1 ms, **20%** of the CPU is wasted; at q = 100 ms it is ≈ 1%, but the system responds sluggishly.
- Today a switch takes a few **µs** and a typical quantum is 1–100 ms (Linux EEVDF base slice 0.75 ms × (1 + $\log_2$ CPUs), 3 ms on 8+ cores; on Windows client ≈ 2 clock ticks). Rule of thumb: ≈ 80% of CPU bursts should be shorter than q.
- Turnaround does not improve monotonically with q (right). With q = 4, the set P1 24, P2 3, P3 3 has average waiting 17 / 3 ≈ 5.67 ms (17 under FCFS).

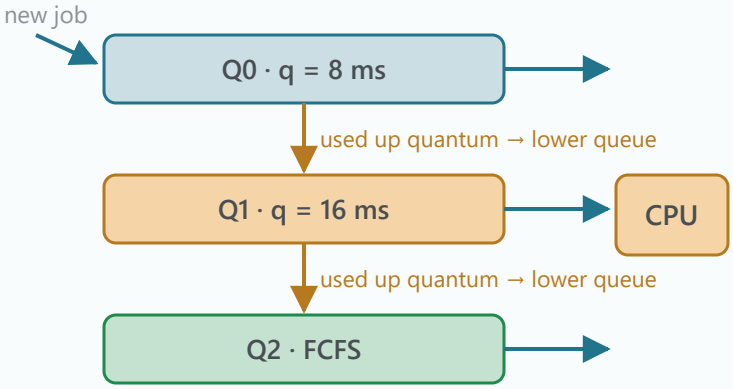
Multilevel Queue (MLQ)

- The ready queue is split into **classes**; each process is **permanently** assigned to one queue by type (e.g. foreground / background).
- Each queue has its own algorithm: RR (short q) for the foreground, FCFS for the background.
- Scheduling among queues: **fixed priority** (a lower queue runs only when the upper one is empty → starvation) or **time slicing** (e.g. 80% of the CPU to the foreground, 20% to the background).
- Low overhead but inflexible: a process cannot change queues.



A structure with fixed assignment is MLQ; one where processes change queues based on their behavior is the **feedback** queue, MLFQ.

Multilevel Feedback Queue (MLFQ)



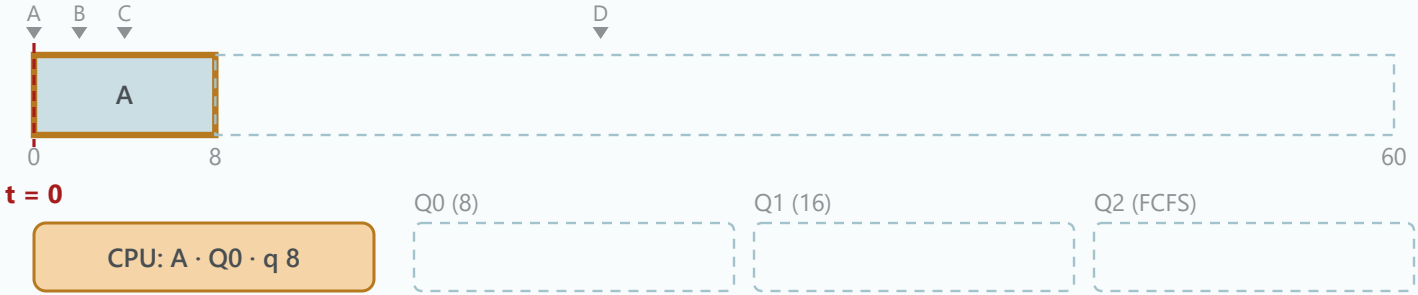
Q1 runs if Q0 is empty; Q2 if Q0 and Q1 are empty

1. Priority(A) > Priority(B) → A runs; if equal, RR.
2. A new job enters the **top** queue (assumed short at first).
3. A job that uses up its **allotment** at a level moves down one level, no matter how many times it gave up the CPU.
4. Every S time units all jobs move to the top queue (**priority boost**): this prevents starvation and gives a job whose behavior changed a new chance.

Parameters: number of queues, each queue's algorithm and quantum, the rules for moving down/up, the entry queue for new jobs. The rules follow OSTEP (Chapter 8); short jobs finish at the top, long CPU jobs sink down.

MLFQ Step by Step

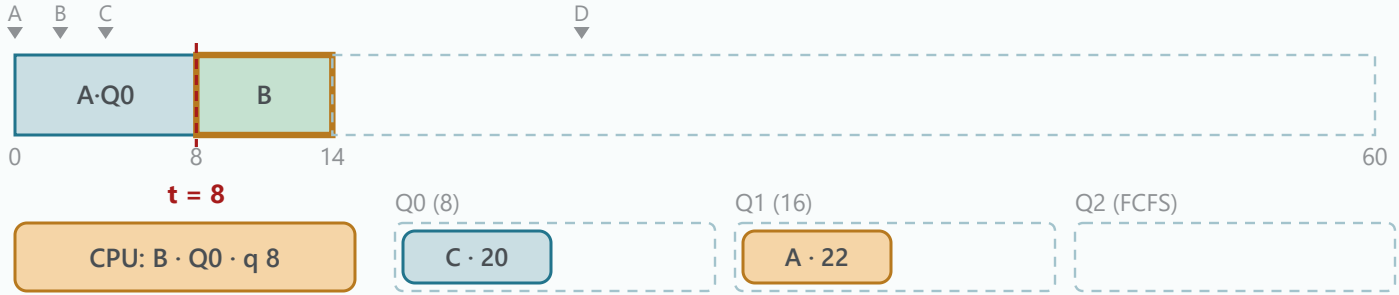
Three queues: Q0 (q = 8), Q1 (q = 16), Q2 (FCFS). Jobs: A (0, 30), B (2, 6), C (4, 20), D (25, 4). A preempted job returns to the head of its queue and completes its quantum.



t = 0: A arrives and enters Q0; it runs with an 8 ms quantum. B (t = 2) and C (t = 4) are added to Q0.

MLFQ Step by Step

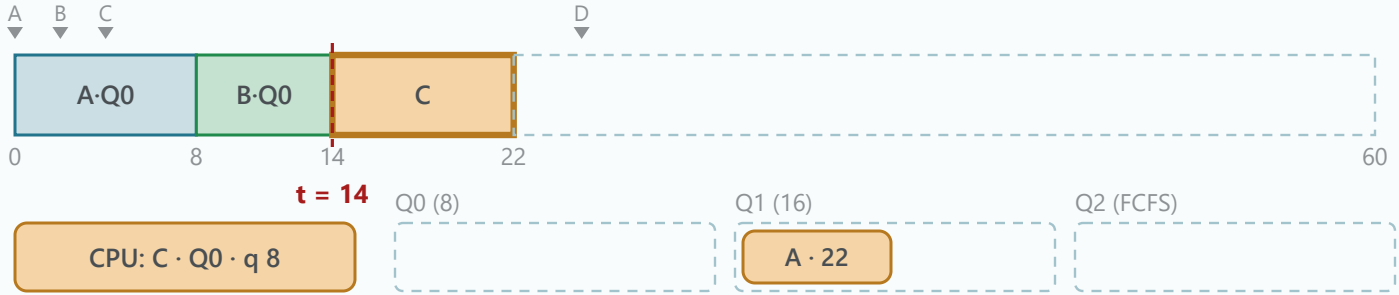
Three queues: Q0 (q = 8), Q1 (q = 16), Q2 (FCFS). Jobs: A (0, 30), B (2, 6), C (4, 20), D (25, 4). A preempted job returns to the head of its queue and completes its quantum.



t = 8: A used up its quantum → down to **Q1**. B, next in Q0, runs.

MLFQ Step by Step

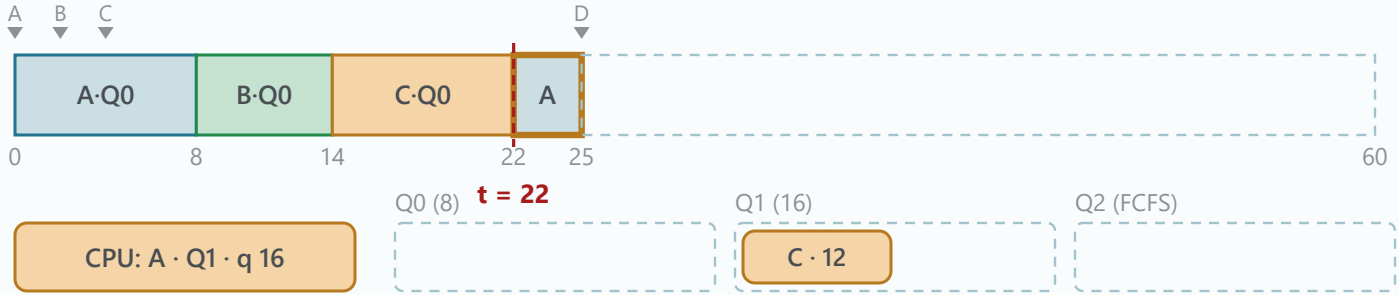
Three queues: Q0 (q = 8), Q1 (q = 16), Q2 (FCFS). Jobs: A (0, 30), B (2, 6), C (4, 20), D (25, 4). A preempted job returns to the head of its queue and completes its quantum.



t = 14: B finished in 6 ms (before its quantum ran out, in Q0). Now C runs 8 ms in Q0.

MLFQ Step by Step

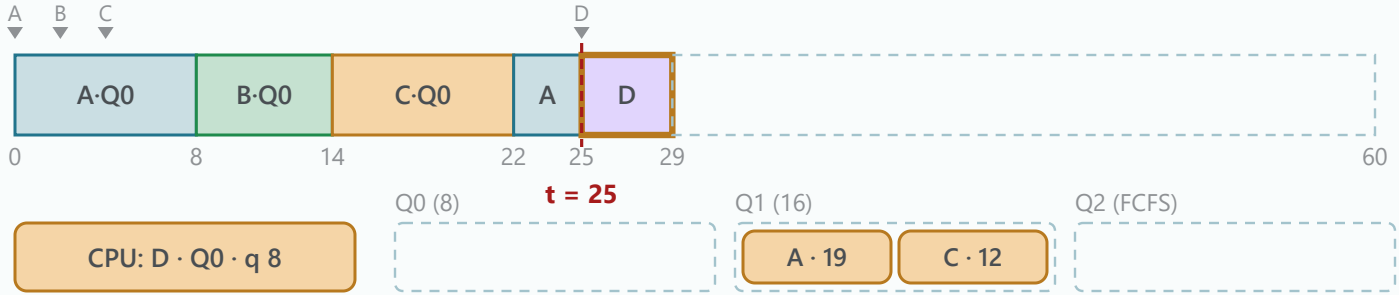
Three queues: Q0 (q = 8), Q1 (q = 16), Q2 (FCFS). Jobs: A (0, 30), B (2, 6), C (4, 20), D (25, 4). A preempted job returns to the head of its queue and completes its quantum.



t = 22: C also moves down to Q1. Q0 is empty → A at the head of Q1 runs with a 16 ms quantum.

MLFQ Step by Step

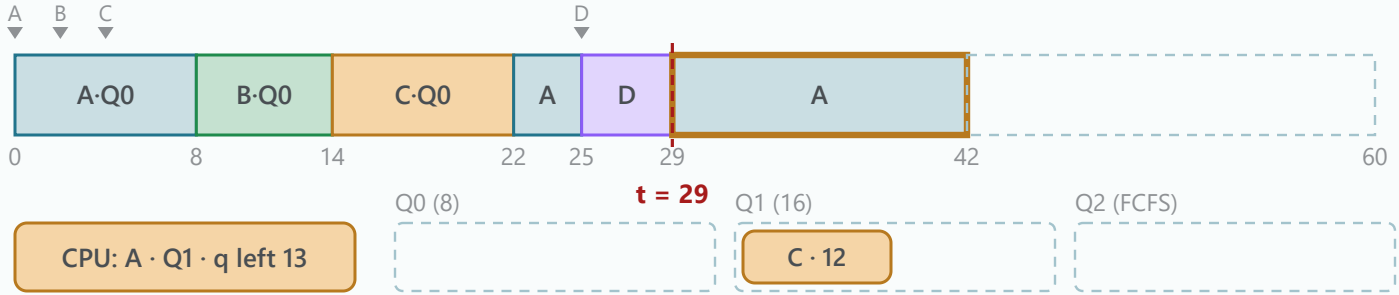
Three queues: Q0 (q = 8), Q1 (q = 16), Q2 (FCFS). Jobs: A (0, 30), B (2, 6), C (4, 20), D (25, 4). A preempted job returns to the head of its queue and completes its quantum.



t = 25: D arrives and enters Q0: since it is the upper queue, **it preempts A**. A returns to the head of Q1 (13 ms of its quantum left).

MLFQ Step by Step

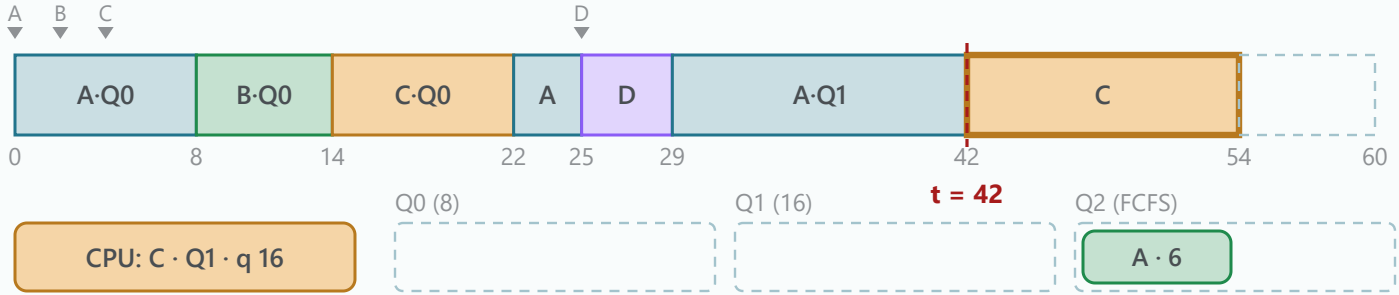
Three queues: Q0 (q = 8), Q1 (q = 16), Q2 (FCFS). Jobs: A (0, 30), B (2, 6), C (4, 20), D (25, 4). A preempted job returns to the head of its queue and completes its quantum.



t = 29: D finished in 4 ms (response 0, waiting 0). A uses the remaining 13 ms of its Q1 quantum.

MLFQ Step by Step

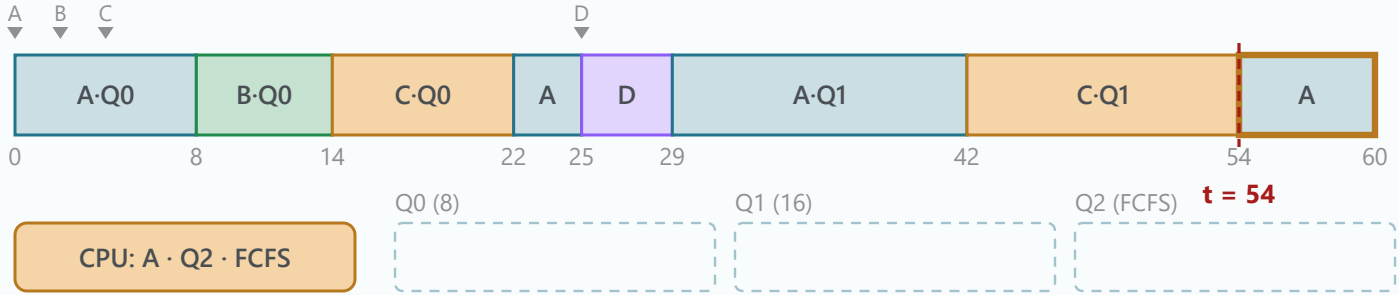
Three queues: Q0 (q = 8), Q1 (q = 16), Q2 (FCFS). Jobs: A (0, 30), B (2, 6), C (4, 20), D (25, 4). A preempted job returns to the head of its queue and completes its quantum.



t = 42: A used up its Q1 quantum too → **Q2**. C runs in Q1; 12 ms is enough.

MLFQ Step by Step

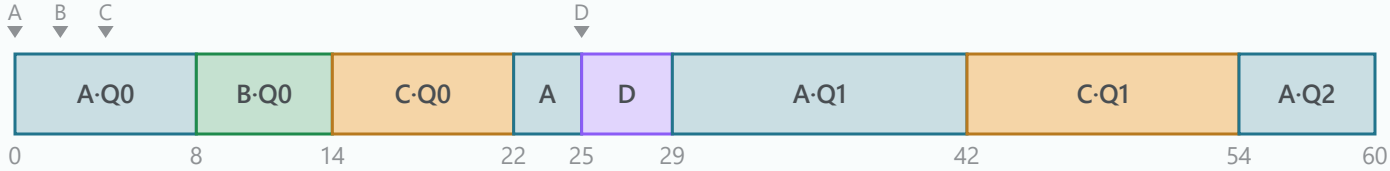
Three queues: Q0 (q = 8), Q1 (q = 16), Q2 (FCFS). Jobs: A (0, 30), B (2, 6), C (4, 20), D (25, 4). A preempted job returns to the head of its queue and completes its quantum.



t = 54: Q0 and Q1 are empty; A runs its last 6 ms FCFS in Q2 and finishes at t = 60.

MLFQ Step by Step

Three queues: Q0 (q = 8), Q1 (q = 16), Q2 (FCFS). Jobs: A (0, 30), B (2, 6), C (4, 20), D (25, 4). A preempted job returns to the head of its queue and completes its quantum.



Job	Arrival	CPU	Completion	Turnaround	Waiting	Response
A	0	30	60	60	30	0
B	2	6	14	12	6	6
C	4	20	54	50	30	10
D	25	4	29	4	0	0
Avg.				**31,5**	**16,5**	**4**

The short B and D finish quickly in the top queue (response 6 and 0); the long A sinks down. Average response **4 ms**.

MLFQ in Practice

Gaming the scheduler

A job that does I/O just before its quantum ends can monopolize the CPU under the "blocked jobs stay on top" rule. The fix: count the total usage at a level (rule 3).

Starvation and changing behavior

Many interactive jobs starve long ones; and a long job that later becomes interactive goes unnoticed. The fix: periodic priority boost (rule 4).

History: CTSS (1962)

Priority classes ran for 1, 2, 4, 8 quanta; a job that used up its quanta moved down a class. Pressing Enter at the terminal moved the process to the top class; users noticed and sped up their jobs by pressing Enter.

Today

The Solaris TS class (table-driven quanta and priorities), FreeBSD ULE, Windows dynamic priorities and macOS use the MLFQ idea. Since 2.6.23 Linux uses fair sharing (CFS, then EEVDF) instead of priority heuristics.

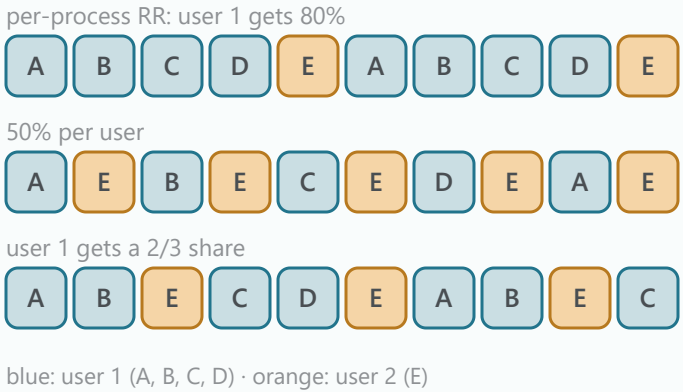


Guaranteed and Fair-Share Scheduling

Guaranteed scheduling: with n users (or processes), each is promised 1/n of the CPU. For each, **ratio = CPU received / CPU entitled** is computed, and the lowest ratio runs. One at 0.5 (half its share) goes ahead; one at 2.0 waits.

Fair-share scheduling: shares go not to processes but to **users or groups**; a user who starts many processes cannot get more CPU.

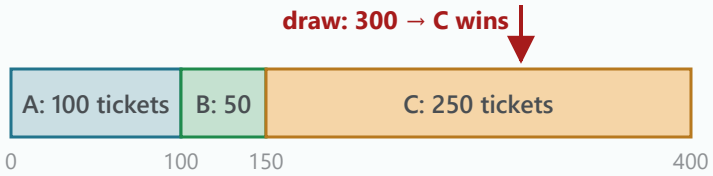
Today: group scheduling with Linux **cgroup v2** `cpu.weight`; systemd puts each user in a `user-UID.slice`, and Kubernetes CPU requests are turned into weights too.



Guaranteed scheduling in Tanenbaum's sense is the share guarantee above, not running to a predetermined timetable.

PROPORTIONAL SHARE

Lottery Scheduling



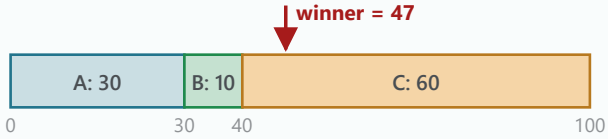
Probabilities: A 100/400 = 25% · B 12.5% · C 62.5%

- Each process gets **tickets**; in every time slice a random ticket is drawn and its holder runs. In the long run, share = tickets / total tickets.
- **Simple to implement:** a random number + a running total over a list (Waldspurger and Weihl, 1994).
- A new process has a chance as soon as it gets tickets: **responsive**; priority means “more tickets”.
- **Ticket transfer:** a client waiting on a server hands its tickets to the server. **Ticket currency** and, in a trusted environment, **inflation**.
- It is probabilistic: it can be unfair in the short run (two jobs with equal tickets may split 10 slices 7–3). Its deterministic counterpart is **stride** scheduling.

Lottery scheduling is one of the simplest algorithms; priority is expressed as a number of tickets.

Lottery Scheduling: Code Trace

```
1  /* one draw per time slice */
2  int counter = 0;
3  int winner = getrandom(0, totaltickets);
4  node_t *current = head;
5  while (current) {
6    counter = counter + current->tickets;
7    if (counter > winner)
8      break;          /* found the winner */
9    current = current->next;
10 }
11 /* current: process that gets the CPU */
12 run(current);
```



Ticket list A 30, B 10, C 60 (100 in total). The number drawn is **47**; `counter` holds the running total.

Step 1 / 10

VARIABLES

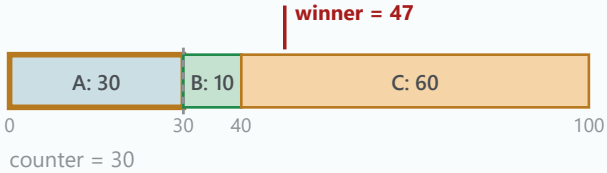
`winner = 47` `counter = 0`

`current = A` `ran = –`

The code is the decision loop from OSTEP Chapter 9; the drawn numbers are fixed for the example.

Lottery Scheduling: Code Trace

```
1  /* one draw per time slice */
2  int counter = 0;
3  int winner = getrandom(0, totaltickets);
4  node_t *current = head;
5  while (current) {
▶    counter = counter + current->tickets;
▶    if (counter > winner)
8      break;          /* found the winner */
▶    current = current->next;
10 }
11 /* current: process that gets the CPU */
12 run(current);
```



A: counter = 30, not above 47 → the drawn ticket is not in A's range; next node.

Step 2 / 10

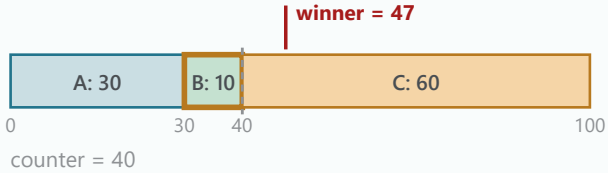
VARIABLES

`winner = 47` `counter = 30` `current = B` `ran = –`

The code is the decision loop from OSTEP Chapter 9; the drawn numbers are fixed for the example.

Lottery Scheduling: Code Trace

```
1  /* one draw per time slice */
2  int counter = 0;
3  int winner = getrandom(0, totaltickets);
4  node_t *current = head;
5  while (current) {
▶    counter = counter + current->tickets;
▶    if (counter > winner)
8      break;          /* found the winner */
▶    current = current->next;
10 }
11 /* current: process that gets the CPU */
12 run(current);
```



B: counter = 40, not above 47 → the drawn ticket is not in B's range; next node.

Step 3 / 10

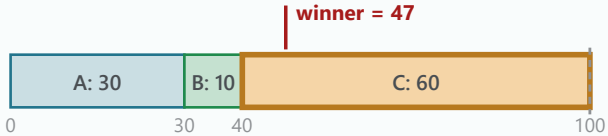
VARIABLES

`winner = 47` `counter = 40` `current = C` `ran = –`

The code is the decision loop from OSTEP Chapter 9; the drawn numbers are fixed for the example.

Lottery Scheduling: Code Trace

```
1  /* one draw per time slice */
2  int counter = 0;
3  int winner = getrandom(0, totaltickets);
4  node_t *current = head;
5  while (current) {
▶    counter = counter + current->tickets;
▶    if (counter > winner)
▶      break;          /* found the winner */
9    current = current->next;
10 }
11 /* current: process that gets the CPU */
12 run(current);
```



counter = 100 · winner: C

C: counter = 100 > 47 → **C wins**; the loop breaks.

Step 4 / 10

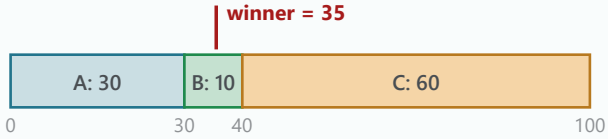
VARIABLES

`winner = 47` `counter = 100` `current = C` `ran = –`

The code is the decision loop from OSTEP Chapter 9; the drawn numbers are fixed for the example.

Lottery Scheduling: Code Trace

```
1  /* one draw per time slice */
2  int counter = 0;
3  int winner = getrandom(0, totaltickets);
4  node_t *current = head;
5  while (current) {
6    counter = counter + current->tickets;
7    if (counter > winner)
8      break;          /* found the winner */
9    current = current->next;
10 }
11 /* current: process that gets the CPU */
12 run(current);
```



Next slice: a new draw, **35**; the counter starts from zero and the list is scanned from the head again.

Step 5 / 10

VARIABLES

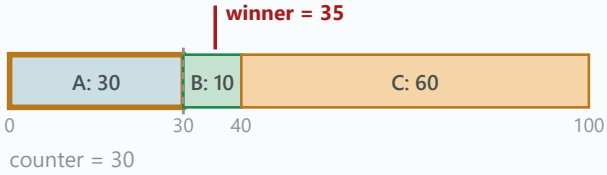
`winner = 35` `counter = 0`

`current = A` `ran = C`

The code is the decision loop from OSTEP Chapter 9; the drawn numbers are fixed for the example.

Lottery Scheduling: Code Trace

```
1  /* one draw per time slice */
2  int counter = 0;
3  int winner = getrandom(0, totaltickets);
4  node_t *current = head;
5  while (current) {
▶    counter = counter + current->tickets;
▶    if (counter > winner)
8      break;          /* found the winner */
▶    current = current->next;
10 }
11 /* current: process that gets the CPU */
12 run(current);
```



counter = 30

A: counter = 30, not above 35 → the drawn ticket is not in A's range; next node.

Step 6 / 10

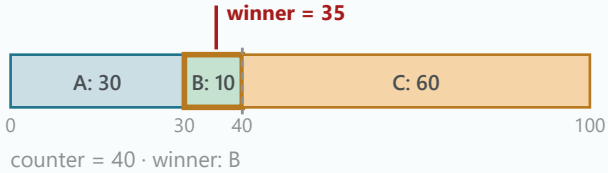
VARIABLES

`winner = 35` `counter = 30` `current = B` `ran = C`

The code is the decision loop from OSTEP Chapter 9; the drawn numbers are fixed for the example.

Lottery Scheduling: Code Trace

```
1  /* one draw per time slice */
2  int counter = 0;
3  int winner = getrandom(0, totaltickets);
4  node_t *current = head;
5  while (current) {
▶    counter = counter + current->tickets;
▶    if (counter > winner)
▶      break;          /* found the winner */
9    current = current->next;
10 }
11 /* current: process that gets the CPU */
12 run(current);
```



counter = 40 · winner: B

B: counter = 40 > 35 → **B wins**; the loop breaks.

Step 7 / 10

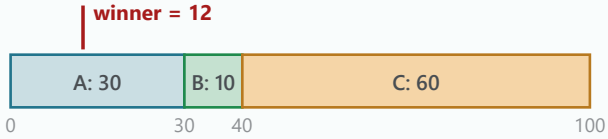
VARIABLES

`winner = 35` `counter = 40` `current = B` `ran = C`

The code is the decision loop from OSTEP Chapter 9; the drawn numbers are fixed for the example.

Lottery Scheduling: Code Trace

```
1  /* one draw per time slice */
2  int counter = 0;
3  int winner = getrandom(0, totaltickets);
4  node_t *current = head;
5  while (current) {
6    counter = counter + current->tickets;
7    if (counter > winner)
8      break;          /* found the winner */
9    current = current->next;
10 }
11 /* current: process that gets the CPU */
12 run(current);
```



Next slice: a new draw, **12**; the counter starts from zero and the list is scanned from the head again.

Step 8 / 10

VARIABLES

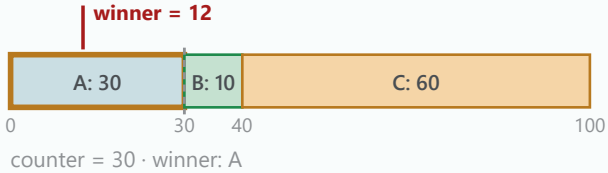
`winner = 12` `counter = 0`

`current = A` `ran = C B`

The code is the decision loop from OSTEP Chapter 9; the drawn numbers are fixed for the example.

Lottery Scheduling: Code Trace

```
1  /* one draw per time slice */
2  int counter = 0;
3  int winner = getrandom(0, totaltickets);
4  node_t *current = head;
5  while (current) {
▶    counter = counter + current->tickets;
▶    if (counter > winner)
▶      break;          /* found the winner */
9    current = current->next;
10 }
11 /* current: process that gets the CPU */
12 run(current);
```



counter = 30 · winner: A

A: counter = 30 > 12 → **A wins**; the loop breaks.

Step 9 / 10

VARIABLES

`winner = 12` `counter = 30` `current = A` `ran = C B`

The code is the decision loop from OSTEP Chapter 9; the drawn numbers are fixed for the example.

Lottery Scheduling: Code Trace

```
1  /* one draw per time slice */
2  int counter = 0;
3  int winner = getrandom(0, totaltickets);
4  node_t *current = head;
5  while (current) {
6    counter = counter + current->tickets;
7    if (counter > winner)
8      break;          /* found the winner */
9    current = current->next;
10 }
11 /* current: process that gets the CPU */
▶  run(current);
```

Draws: 47 → C · 35 → B · 12 → A

Expected share: A 30% · B 10% · C 60%

In three slices each ran once:

short-run share is not proportional to tickets

In three slices A, B and C each ran once; the share approaches 30 : 10 : 60 only **in the long run**. Sorting the list by tickets in decreasing order shortens the scan.

Step 10 / 10

VARIABLES

`ran = C B A`

The code is the decision loop from OSTEP Chapter 9; the drawn numbers are fixed for the example.

PROPORTIONAL SHARE

Stride Scheduling Step by Step

Stride = large number / tickets (10,000 / tickets): A 100, B 200, C 40. In each slice the process with the smallest **pass** value runs and adds its stride to its pass; ties go in the order A, B, C.



Step 1: pass values A 0, B 0, C 0 → smallest **A** runs; pass(A) += 100.

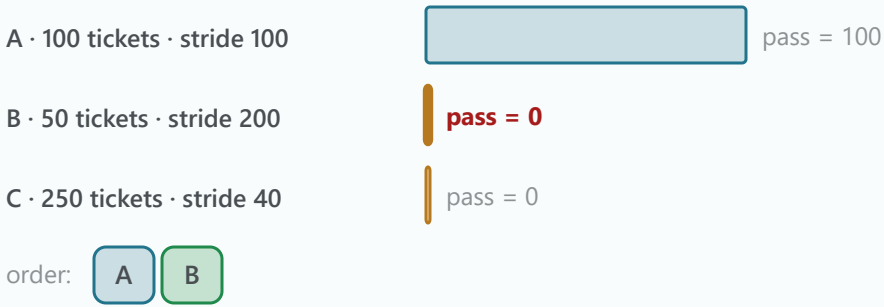
Step 1 / 9

Stride is the deterministic form of lottery (Waldspurger, 1995). Its drawback is global state: a newcomer with pass 0 would monopolize the CPU, so it starts at the current minimum pass. The vruntime of Linux CFS is a very close idea.

PROPORTIONAL SHARE

Stride Scheduling Step by Step

Stride = large number / tickets (10,000 / tickets): A 100, B 200, C 40. In each slice the process with the smallest **pass** value runs and adds its stride to its pass; ties go in the order A, B, C.



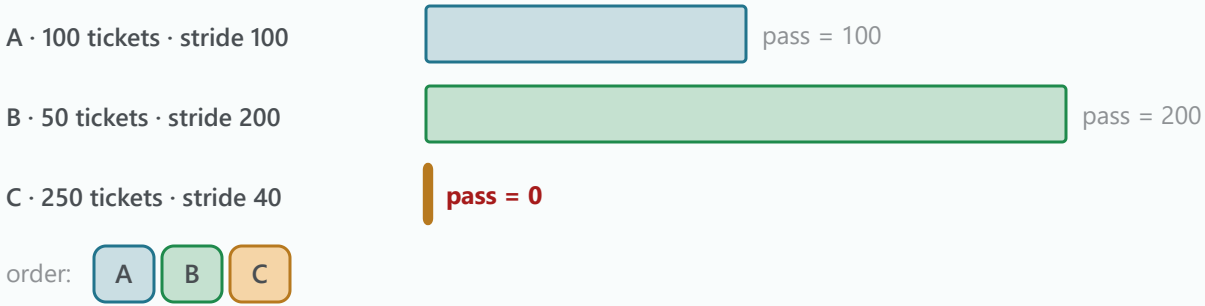
Step 2: pass values A 100, B 0, C 0 → smallest **B** runs; pass(B) += 200.

Stride is the deterministic form of lottery (Waldspurger, 1995). Its drawback is global state: a newcomer with pass 0 would monopolize the CPU, so it starts at the current minimum pass. The vruntime of Linux CFS is a very close idea.

PROPORTIONAL SHARE

Stride Scheduling Step by Step

Stride = large number / tickets (10,000 / tickets): A 100, B 200, C 40. In each slice the process with the smallest **pass** value runs and adds its stride to its pass; ties go in the order A, B, C.



Step 3: pass values A 100, B 200, C 0 → smallest **C** runs; pass(C) += 40.

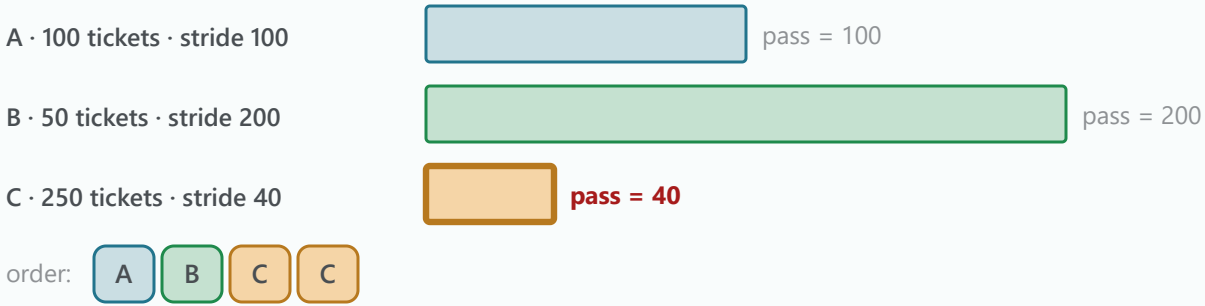
Step 3 / 9

Stride is the deterministic form of lottery (Waldspurger, 1995). Its drawback is global state: a newcomer with pass 0 would monopolize the CPU, so it starts at the current minimum pass. The vruntime of Linux CFS is a very close idea.

PROPORTIONAL SHARE

Stride Scheduling Step by Step

Stride = large number / tickets (10,000 / tickets): A 100, B 200, C 40. In each slice the process with the smallest **pass** value runs and adds its stride to its pass; ties go in the order A, B, C.



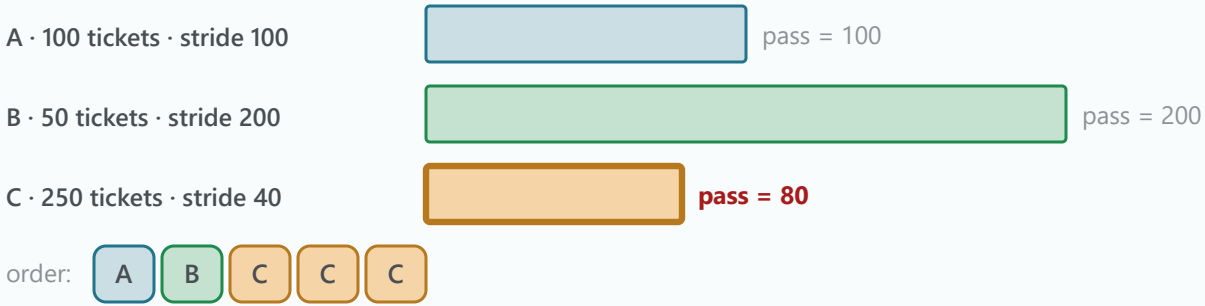
Step 4: pass values A 100, B 200, C 40 → smallest **C** runs; pass(C) += 40.

Stride is the deterministic form of lottery (Waldspurger, 1995). Its drawback is global state: a newcomer with pass 0 would monopolize the CPU, so it starts at the current minimum pass. The vruntime of Linux CFS is a very close idea.

PROPORTIONAL SHARE

Stride Scheduling Step by Step

Stride = large number / tickets (10,000 / tickets): A 100, B 200, C 40. In each slice the process with the smallest **pass** value runs and adds its stride to its pass; ties go in the order A, B, C.



Step 5: pass values A 100, B 200, C 80 → smallest **C** runs; pass(C) += 40.

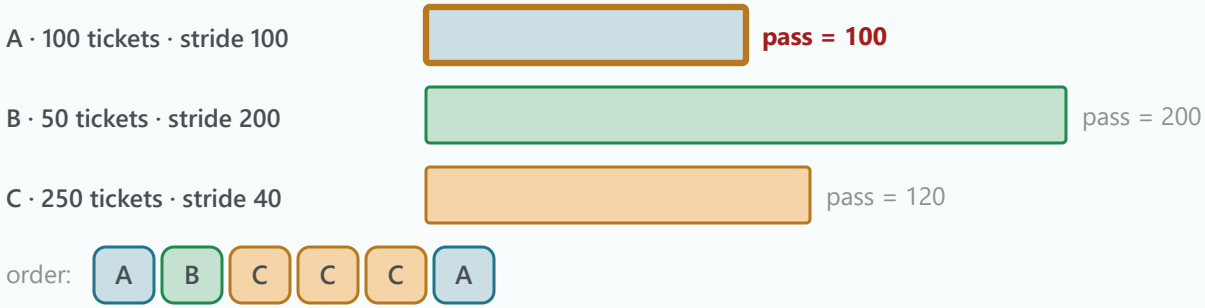
Step 5 / 9

Stride is the deterministic form of lottery (Waldspurger, 1995). Its drawback is global state: a newcomer with pass 0 would monopolize the CPU, so it starts at the current minimum pass. The vruntime of Linux CFS is a very close idea.

PROPORTIONAL SHARE

Stride Scheduling Step by Step

Stride = large number / tickets (10,000 / tickets): A 100, B 200, C 40. In each slice the process with the smallest **pass** value runs and adds its stride to its pass; ties go in the order A, B, C.



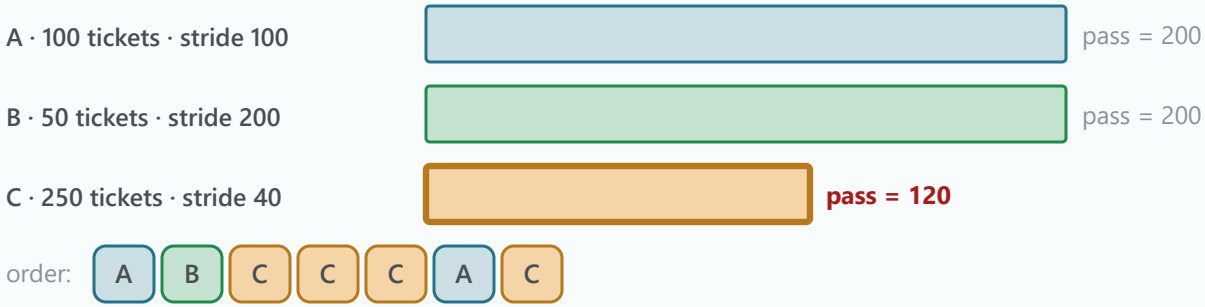
Step 6: pass values A 100, B 200, C 120 → smallest **A** runs; pass(A) += 100.

Step 6 / 9

Stride is the deterministic form of lottery (Waldspurger, 1995). Its drawback is global state: a newcomer with pass 0 would monopolize the CPU, so it starts at the current minimum pass. The vruntime of Linux CFS is a very close idea.

Stride Scheduling Step by Step

Stride = large number / tickets (10,000 / tickets): A 100, B 200, C 40. In each slice the process with the smallest **pass** value runs and adds its stride to its pass; ties go in the order A, B, C.

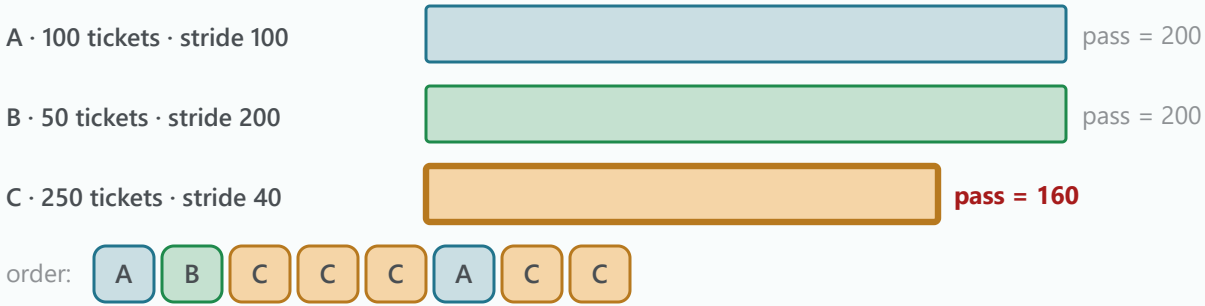


Step 7: pass values A 200, B 200, C 120 → smallest **C** runs; pass(C) += 40.

Stride is the deterministic form of lottery (Waldspurger, 1995). Its drawback is global state: a newcomer with pass 0 would monopolize the CPU, so it starts at the current minimum pass. The vruntime of Linux CFS is a very close idea.

Stride Scheduling Step by Step

Stride = large number / tickets (10,000 / tickets): A 100, B 200, C 40. In each slice the process with the smallest **pass** value runs and adds its stride to its pass; ties go in the order A, B, C.

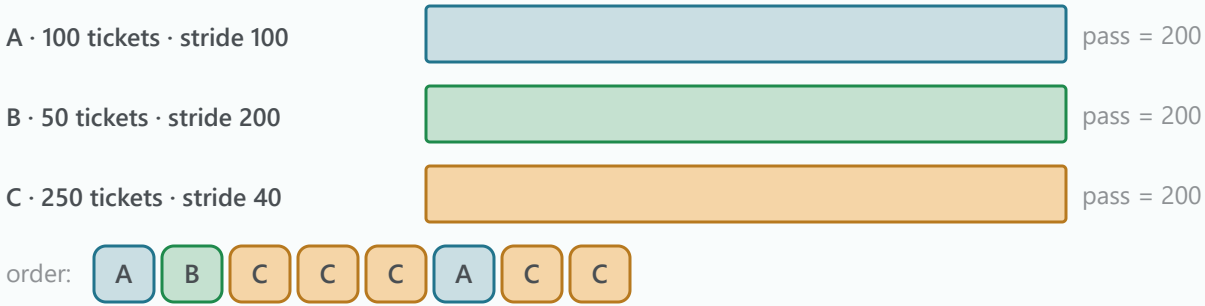


Step 8: pass values A 200, B 200, C 160 → smallest **C** runs; pass(C) += 40.

Stride is the deterministic form of lottery (Waldspurger, 1995). Its drawback is global state: a newcomer with pass 0 would monopolize the CPU, so it starts at the current minimum pass. The vruntime of Linux CFS is a very close idea.

Stride Scheduling Step by Step

Stride = large number / tickets (10,000 / tickets): A 100, B 200, C 40. In each slice the process with the smallest **pass** value runs and adds its stride to its pass; ties go in the order A, B, C.

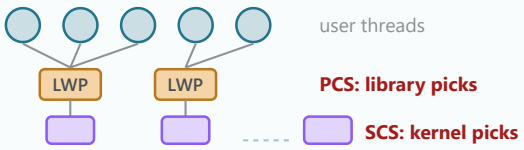


In 8 slices A ran 2 times, B once, C 5 times: exactly 100 : 50 : 250 = 2 : 1 : 5. With a lottery this ratio holds only in the long run.

Stride is the deterministic form of lottery (Waldspurger, 1995). Its drawback is global state: a newcomer with pass 0 would monopolize the CPU, so it starts at the current minimum pass. The vruntime of Linux CFS is a very close idea.

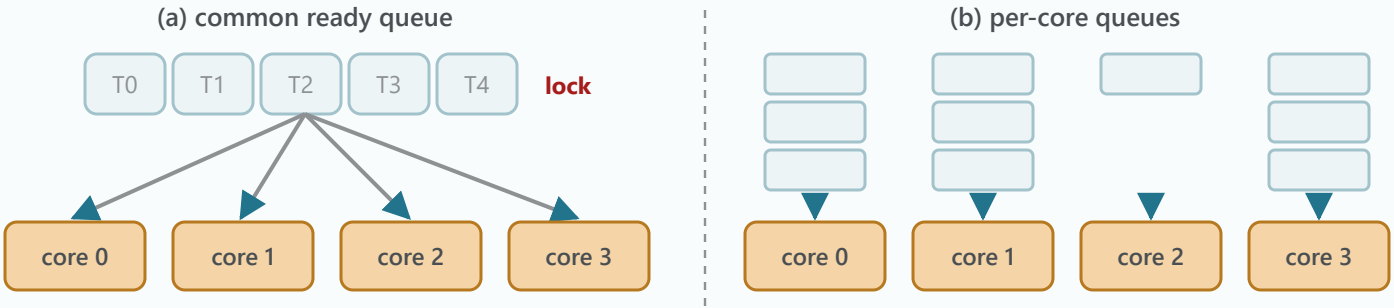
Thread Scheduling: PCS and SCS

- The kernel schedules **kernel threads**, not processes. A library manages user threads.
- **PCS** (process-contention scope): in the N:1 and M:N models the library has user threads compete for the LWPs within their process.
- **SCS** (system-contention scope): the kernel chooses among all threads in the system. Linux, Windows and macOS, with the 1:1 model, use only SCS.
- A current PCS example: Java 21 **virtual threads** and Go goroutines are mapped onto a few carrier threads by the run-time's own scheduler.



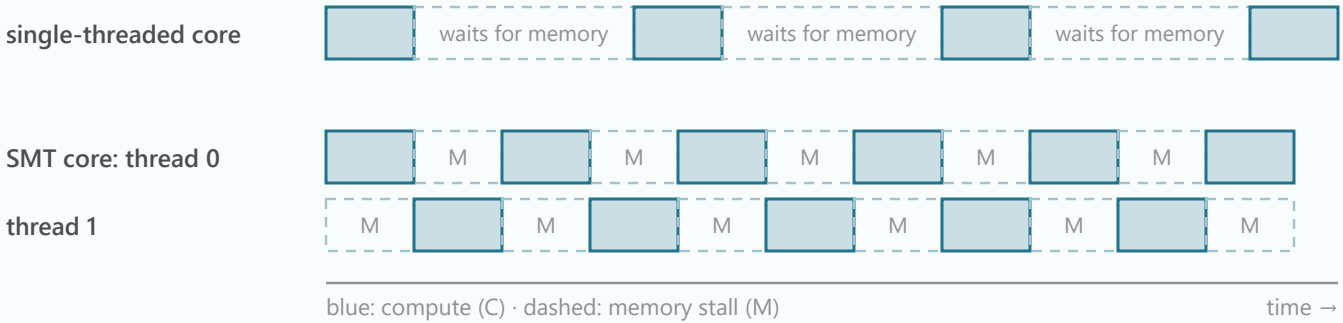

```c
pthread_attr_t attr;
int scope;
pthread_attr_init(&attr);
pthread_attr_getscope(&attr, &scope);
if (scope == PTHREAD_SCOPE_PROCESS)
  printf("PCS\n");
/* Linux, macOS: SYSTEM only */
pthread_attr_setscope(&attr,
                      PTHREAD_SCOPE_SYSTEM);
pthread_create(&tid, &attr, runner, NULL);
```

Approaches to Multiprocessor Scheduling



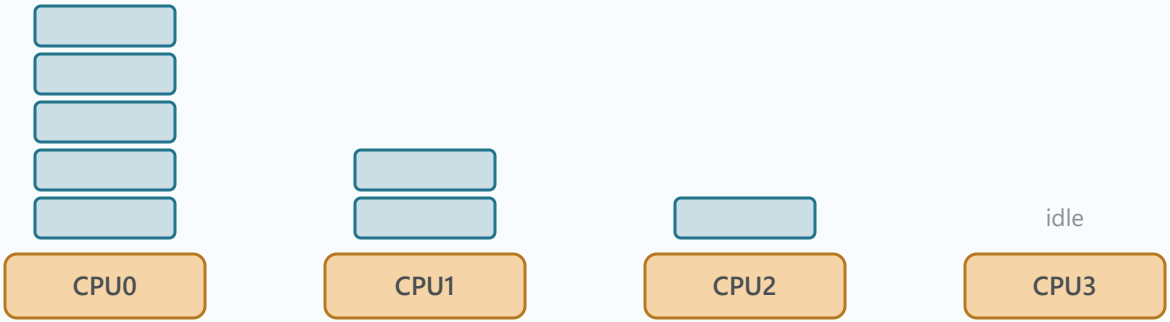
- **Asymmetric** (AMP): a single master processor does all scheduling and I/O; simple but a bottleneck.
- **Symmetric** (SMP): each processor schedules itself; most modern general-purpose systems. AMP-like arrangements are still used in embedded/heterogeneous systems.
- (a) Common queue: the load balances itself, but there is lock contention and poor cache affinity.
- (b) Per-core queues (**two-level scheduling**): a job is first assigned to a core, then runs from that core's queue. Locking is almost nil and caches stay warm. Linux, Windows and FreeBSD use this.
- The price: queues can become unbalanced → **load balancing** is needed.

Multicore and Hardware Threads



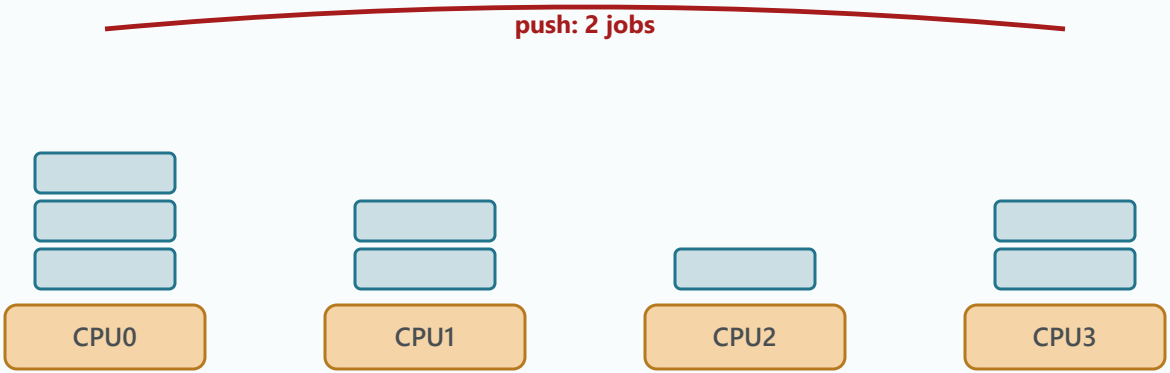
- **Memory stall**: on cache misses a core can spend up to 50% of its time waiting for data.
- **SMT** (Intel Hyper-Threading, AMD SMT): 2 hardware threads per core; one runs while the other waits. The OS sees each hardware thread as a separate **logical CPU**.
- Two-level scheduling: the OS chooses which software thread runs on which logical CPU; the core's hardware chooses which hardware thread executes at the moment.
- Sibling logical CPUs share execution units: putting two heavy jobs on the same core while a physical core is idle is a bad decision.

Load Balancing



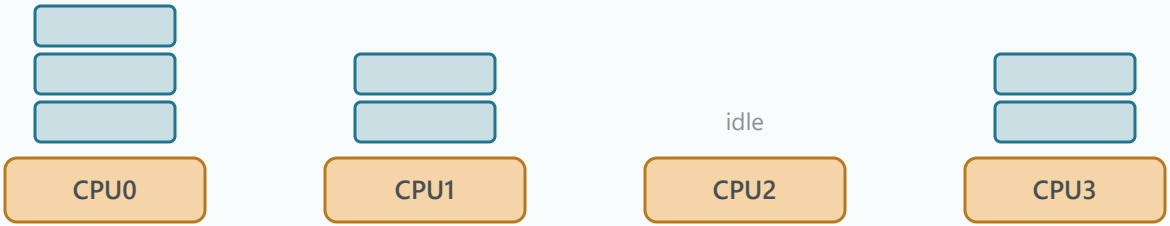
Start: CPU0's queue has 5 jobs, CPU3 is idle. With per-core queues the imbalance does not fix itself.

Load Balancing



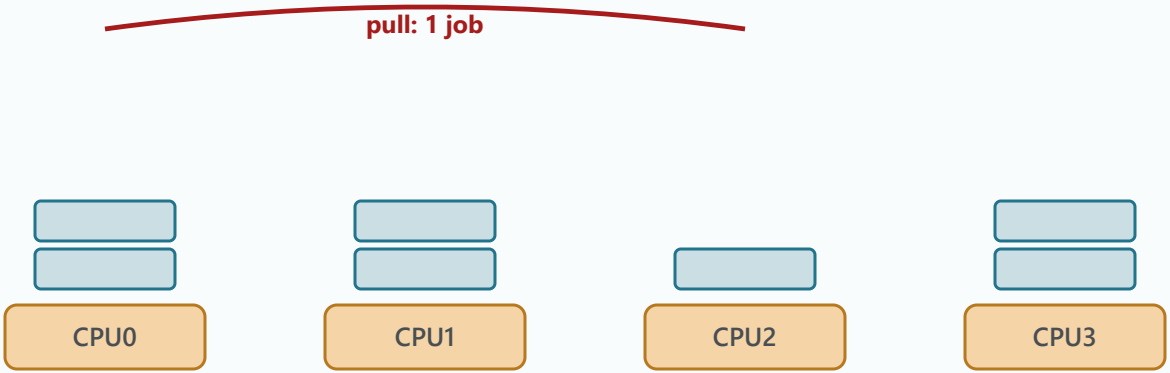
Push migration: a periodic balancer compares loads and pushes 2 jobs from the overloaded CPU0 to CPU3.

Load Balancing



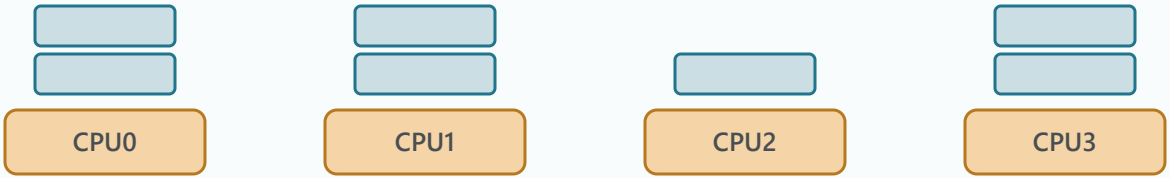
The only job on CPU2 finished; CPU2 is now idle.

Load Balancing



Pull migration: the idle CPU2 steals a job from the busiest queue. Linux uses both.

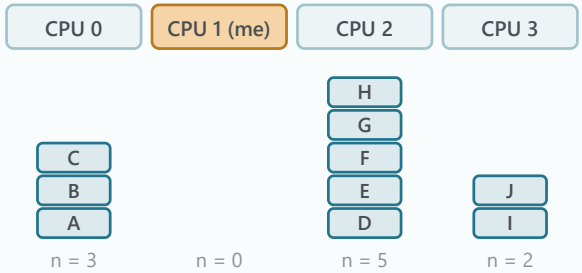
Load Balancing



Balance is restored, but the moved job's cache is **cold** on the new CPU. Linux balances first within SMT siblings and shared caches, and less often across NUMA nodes (scheduling domains).

Load Balancing: Code Trace

```
▶ void balance(int me) {    /* idle CPU */
▶   int max = me;
3   for (int c = 0; c < NCPU; c++)
4     if (q[c].n > q[max].n) /* busier */
5       max = c;
6   int t = (q[max].n - q[me].n) / 2;
7   for (int k = 0; k < t; k++)
8     migrate(&q[max], &q[me]); /* from tail */
9 }
```



CPU 1's queue is empty. Pull balancing: `balance(1)`. Initially max = me = 1.

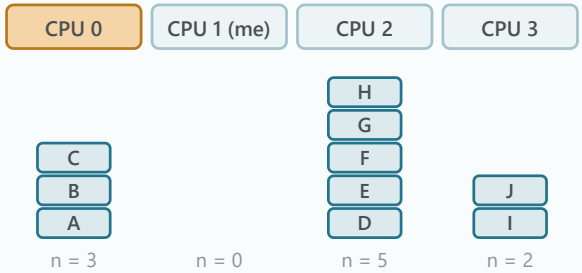
Step 1 / 8

VARIABLES

`max = 1` `c = –` `t = –` `k = –`

Load Balancing: Code Trace

```
1 void balance(int me) {    /* idle CPU */
2   int max = me;
▶   for (int c = 0; c < NCPU; c++)
▶     if (q[c].n > q[max].n) /* busier */
▶       max = c;
6   int t = (q[max].n - q[me].n) / 2;
7   for (int k = 0; k < t; k++)
8     migrate(&q[max], &q[me]); /* from tail */
9 }
```



c = 0: n = 3 > the busiest so far → max = 0.

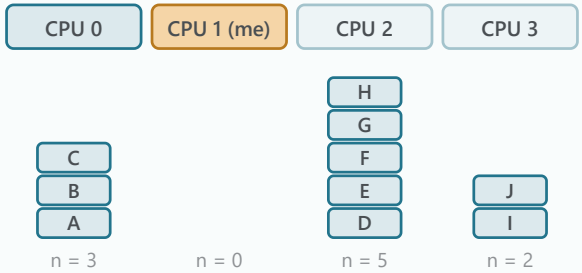
Step 2 / 8

VARIABLES

`max = 0` `c = 0` `t = –` `k = –`

Load Balancing: Code Trace

```
1 void balance(int me) {    /* idle CPU */
2   int max = me;
▶   for (int c = 0; c < NCPU; c++)
▶     if (q[c].n > q[max].n) /* busier */
5       max = c;
6   int t = (q[max].n - q[me].n) / 2;
7   for (int k = 0; k < t; k++)
8     migrate(&q[max], &q[me]); /* from tail */
9 }
```



c = 1: n = 0, not busier.

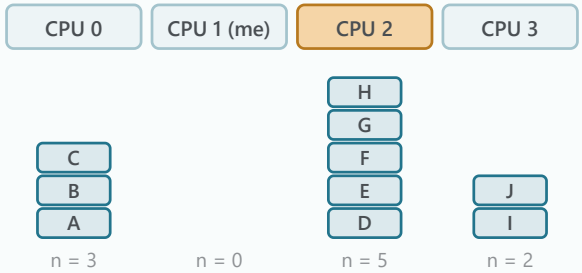
Step 3 / 8

VARIABLES

`max = 0` `c = 1` `t = –` `k = –`

Load Balancing: Code Trace

```
1 void balance(int me) {    /* idle CPU */
2   int max = me;
▶   for (int c = 0; c < NCPU; c++)
▶     if (q[c].n > q[max].n) /* busier */
▶       max = c;
6   int t = (q[max].n - q[me].n) / 2;
7   for (int k = 0; k < t; k++)
8     migrate(&q[max], &q[me]); /* from tail */
9 }
```



c = 2: n = 5 > the busiest so far → max = 2.

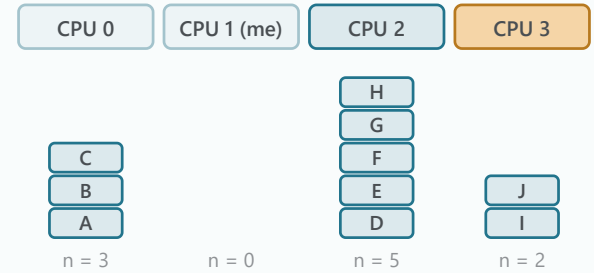
Step 4 / 8

VARIABLES

`max = 2` `c = 2` `t = –` `k = –`

Load Balancing: Code Trace

```
1 void balance(int me) {    /* idle CPU */
2   int max = me;
▶   for (int c = 0; c < NCPU; c++)
▶     if (q[c].n > q[max].n) /* busier */
5       max = c;
6   int t = (q[max].n - q[me].n) / 2;
7   for (int k = 0; k < t; k++)
8     migrate(&q[max], &q[me]); /* from tail */
9 }
```



c = 3: n = 2, not busier.

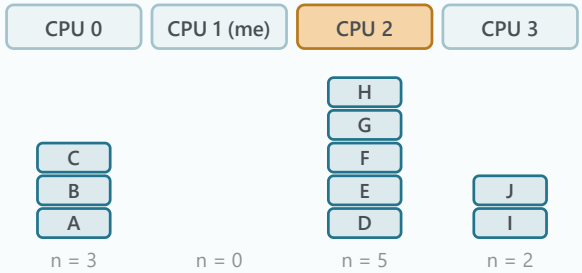
Step 5 / 8

VARIABLES

`max = 2` `c = 3` `t = –` `k = –`

Load Balancing: Code Trace

```
1 void balance(int me) {    /* idle CPU */
2   int max = me;
3   for (int c = 0; c < NCPU; c++)
4     if (q[c].n > q[max].n) /* busier */
5       max = c;
▶ int t = (q[max].n - q[me].n) / 2;
7   for (int k = 0; k < t; k++)
8     migrate(&q[max], &q[me]); /* from tail */
9 }
```



t = (5 − 0) / 2 = 2: moving half the difference brings the two queues close to equal (CPU 2: 3, CPU 1: 2).

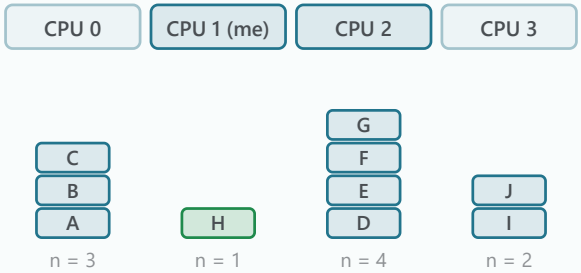
Step 6 / 8

VARIABLES

`max = 2` `c = 3` `t = 2` `k = –`

Load Balancing: Code Trace

```
1 void balance(int me) {    /* idle CPU */
2   int max = me;
3   for (int c = 0; c < NCPU; c++)
4     if (q[c].n > q[max].n) /* busier */
5       max = c;
6   int t = (q[max].n - q[me].n) / 2;
▶   for (int k = 0; k < t; k++)
▶     migrate(&q[max], &q[me]); /* from tail */
9 }
```



k = 0: **H**, at the tail of CPU 2's queue, migrates to CPU 1. Jobs at the tail are not the longest-waiting but the most recent arrivals: those with the coldest caches.

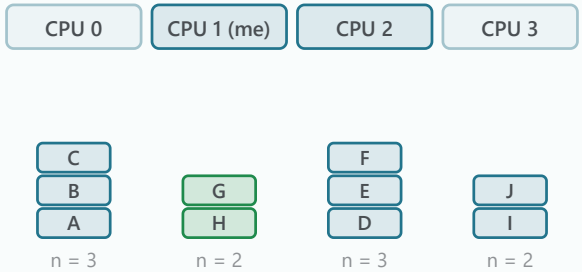
Step 7 / 8

VARIABLES

`max = 2` `c = 3` `t = 2` `k = 0`

Load Balancing: Code Trace

```
1 void balance(int me) {    /* idle CPU */
2   int max = me;
3   for (int c = 0; c < NCPU; c++)
4     if (q[c].n > q[max].n) /* busier */
5       max = c;
6   int t = (q[max].n - q[me].n) / 2;
▶   for (int k = 0; k < t; k++)
▶     migrate(&q[max], &q[me]); /* from tail */
9 }
```



k = 1: **G**, at the tail of CPU 2's queue, migrates to CPU 1. Queue lengths 3, 2, 3, 2. Migrated jobs find a cold cache and TLB on the new CPU: the price of affinity.

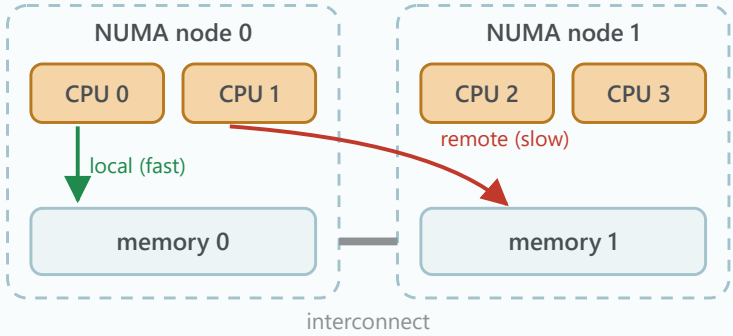
Step 8 / 8

VARIABLES

`max = 2` `c = 3` `t = 2` `k = 1`

Processor Affinity, NUMA and Heterogeneous Cores

- **Processor affinity**: keeping a thread on the CPU where its cache and TLB are warm. **Soft**: the scheduler tries but does not guarantee it. **Hard**: an allowed CPU set (`sched_setaffinity()`, `taskset -c 0-3`).
- **NUMA**: running a thread on the node where its memory is; Linux automatic NUMA balancing moves pages and tasks closer together.
- Load balancing and affinity **conflict**: migration restores balance but breaks cache warmth and locality.



Heterogeneous multiprocessing: ARM big.LITTLE / DynamIQ, P/E cores since Intel's 12th generation (Thread Director gives hardware hints), Apple M series. Linux **EAS** (Energy Aware Scheduling) places work on the core that meets the performance target with the least energy; background jobs go to efficiency cores.

Real-Time Systems and Latency

Hard real time

Missing a deadline means system failure: ABS brakes, airbags, industrial robot control, flight control.

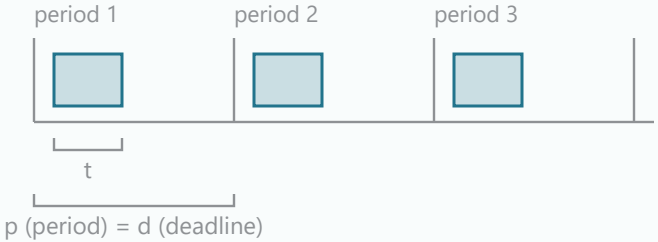
Soft real time

Critical tasks get preference but there is no guarantee; a miss lowers quality: video streaming, VoIP, games, audio playback.

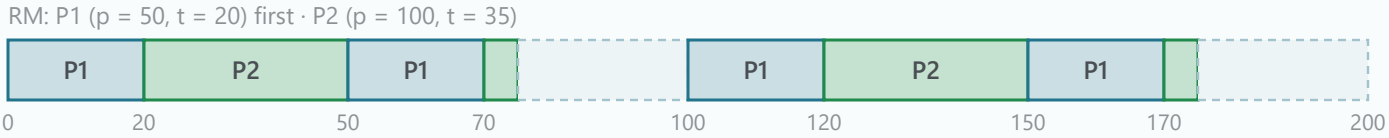


- **Conflicts:** preempting any work running in the kernel, and the release of resources held by low-priority processes. Hence a preemptive, priority-based kernel is needed; for hard guarantees, **admission control** as well.
- Algorithms can be **static** (priorities computed in advance, e.g. RM) or **dynamic** (decisions at run time, e.g. EDF).

Periodic Tasks and Rate-Monotonic (RM) Scheduling

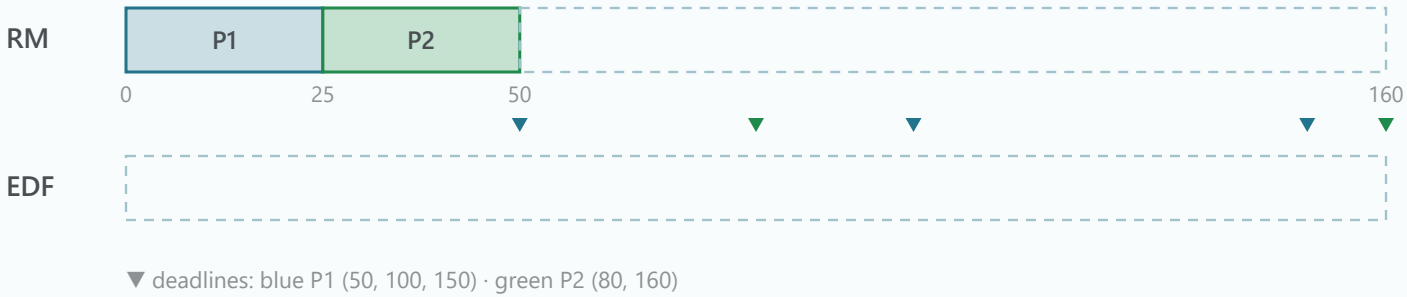


- Periodic task: $0 \le t \le d \le p$; utilization $U = t / p$.
- **Rate-monotonic** (RM): static priority = $1 / p$; a shorter period means higher priority. It is **optimal** among static-priority algorithms.
- Liu–Layland bound: if $\Sigma U \le n(2^{1/n} - 1)$, RM meets every deadline. $n = 1$: $1 \cdot n = 2$: $\approx 0.828 \cdot n = 3$: $\approx 0.780 \cdot n \to \infty$: $\ln 2 \approx 0.693$. A sufficient condition, not a necessary one.



Example: $U = 20/50 + 35/100 = 0.40 + 0.35 = 0.75 \le 0.828$ → schedulable with RM. Recomputed from Silberschatz Figure 5.21.

RM and EDF Step by Step



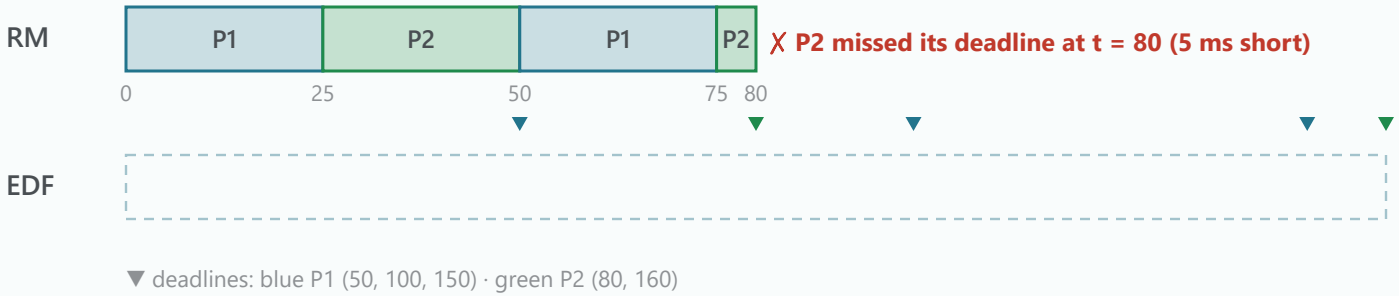
▼ deadlines: blue P1 (50, 100, 150) · green P2 (80, 160)

U = 25/50 + 35/80 = 0.5 + 0.4375 = **0.94** > 0.828: no guarantee for RM. RM: P1 (shorter period) first, 0–25; P2 25–50.

Step 1 / 5

Proportional-share scheduling: T shares in total are allocated; an application with N shares gets N/T of the CPU, and admission control rejects requests that exceed the total. Linux `SCHED_DEADLINE` (3.14+) combines EDF with CBS bandwidth protection.

RM and EDF Step by Step

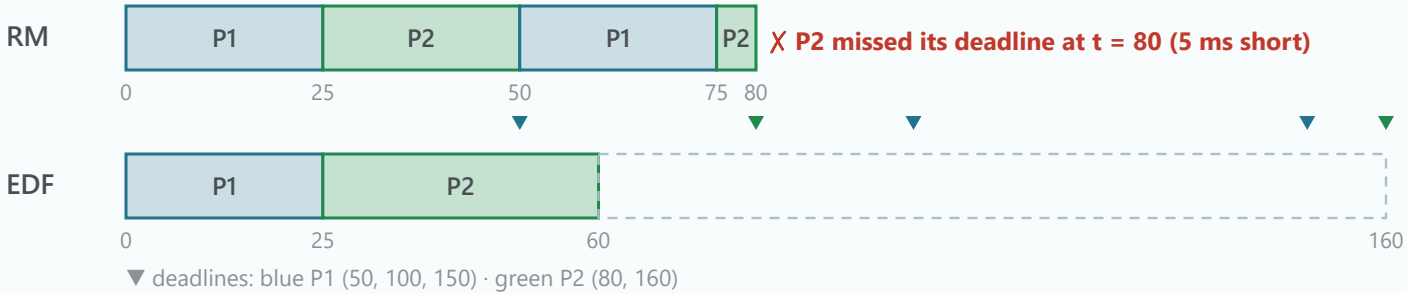


t = 50: P1's next job arrives and, with its higher static priority, preempts P2. P2 runs 5 more ms at 75–80 but time is up: **deadline missed**.

Step 2 / 5

Proportional-share scheduling: T shares in total are allocated; an application with N shares gets N/T of the CPU, and admission control rejects requests that exceed the total. Linux `SCHED_DEADLINE` (3.14+) combines EDF with CBS bandwidth protection.

RM and EDF Step by Step

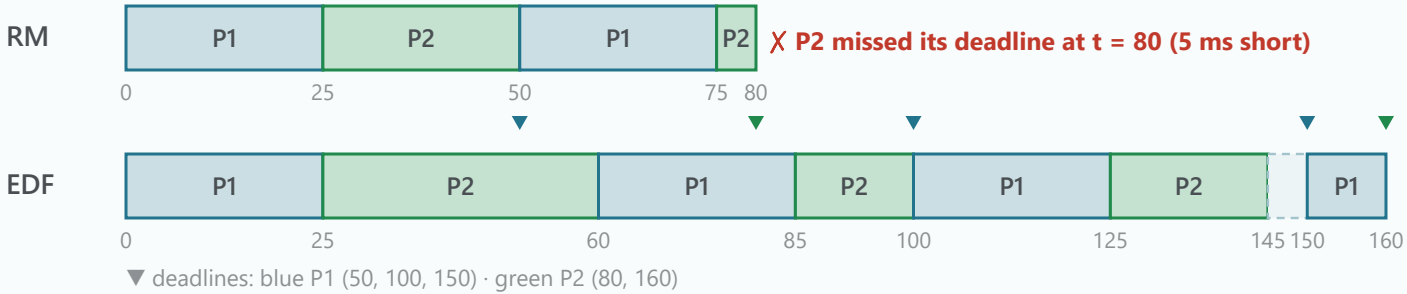


EDF: the deadline sets the priority. At t = 50 P1's deadline is 100 and P2's is 80 → **P2 continues** and finishes at 60.

Step 3 / 5

Proportional-share scheduling: T shares in total are allocated; an application with N shares gets N/T of the CPU, and admission control rejects requests that exceed the total. Linux `SCHED_DEADLINE` (3.14+) combines EDF with CBS bandwidth protection.

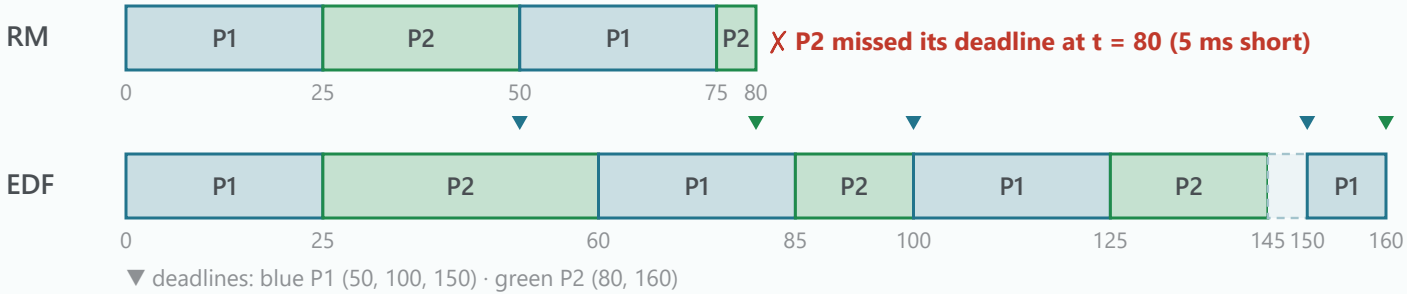
RM and EDF Step by Step



At t = 80 P2's next job has deadline 160 and P1's is 100 → P1 runs until 85. At 100, P1 (150) goes first again. Every deadline is met.

Proportional-share scheduling: T shares in total are allocated; an application with N shares gets N/T of the CPU, and admission control rejects requests that exceed the total. Linux `SCHED_DEADLINE` (3.14+) combines EDF with CBS bandwidth protection.

RM and EDF Step by Step



For EDF the condition is ΣU ≤ 1 (preemptive, independent tasks, d = p): 0.94 ≤ 1, and there is no miss over the 400 ms hyperperiod. The price: dynamic priorities and unpredictable losses under overload.

Proportional-share scheduling: T shares in total are allocated; an application with N shares gets N/T of the CPU, and admission control rejects requests that exceed the total. Linux `SCHED_DEADLINE` (3.14+) combines EDF with CBS bandwidth protection.

The Liu–Layland Test: Code Trace

```
▶ int rm_test(Task g[], int n) {
▶   double U = 0;
3   for (int i = 0; i < n; i++)
4     U += (double) g[i].t / g[i].p;
5   double bound = n * (pow(2, 1.0/n) - 1);
6   if (U <= bound) return 1; /* enough */
7   if (U <= 1.0) return 0;   /* unknown */
8   return -1;              /* impossible */
9 }
```



U = 0

Tasks: T1(p=20, t=5), T2(p=50, t=10), T3(p=100, t=30). The total utilization U starts at zero.

Step 1 / 9

VARIABLES

`n = 3` `U = 0`

OUTPUT

The test is a sufficient condition: if it returns 1, RM certainly works; if it returns 0, a schedule must be drawn to decide (e.g. from 0 to the longest period).

The Liu–Layland Test: Code Trace

```
1 int rm_test(Task g[], int n) {
2   double U = 0;
▶   for (int i = 0; i < n; i++)
▶     U += (double) g[i].t / g[i].p;
5   double bound = n * (pow(2, 1.0/n) - 1);
6   if (U <= bound) return 1; /* enough */
7   if (U <= 1.0) return 0;   /* unknown */
8   return -1;              /* impossible */
9 }
```



U = 0.25

T1: 5 / 20 = 0.25 is added → U = **0.25**.

Step 2 / 9

VARIABLES

`n = 3` `i = 0` `U = 0.25`

OUTPUT

The test is a sufficient condition: if it returns 1, RM certainly works; if it returns 0, a schedule must be drawn to decide (e.g. from 0 to the longest period).

The Liu–Layland Test: Code Trace

```
1 int rm_test(Task g[], int n) {
2   double U = 0;
▶   for (int i = 0; i < n; i++)
▶     U += (double) g[i].t / g[i].p;
5   double bound = n * (pow(2, 1.0/n) - 1);
6   if (U <= bound) return 1; /* enough */
7   if (U <= 1.0) return 0;   /* unknown */
8   return -1;              /* impossible */
9 }
```



U = 0.45

T2: 10 / 50 = 0.2 is added → U = **0.45**.

Step 3 / 9

VARIABLES

`n = 3` `i = 1` `U = 0.45`

OUTPUT

The test is a sufficient condition: if it returns 1, RM certainly works; if it returns 0, a schedule must be drawn to decide (e.g. from 0 to the longest period).

The Liu–Layland Test: Code Trace

```
1 int rm_test(Task g[], int n) {
2   double U = 0;
▶   for (int i = 0; i < n; i++)
▶     U += (double) g[i].t / g[i].p;
5   double bound = n * (pow(2, 1.0/n) - 1);
6   if (U <= bound) return 1; /* enough */
7   if (U <= 1.0) return 0;   /* unknown */
8   return -1;              /* impossible */
9 }
```



U = 0.75

T3: 30 / 100 = 0.3 is added → U = **0.75**.

Step 4 / 9

VARIABLES

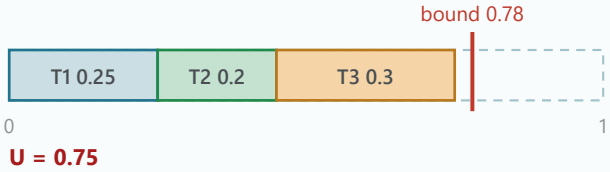
`n = 3` `i = 2` `U = 0.75`

OUTPUT

The test is a sufficient condition: if it returns 1, RM certainly works; if it returns 0, a schedule must be drawn to decide (e.g. from 0 to the longest period).

The Liu–Layland Test: Code Trace

```
1 int rm_test(Task g[], int n) {
2   double U = 0;
3   for (int i = 0; i < n; i++)
4     U += (double) g[i].t / g[i].p;
▶   double bound = n * (pow(2, 1.0/n) - 1);
▶   if (U <= bound) return 1; /* enough */
7   if (U <= 1.0) return 0;   /* unknown */
8   return -1;              /* impossible */
9 }
```



bound = $3(2^{1/3} - 1) = 0.7798$; U = 0.75 ≤ bound → **1**: RM meets every deadline.

Step 5 / 9

VARIABLES

`n = 3` `U = 0.75` `bound = 0.7798`

OUTPUT

```
rm_test = 1
```

The test is a sufficient condition: if it returns 1, RM certainly works; if it returns 0, a schedule must be drawn to decide (e.g. from 0 to the longest period).

The Liu–Layland Test: Code Trace

```
▶ int rm_test(Task g[], int n) {
▶   double U = 0;
3   for (int i = 0; i < n; i++)
4     U += (double) g[i].t / g[i].p;
5   double bound = n * (pow(2, 1.0/n) - 1);
6   if (U <= bound) return 1; /* enough */
7   if (U <= 1.0) return 0;   /* unknown */
8   return -1;              /* impossible */
9 }
```



U = 0

The second call, on the set from the RM and EDF example: P1(p=50, t=25), P2(p=80, t=35).

Step 6 / 9

VARIABLES

`n = 2` `U = 0`

OUTPUT

```
rm_test = 1
```

The test is a sufficient condition: if it returns 1, RM certainly works; if it returns 0, a schedule must be drawn to decide (e.g. from 0 to the longest period).

The Liu–Layland Test: Code Trace

```
1 int rm_test(Task g[], int n) {
2   double U = 0;
▶   for (int i = 0; i < n; i++)
▶     U += (double) g[i].t / g[i].p;
5   double bound = n * (pow(2, 1.0/n) - 1);
6   if (U <= bound) return 1; /* enough */
7   if (U <= 1.0) return 0;   /* unknown */
8   return -1;              /* impossible */
9 }
```



U = 0.5

P1: 25 / 50 = 0.5 is added → U = **0.5**.

Step 7 / 9

VARIABLES

`n = 2` `i = 0` `U = 0.5`

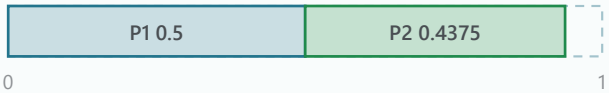
OUTPUT

```
rm_test = 1
```

The test is a sufficient condition: if it returns 1, RM certainly works; if it returns 0, a schedule must be drawn to decide (e.g. from 0 to the longest period).

The Liu–Layland Test: Code Trace

```
1 int rm_test(Task g[], int n) {
2   double U = 0;
▶   for (int i = 0; i < n; i++)
▶     U += (double) g[i].t / g[i].p;
5   double bound = n * (pow(2, 1.0/n) - 1);
6   if (U <= bound) return 1; /* enough */
7   if (U <= 1.0) return 0;   /* unknown */
8   return -1;              /* impossible */
9 }
```



U = 0.9375

P2: 35 / 80 = 0.4375 is added → U = **0.9375**.

Step 8 / 9

VARIABLES

`n = 2` `i = 1` `U = 0.9375`

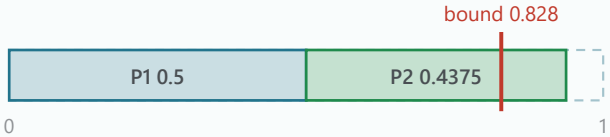
OUTPUT

```
rm_test = 1
```

The test is a sufficient condition: if it returns 1, RM certainly works; if it returns 0, a schedule must be drawn to decide (e.g. from 0 to the longest period).

The Liu–Layland Test: Code Trace

```
1 int rm_test(Task g[], int n) {
2   double U = 0;
3   for (int i = 0; i < n; i++)
4     U += (double) g[i].t / g[i].p;
▶   double bound = n * (pow(2, 1.0/n) - 1);
▶   if (U <= bound) return 1; /* enough */
▶   if (U <= 1.0) return 0;   /* unknown */
8   return -1;              /* impossible */
9 }
```



U = 0.9375

bound = 0.8284; U = 0.9375 exceeds the bound but is ≤ 1 → **0**: the test cannot decide. On this set RM really does miss a deadline, while EDF meets them all.

Step 9 / 9

VARIABLES

`n = 2` `U = 0.9375` `bound = 0.8284`

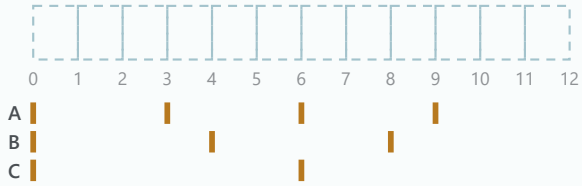
OUTPUT

```
rm_test = 1
rm_test = 0
```

The test is a sufficient condition: if it returns 1, RM certainly works; if it returns 0, a schedule must be drawn to decide (e.g. from 0 to the longest period).

The EDF Scheduler: Code Trace

```
▶ /* A(P=3,C=1) B(P=4,C=1) C(P=6,C=2) */
2  for (int t = 0; t < 12; t++) {
3      for (int i = 0; i < n; i++)
4          if (t % P[i] == 0) {      /* new job */
5              left[i] = C[i];
6              due[i] = t + P[i];
7          }
8      int s = -1;
9      for (int i = 0; i < n; i++)
10         if (left[i] > 0 &&
11             (s < 0 || due[i] < due[s]))
12             s = i;
13     if (s >= 0) left[s]--;     /* 1 ms */
14 }
```



Three periodic jobs, U = 1/3 + 1/4 + 2/6 = **0.917** ≤ 1. The common period is 12 ms; a decision is made every millisecond. Arrows mark job arrivals; the `due` array holds each job's deadline.

Step 1 / 10

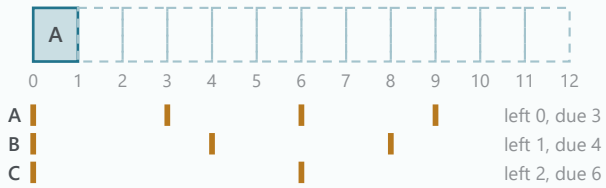
VARIABLES

U = **0.917**

OUTPUT

The EDF Scheduler: Code Trace

```
1  /* A(P=3,C=1) B(P=4,C=1) C(P=6,C=2) */
   for (int t = 0; t < 12; t++) {
       for (int i = 0; i < n; i++)
           if (t % P[i] == 0) {      /* new job */
               left[i] = C[i];
               due[i] = t + P[i];
7          }
       int s = -1;
       for (int i = 0; i < n; i++)
           if (left[i] > 0 &&
               (s < 0 || due[i] < due[s]))
               s = i;
       if (s >= 0) left[s]--;     /* 1 ms */
14 }
```



t = 0: A, B, C start a new period. Ready: A, B, C; deadlines A 3, B 4, C 6 → the earliest, **A**, runs.

Step 2 / 10

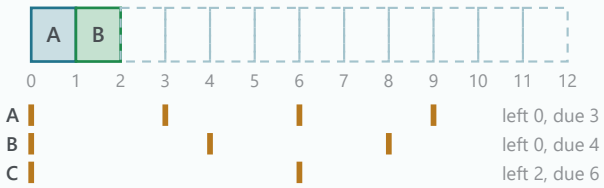
VARIABLES

`t = 0` `left = [0,1,2]` `due = [3,4,6]` `s = A`

OUTPUT

The EDF Scheduler: Code Trace

```
1  /* A(P=3,C=1) B(P=4,C=1) C(P=6,C=2) */
▶  for (int t = 0; t < 12; t++) {
3      for (int i = 0; i < n; i++)
4          if (t % P[i] == 0) {      /* new job */
5              left[i] = C[i];
6              due[i] = t + P[i];
7          }
▶      int s = -1;
▶      for (int i = 0; i < n; i++)
▶          if (left[i] > 0 &&
▶              (s < 0 || due[i] < due[s]))
▶              s = i;
▶      if (s >= 0) left[s]--;     /* 1 ms */
14 }
```



Ready: B, C; deadlines B 4, C 6 → the earliest, **B**, runs.

Step 3 / 10

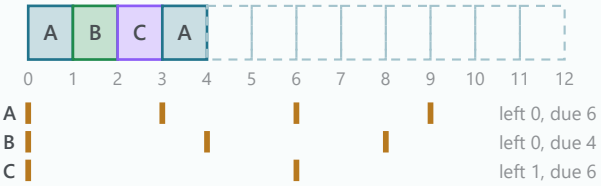
VARIABLES

`t = 1` `left = [0,0,2]` `due = [3,4,6]` `s = B`

OUTPUT

The EDF Scheduler: Code Trace

```
1  /* A(P=3,C=1) B(P=4,C=1) C(P=6,C=2) */
   for (int t = 0; t < 12; t++) {
       for (int i = 0; i < n; i++)
           if (t % P[i] == 0) {      /* new job */
               left[i] = C[i];
               due[i] = t + P[i];
7          }
       int s = -1;
       for (int i = 0; i < n; i++)
           if (left[i] > 0 &&
               (s < 0 || due[i] < due[s]))
               s = i;
       if (s >= 0) left[s]--;     /* 1 ms */
14 }
```



t = 2–3: C → A. Arriving: A at t = 3. Every millisecond the ready job with the earliest deadline is chosen.

Step 4 / 10

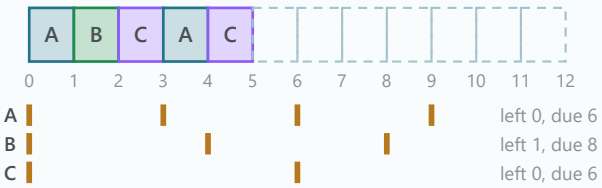
VARIABLES

`t = 3` `left = [0,0,1]` `due = [6,4,6]` `s = A`

OUTPUT

The EDF Scheduler: Code Trace

```
1  /* A(P=3,C=1) B(P=4,C=1) C(P=6,C=2) */
   for (int t = 0; t < 12; t++) {
       for (int i = 0; i < n; i++)
           if (t % P[i] == 0) {      /* new job */
               left[i] = C[i];
               due[i] = t + P[i];
7          }
       int s = -1;
       for (int i = 0; i < n; i++)
           if (left[i] > 0 &&
               (s < 0 || due[i] < due[s]))
               s = i;
       if (s >= 0) left[s]--;     /* 1 ms */
14 }
```



t = 4: B starts a new period. Ready: B, C; deadlines B 8, C 6 → the earliest, **C**, runs.

Step 5 / 10

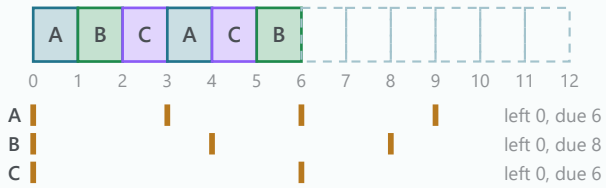
VARIABLES

`t = 4` `left = [0,1,0]` `due = [6,8,6]` `s = C`

OUTPUT

The EDF Scheduler: Code Trace

```
1  /* A(P=3,C=1) B(P=4,C=1) C(P=6,C=2) */
▶  for (int t = 0; t < 12; t++) {
3      for (int i = 0; i < n; i++)
4          if (t % P[i] == 0) {      /* new job */
5              left[i] = C[i];
6              due[i] = t + P[i];
7          }
▶      int s = -1;
▶      for (int i = 0; i < n; i++)
▶          if (left[i] > 0 &&
▶              (s < 0 || due[i] < due[s]))
▶              s = i;
▶      if (s >= 0) left[s]--;     /* 1 ms */
14 }
```



Ready: B; deadline 8 → the earliest, **B**, runs.

Step 6 / 10

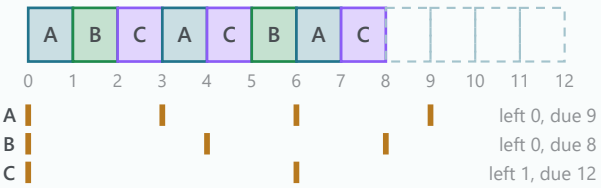
VARIABLES

`t = 5` `left = [0,0,0]` `due = [6,8,6]` `s = B`

OUTPUT

The EDF Scheduler: Code Trace

```
1  /* A(P=3,C=1) B(P=4,C=1) C(P=6,C=2) */
▶  for (int t = 0; t < 12; t++) {
3      for (int i = 0; i < n; i++)
4          if (t % P[i] == 0) {      /* new job */
5              left[i] = C[i];
6              due[i] = t + P[i];
7          }
▶      int s = -1;
▶      for (int i = 0; i < n; i++)
▶          if (left[i] > 0 &&
▶              (s < 0 || due[i] < due[s]))
▶              s = i;
▶      if (s >= 0) left[s]--;     /* 1 ms */
14 }
```



t = 6–7: A → C. Arriving: A and C at t = 6. Every millisecond the ready job with the earliest deadline is chosen.

Step 7 / 10

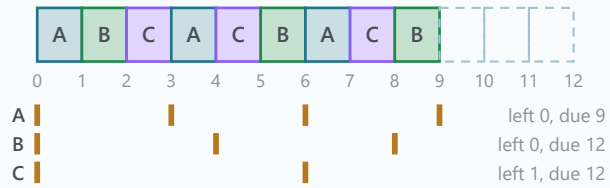
VARIABLES

`t = 7` `left = [0,0,1]` `due = [9,8,12]` `s = C`

OUTPUT

The EDF Scheduler: Code Trace

```
/* A(P=3,C=1) B(P=4,C=1) C(P=6,C=2) */
for (int t = 0; t < 12; t++) {
    for (int i = 0; i < n; i++)
        if (t % P[i] == 0) {      /* new job */
            left[i] = C[i];
            due[i] = t + P[i];
        }
    int s = -1;
    for (int i = 0; i < n; i++)
        if (left[i] > 0 &&
            (s < 0 || due[i] < due[s]))
            s = i;
    if (s >= 0) left[s]--;     /* 1 ms */
}
```



t = 8: B starts a new period. Ready: B, C; deadlines B 12, C 12 → the earliest, **B**, runs.

Step 8 / 10

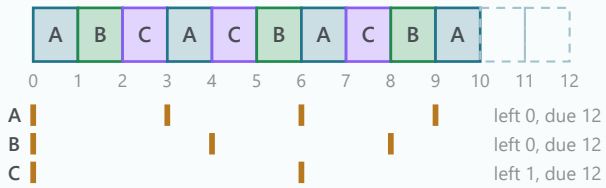
VARIABLES

`t = 8` `left = [0,0,1]` `due = [9,12,12]` `s = B`

OUTPUT

The EDF Scheduler: Code Trace

```
/* A(P=3,C=1) B(P=4,C=1) C(P=6,C=2) */
for (int t = 0; t < 12; t++) {
    for (int i = 0; i < n; i++)
        if (t % P[i] == 0) {      /* new job */
            left[i] = C[i];
            due[i] = t + P[i];
        }
    int s = -1;
    for (int i = 0; i < n; i++)
        if (left[i] > 0 &&
            (s < 0 || due[i] < due[s]))
            s = i;
    if (s >= 0) left[s]--;     /* 1 ms */
}
```



t = 9: A starts a new period. Ready: A, C; deadlines A 12, C 12 → the earliest, **A**, runs.

Step 9 / 10

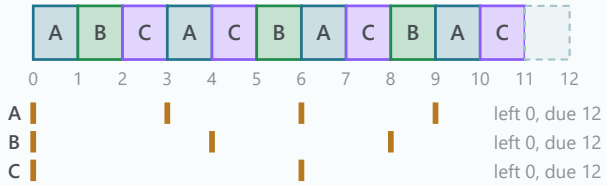
VARIABLES

`t = 9` `left = [0,0,1]` `due = [12,12,12]` `s = A`

OUTPUT

The EDF Scheduler: Code Trace

```
1  /* A(P=3,C=1) B(P=4,C=1) C(P=6,C=2) */
▶  for (int t = 0; t < 12; t++) {
3      for (int i = 0; i < n; i++)
4          if (t % P[i] == 0) {      /* new job */
5              left[i] = C[i];
6              due[i] = t + P[i];
7          }
▶      int s = -1;
▶      for (int i = 0; i < n; i++)
▶          if (left[i] > 0 &&
▶              (s < 0 || due[i] < due[s]))
▶              s = i;
▶      if (s >= 0) left[s]--;     /* 1 ms */
14 }
```



t = 10–11: C → idle. 11 ms of work in 12 ms: 1 ms idle (1 − 0.917). No job missed its deadline.

Step 10 / 10

VARIABLES

`t = 11` `left = [0,0,0]` `due = [12,12,12]` `s = -1`

OUTPUT

POSIX Real-Time Scheduling

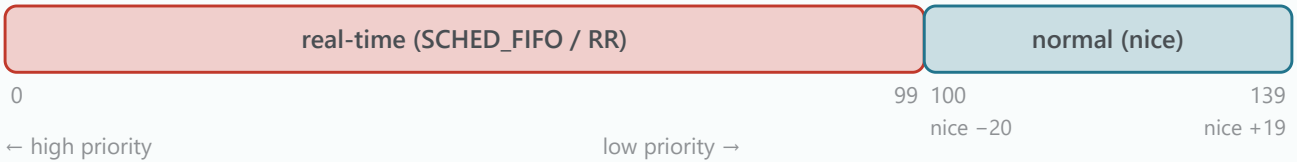
```c
pthread_attr_t attr;
struct sched_param sp;
int policy;
pthread_attr_init(&attr);
pthread_attr_getschedpolicy(&attr, &policy);
/* policy is ignored unless inheritance is off */
pthread_attr_setinheritsched(&attr,
                     PTHREAD_EXPLICIT_SCHED);
pthread_attr_setschedpolicy(&attr, SCHED_FIFO);
sp.sched_priority = 50;          /* 1..99 */
pthread_attr_setschedparam(&attr, &sp);
pthread_create(&tid, &attr, runner, NULL);
```

Policy	Behavior
`SCHED_FIFO`	Fixed priority (1–99), no time slice; runs until it blocks or yields.
`SCHED_RR`	FIFO + time slicing among equal priorities.
`SCHED_OTHER`	The default (EEVDF on Linux); also `BATCH`, `IDLE`.
`SCHED_DEADLINE`	Linux: EDF with runtime / deadline / period.

- A POSIX.1b extension; requires privilege (`CAP_SYS_NICE`); in the shell, `chrt -f 50 ./prog` . Linux gives RT jobs at most 0.95 s of every second (`sched_rt_runtime_us`).

The function names are `pthread_attr_getschedpolicy` and `pthread_attr_setschedpolicy` : no underscore between “sched” and “policy”.

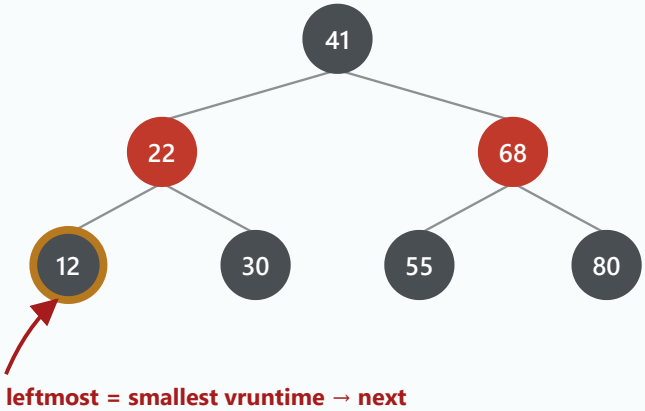
Linux: Scheduling Classes



Class (in priority order)	Policies	Selection
`stop`	— (in-kernel)	CPU stop, migration; preempts all
`deadline`	`SCHED_DEADLINE`	EDF + CBS
`rt`	`SCHED_FIFO`, `SCHED_RR`	100 priority queues + bitmap, O(1)
`fair`	`SCHED_NORMAL`, `SCHED_BATCH`, `SCHED_IDLE`	CFS (2.6.23–6.5), **EEVDF** (6.6+)
`ext`	`SCHED_EXT`	BPF-written scheduler (sched_ext, 6.12+)
`idle`	—	idle loop when there is no work

Kernel priority (lower is higher): 0–99 real-time (99 − rt_priority), 100–139 fair (nice −20 … 19 → 100 … 139). `SCHED_RR` slice: 100 ms.

Linux CFS: Virtual Runtime



- **Completely Fair Scheduler** (2.6.23–6.5): instead of a fixed time slice, each task's **virtual runtime** (vruntime) is tracked; the smallest runs.
- vruntime += Δ × 1024 / weight. The weight comes from nice: nice 0 → 1024, nice 5 → 335, nice −5 → 3121; each nice step is ≈ ×1.25 (≈ 10% CPU difference).
- Example: CPU-bound A (nice 0) and B (nice 5): A gets 1024 / 1359 ≈ **75%**, B ≈ **25%**. When B runs 10 ms, its vruntime grows by 10 × 1024 / 335 ≈ 30.6 ms.
- A sleeping I/O-bound task's vruntime grows little; when it wakes it goes straight to the front (though tasks that fell behind get only a bounded advantage).
- The tree: insertion O(log n), the leftmost is cached → selection O(1).

CFS: vruntime Code Trace

```
#define DELTA 3000  /* 3 ms, in µs */
/* nice 0 → 1024, nice 5 → 335 */
void tick(void) {
  Task *g = leftmost();   /* leftmost */
  run(g, DELTA);
  g->vruntime += DELTA * 1024 / g->weight;
  tree_insert(g);     /* re-sort */
}
```



vruntime (µs); short bar = task that fell behind

A (nice 0, weight 1024) and B (nice 5, weight 335) are CPU-bound; both have vruntime 0. On a tie A, inserted first, is leftmost in the tree.

Step 1 / 11

VARIABLES

`A.vruntime = 0` `B.vruntime = 0`

A simplified tick handler: the real CFS measures time in ns and uses a “scheduler latency” instead of a slice; the selection rule is the same.

CFS: vruntime Code Trace

```
1 #define DELTA 3000  /* 3 ms, in µs */
2 /* nice 0 → 1024, nice 5 → 335 */
3 void tick(void) {
▶   Task *g = leftmost();   /* leftmost */
▶   run(g, DELTA);
▶   g->vruntime += DELTA * 1024 / g->weight;
7   tree_insert(g);     /* re-sort */
8 }
```



A's vruntime (0) is less than or equal to B's (0) → A runs; 3000 × 1024 / 1024 = **3000** is added.

Step 2 / 11

VARIABLES

`g = A` `A.vruntime = 3000` `B.vruntime = 0`

A simplified tick handler: the real CFS measures time in ns and uses a “scheduler latency” instead of a slice; the selection rule is the same.

CFS: vruntime Code Trace

```
1 #define DELTA 3000  /* 3 ms, in µs */
2 /* nice 0 → 1024, nice 5 → 335 */
3 void tick(void) {
▶   Task *g = leftmost();   /* leftmost */
▶   run(g, DELTA);
▶   g->vruntime += DELTA * 1024 / g->weight;
7   tree_insert(g);     /* re-sort */
8 }
```



Now B is leftmost (0 < 3,000) → B runs; 3000 × 1024 / 335 = **9170** is added: a low weight means fast "aging".

Step 3 / 11

VARIABLES

`g = B` `A.vruntime = 3000` `B.vruntime = 9170`

A simplified tick handler: the real CFS measures time in ns and uses a "scheduler latency" instead of a slice; the selection rule is the same.

CFS: vruntime Code Trace

```
1 #define DELTA 3000  /* 3 ms, in µs */
2 /* nice 0 → 1024, nice 5 → 335 */
3 void tick(void) {
▶   Task *g = leftmost();   /* leftmost */
▶   run(g, DELTA);
▶   g->vruntime += DELTA * 1024 / g->weight;
7   tree_insert(g);     /* re-sort */
8 }
```



A's vruntime (3,000) is less than or equal to B's (9,170) → A runs; 3000 × 1024 / 1024 = **3000** is added.

Step 4 / 11

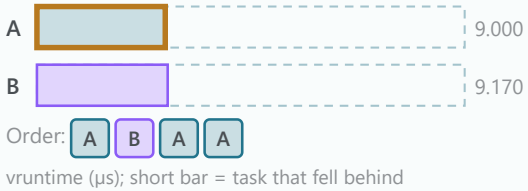
VARIABLES

`g = A` `A.vruntime = 6000` `B.vruntime = 9170`

A simplified tick handler: the real CFS measures time in ns and uses a “scheduler latency” instead of a slice; the selection rule is the same.

CFS: vruntime Code Trace

```
1 #define DELTA 3000  /* 3 ms, in µs */
2 /* nice 0 → 1024, nice 5 → 335 */
3 void tick(void) {
▶   Task *g = leftmost();   /* leftmost */
▶   run(g, DELTA);
▶   g->vruntime += DELTA * 1024 / g->weight;
7   tree_insert(g);     /* re-sort */
8 }
```



A is leftmost again (6,000 ≤ 9,170) → A runs, +3000.

Step 5 / 11

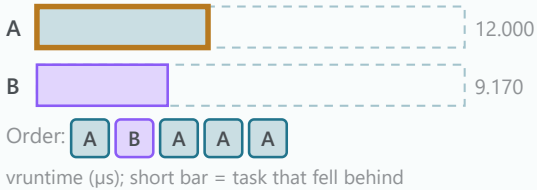
VARIABLES

`g = A` `A.vruntime = 9000` `B.vruntime = 9170`

A simplified tick handler: the real CFS measures time in ns and uses a “scheduler latency” instead of a slice; the selection rule is the same.

CFS: vruntime Code Trace

```
1 #define DELTA 3000  /* 3 ms, in µs */
2 /* nice 0 → 1024, nice 5 → 335 */
3 void tick(void) {
▶   Task *g = leftmost();   /* leftmost */
▶   run(g, DELTA);
▶   g->vruntime += DELTA * 1024 / g->weight;
7   tree_insert(g);     /* re-sort */
8 }
```



A is leftmost again (9,000 ≤ 9,170) → A runs, +3000.

Step 6 / 11

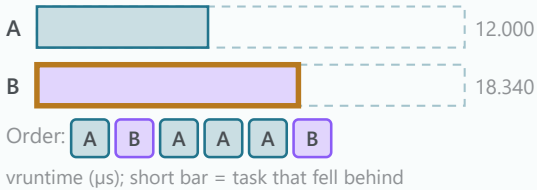
VARIABLES

`g = A` `A.vruntime = 12000` `B.vruntime = 9170`

A simplified tick handler: the real CFS measures time in ns and uses a “scheduler latency” instead of a slice; the selection rule is the same.

CFS: vruntime Code Trace

```
1 #define DELTA 3000  /* 3 ms, in µs */
2 /* nice 0 → 1024, nice 5 → 335 */
3 void tick(void) {
▶   Task *g = leftmost();   /* leftmost */
▶   run(g, DELTA);
▶   g->vruntime += DELTA * 1024 / g->weight;
7   tree_insert(g);     /* re-sort */
8 }
```



Now B is leftmost (9,170 < 12,000) → B runs; 3000 × 1024 / 335 = **9170** is added: a low weight means fast "aging".

Step 7 / 11

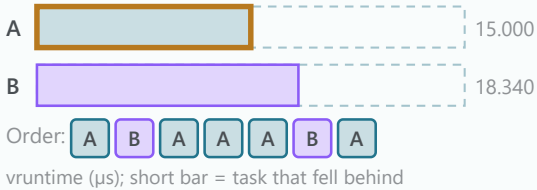
VARIABLES

`g = B` `A.vruntime = 12000` `B.vruntime = 18340`

A simplified tick handler: the real CFS measures time in ns and uses a "scheduler latency" instead of a slice; the selection rule is the same.

CFS: vruntime Code Trace

```
1 #define DELTA 3000  /* 3 ms, in µs */
2 /* nice 0 → 1024, nice 5 → 335 */
3 void tick(void) {
▶   Task *g = leftmost();   /* leftmost */
▶   run(g, DELTA);
▶   g->vruntime += DELTA * 1024 / g->weight;
7   tree_insert(g);     /* re-sort */
8 }
```



A's vruntime (12,000) is less than or equal to B's (18,340) → A runs; 3000 × 1024 / 1024 = **3000** is added.

Step 8 / 11

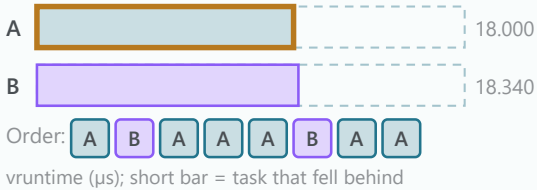
VARIABLES

`g = A` `A.vruntime = 15000` `B.vruntime = 18340`

A simplified tick handler: the real CFS measures time in ns and uses a “scheduler latency” instead of a slice; the selection rule is the same.

CFS: vruntime Code Trace

```
1 #define DELTA 3000  /* 3 ms, in µs */
2 /* nice 0 → 1024, nice 5 → 335 */
3 void tick(void) {
▶   Task *g = leftmost();   /* leftmost */
▶   run(g, DELTA);
▶   g->vruntime += DELTA * 1024 / g->weight;
7   tree_insert(g);     /* re-sort */
8 }
```



A is leftmost again (15,000 ≤ 18,340) → A runs, +3000.

Step 9 / 11

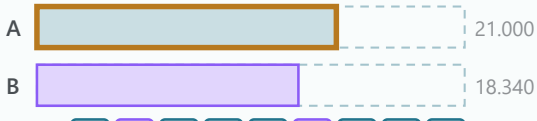
VARIABLES

`g = A` `A.vruntime = 18000` `B.vruntime = 18340`

A simplified tick handler: the real CFS measures time in ns and uses a “scheduler latency” instead of a slice; the selection rule is the same.

CFS: vruntime Code Trace

```
1 #define DELTA 3000  /* 3 ms, in µs */
2 /* nice 0 → 1024, nice 5 → 335 */
3 void tick(void) {
▶   Task *g = leftmost();   /* leftmost */
▶   run(g, DELTA);
▶   g->vruntime += DELTA * 1024 / g->weight;
7   tree_insert(g);     /* re-sort */
8 }
```



vruntime (µs); short bar = task that fell behind

A is leftmost again (18,000 ≤ 18,340) → A runs, +3000.

Step 10 / 11

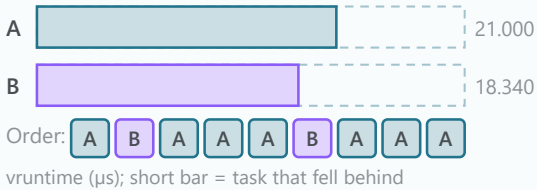
VARIABLES

`g = A` `A.vruntime = 21000` `B.vruntime = 18340`

A simplified tick handler: the real CFS measures time in ns and uses a “scheduler latency” instead of a slice; the selection rule is the same.

CFS: vruntime Code Trace

```
1 #define DELTA 3000  /* 3 ms, in µs */
2 /* nice 0 → 1024, nice 5 → 335 */
3 void tick(void) {
4   Task *g = leftmost();   /* leftmost */
5   run(g, DELTA);
6   g->vruntime += DELTA * 1024 / g->weight;
▶   tree_insert(g);      /* re-sort */
8 }
```



In 9 ticks A ran 7 times and B twice; in the long run the share approaches the 1024 : 335 weight ratio (≈ 75% – 25%). There is no fixed time slice; the share comes from the weight.

Step 11 / 11

VARIABLES

`A.vruntime = 21000`

`B.vruntime = 18340`

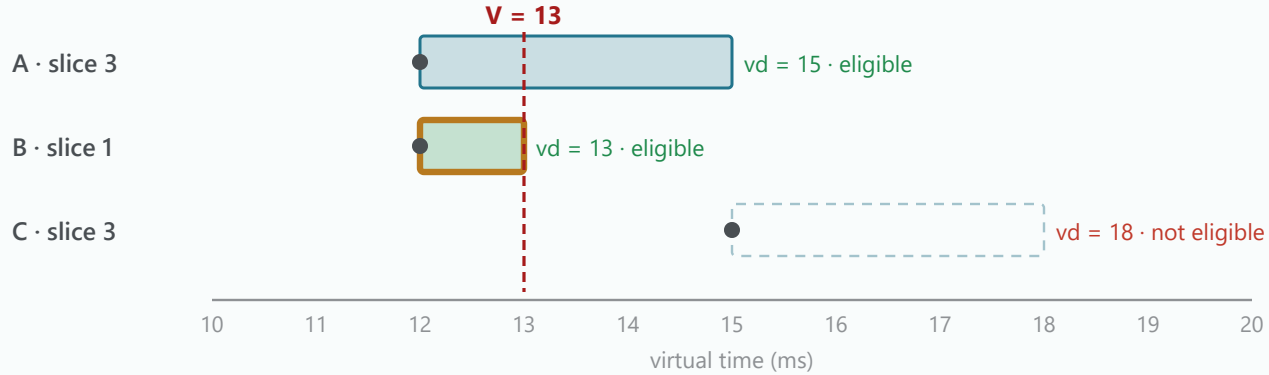
`A = 7 ticks`

`B = 2 ticks`

A simplified tick handler: the real CFS measures time in ns and uses a “scheduler latency” instead of a slice; the selection rule is the same.

EEVDF: The Successor to CFS

EEVDF (Earliest Eligible Virtual Deadline First, Stoica 1995; Linux 6.6+): a task is **eligible** if it is behind or exactly at its share (lag ≥ 0, i.e. v ≤ V); its virtual deadline is vd = v + slice / weight. Among eligible tasks the one with the earliest vd runs. Three tasks of equal weight:



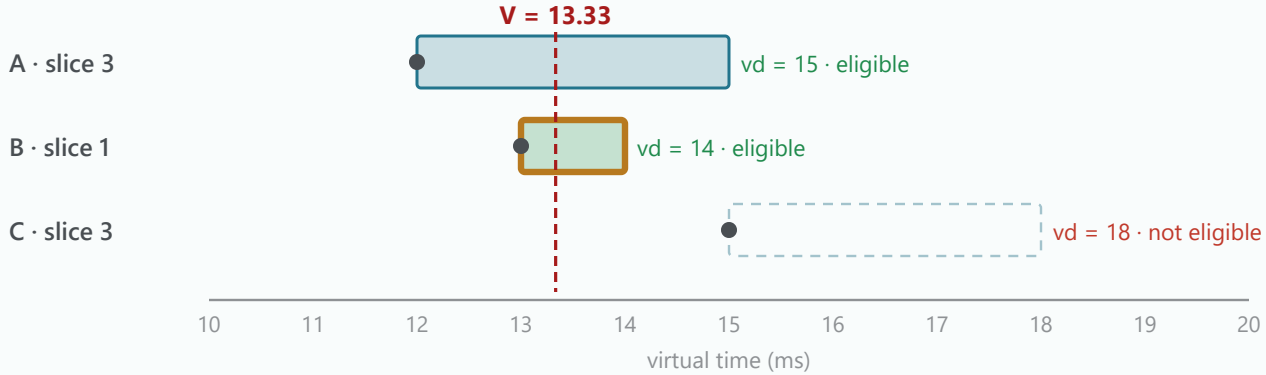
v: A 12, B 12, C 15 → V = average = **13**. C is ahead (v > V, negative lag): not eligible. A and B are eligible; vd: A 15, B 13 → **B** (short slice, early deadline).

Step 1 / 5

Two crisp criteria (eligibility and deadline) replace CFS's latency heuristics; a latency-sensitive task can ask for a short slice with `sched_setattr()` (6.12+). Weights come from nice, as in CFS.

EEVDF: The Successor to CFS

EEVDF (Earliest Eligible Virtual Deadline First, Stoica 1995; Linux 6.6+): a task is **eligible** if it is behind or exactly at its share (lag ≥ 0, i.e. v ≤ V); its virtual deadline is vd = v + slice / weight. Among eligible tasks the one with the earliest vd runs. Three tasks of equal weight:

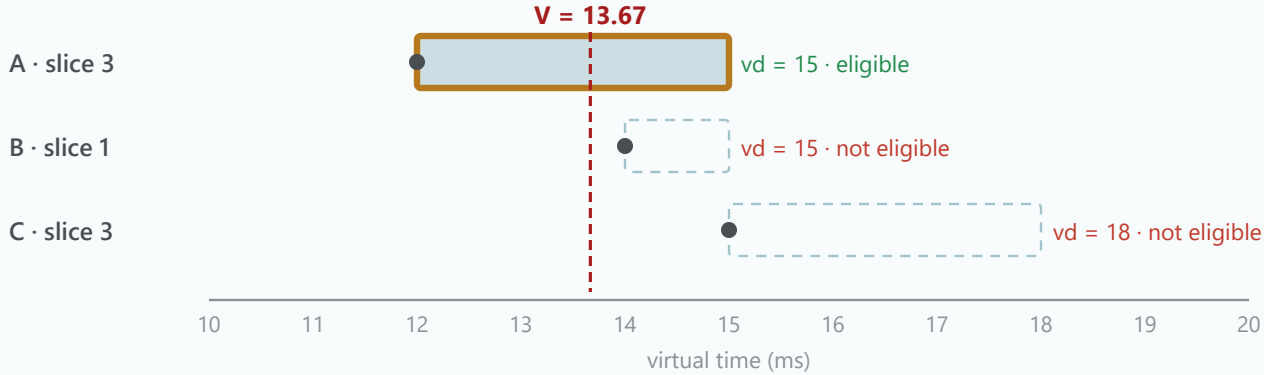


B ran 1 ms → v = 13, new vd = 14. V = 13.33; eligible: A and B → earliest vd: **B**.

Two crisp criteria (eligibility and deadline) replace CFS's latency heuristics; a latency-sensitive task can ask for a short slice with `sched_setattr()` (6.12+). Weights come from nice, as in CFS.

EEVDF: The Successor to CFS

EEVDF (Earliest Eligible Virtual Deadline First, Stoica 1995; Linux 6.6+): a task is **eligible** if it is behind or exactly at its share (lag ≥ 0, i.e. v ≤ V); its virtual deadline is vd = v + slice / weight. Among eligible tasks the one with the earliest vd runs. Three tasks of equal weight:

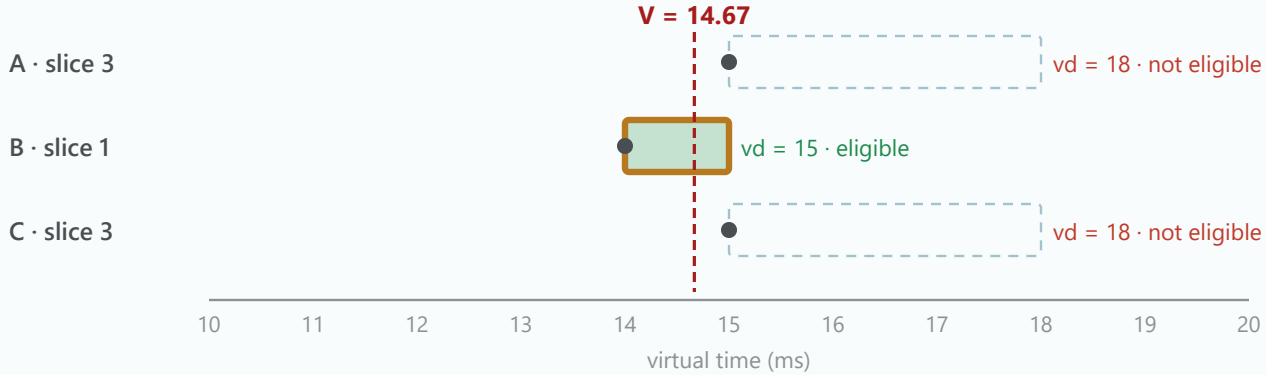


B advanced to 14; V = 13.67. Now only **A** is eligible (B and C are ahead of the average) → A runs its 3 ms slice.

Two crisp criteria (eligibility and deadline) replace CFS's latency heuristics; a latency-sensitive task can ask for a short slice with `sched_setattr()` (6.12+). Weights come from nice, as in CFS.

EEVDF: The Successor to CFS

EEVDF (Earliest Eligible Virtual Deadline First, Stoica 1995; Linux 6.6+): a task is **eligible** if it is behind or exactly at its share (lag ≥ 0, i.e. v ≤ V); its virtual deadline is vd = v + slice / weight. Among eligible tasks the one with the earliest vd runs. Three tasks of equal weight:

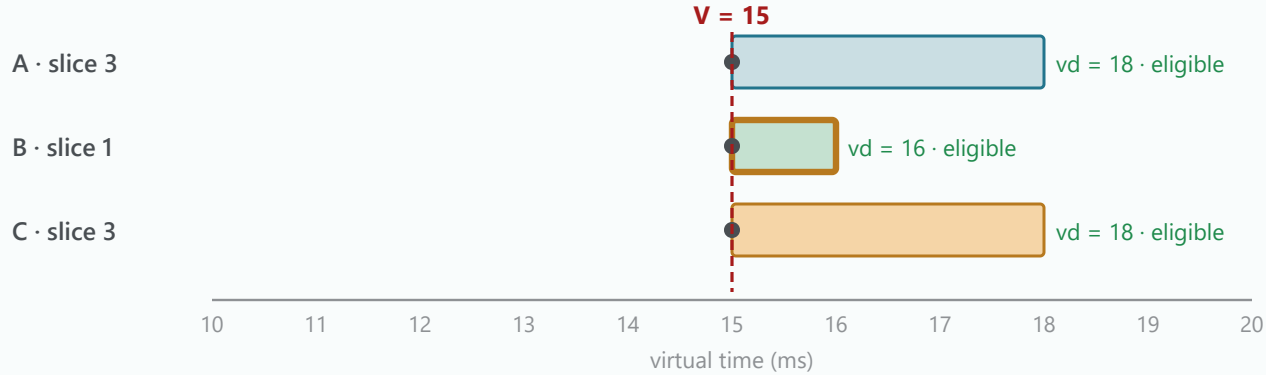


A ran 3 ms → v = 15, new vd = 18. V = 14.67; only B (v = 14) is eligible → **B**.

Two crisp criteria (eligibility and deadline) replace CFS's latency heuristics; a latency-sensitive task can ask for a short slice with `sched_setattr()` (6.12+). Weights come from nice, as in CFS.

EEVDF: The Successor to CFS

EEVDF (Earliest Eligible Virtual Deadline First, Stoica 1995; Linux 6.6+): a task is **eligible** if it is behind or exactly at its share (lag ≥ 0, i.e. v ≤ V); its virtual deadline is vd = v + slice / weight. Among eligible tasks the one with the earliest vd runs. Three tasks of equal weight:



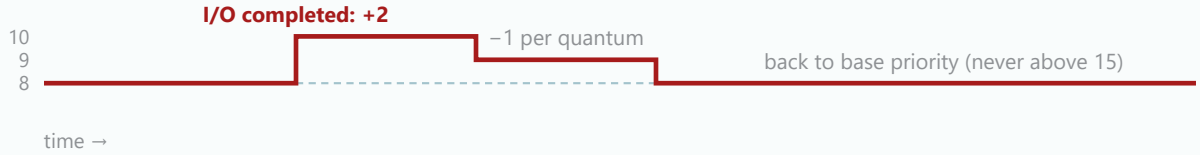
V = 15: all three are eligible. vd: A 18, B 16, C 18 → **B**. B, asking for a short slice, runs more often but briefly; thanks to the eligibility rule its long-run share is still equal.

Two crisp criteria (eligibility and deadline) replace CFS's latency heuristics; a latency-sensitive task can ask for a short slice with `sched_setattr()` (6.12+). Weights come from nice, as in CFS.

Windows Scheduling and Priorities

Relative priority	REALTIME	HIGH	ABOVE_NORMAL	NORMAL	BELOW_NORMAL	IDLE
time-critical	31	15	15	15	15	15
highest	26	15	12	10	8	6
above normal	25	14	11	9	7	5
normal	**24**	**13**	**10**	**8**	**6**	**4**
below normal	23	12	9	7	5	3
lowest	22	11	8	6	4	2
idle	16	1	1	1	1	1

- Preemptive and priority-based: the highest-priority ready thread runs until it blocks, its quantum ends or a higher-priority thread arrives. 32 levels: **1–15 variable**, **16–31 real-time**, 0 for the zero-page thread. One queue per level; with no work, the idle thread runs.
- **Dynamic boosts** in the variable class: on I/O completion (more for keyboard/mouse), for the foreground window and on GUI wakeups the priority rises temporarily, drops by 1 each quantum and never exceeds 15; a thread starved for ≈ 4 seconds is raised to 15 for one quantum. The quantum is ≈ 2 clock ticks on clients, 12 on servers.



Evaluating Algorithms

Deterministic modeling

Compute each algorithm's result for a given workload (the Gantt charts in this chapter). Exact and simple, but valid only for that input.

Queueing models

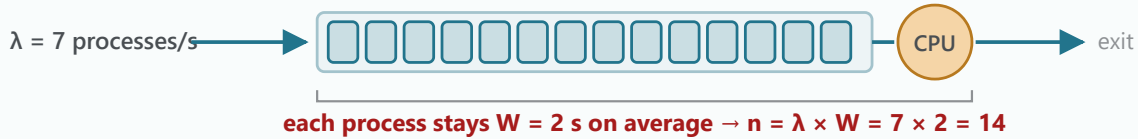
Averages from arrival and service distributions. **Little's law**: $n = \lambda \times W$. If $\lambda = 7$ processes arrive per second and each stays $W = 2$ s on average, there are on average $n = 14$ processes in the system.

Simulation

A programmed model of the system is driven by randomly generated data or a **trace** recorded from a real system. Accurate, but can take hours.

Implementation

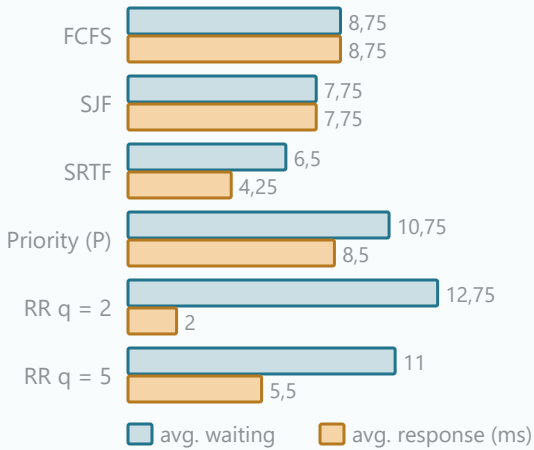
Try the algorithm in a real kernel. Today, with **sched_ext**, BPF schedulers can be loaded without rebuilding the kernel and A/B-tested in production. The catch: user behavior adapts to the algorithm too.



Comparison on the Common Set

Algorithm	Waiting	Turnaround	Response	Switches
FCFS	8,75	15,25	8,75	3
SJF	7,75	14,25	7,75	3
SRTF	6,5	13	4,25	4
Priority (P)	10,75	17,25	8,5	4
RR q = 2	12,75	19,25	2	12
RR q = 5	11	17,5	5,5	5

Averages in ms; P = preemptive; switches = number of context switches.



- **SRTF** is best in average waiting (6.5) but needs to know the burst lengths; **RR q = 2** is best in response (2) but makes the most switches.
- Priority optimizes **importance**, not waiting time. There is no single “best” algorithm: the choice depends on the goal (response, throughput, fairness, deadlines).

Exercises: Turnaround, Response and Fairness

1. A, B, C all arrive at t = 0 and each needs 5 ms of CPU. What are the average turnaround and average response time under SJF and RR (q = 1 ms)?
2. A (arrival 0, length 100), B (10, 10), C (10, 10). What are the average turnaround and response time under nonpreemptive SJF and STCF (SRTF)?
3. Unfairness metric U = completion time of the first job to finish / that of the second. Two jobs with equal tickets each need L slices. What is U under RR? How does U change with L under lottery?

▶ **Answers**

Review Questions (1/2)

1. At which state transitions does a nonpreemptive scheduler make decisions?
2. P1 = 5, P2 = 2, P3 = 1 ms, all arriving at t = 0 in this order. What is the average waiting time under FCFS and SJF?
3. $\alpha = 0.5$ and $\tau_0 = 8$. If the measured bursts are 4 and 12 ms, what is the next prediction?
4. What happens in RR if the time quantum is very large or very small? With q = 4 ms and a 1 ms switch, what fraction is lost?

▶ **Answers**

Review Questions (2/2)

5. Why does SRTF preempt P1 at t = 1 on the common set?
6. With per-core queues, CPU 1 became idle; CPUs 0, 2 and 3 hold 3, 5 and 2 jobs. How many jobs does pull balancing move, from where, and at what cost?
7. Are P1 (p = 40, t = 10) and P2 (p = 60, t = 30) schedulable with RM?
8. How do two CPU-bound tasks with nice 0 and nice 5 share the CPU under CFS? What does “eligible” mean in EEVDF?

▶ **Answers**

References

- A. Silberschatz, P. B. Galvin, G. Gagne, *Operating System Concepts*, 10th ed., Wiley, 2018 — Chapter 5: CPU Scheduling.
- A. S. Tanenbaum, H. Bos, *Modern Operating Systems*, 4th ed. (2014) / 5th ed. (2023), Pearson — Section 2.4: Scheduling.
- R. H. Arpaci-Dusseau, A. C. Arpaci-Dusseau, *Operating Systems: Three Easy Pieces* (OSTEP), ostep.org — Chapters 7–10: Scheduling, MLFQ, Lottery, Multiprocessor.
- P. Yosifovich, A. Ionescu, M. Russinovich, D. Solomon, *Windows Internals*, 7th ed., Part 1, 2017 — Chapter 4: Thread scheduling.
- Linux kernel documentation: `sched-eevdf.rst`, `sched-design-CFS.rst`, `sched-deadline.rst`, `sched-ext.rst`; the `sched(7)` man page.
- I. Stoica, H. Abdel-Wahab, "Earliest Eligible Virtual Deadline First", 1995; C. L. Liu, J. W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment", *JACM*, 1973; C. A. Waldspurger, W. E. Weihl, "Lottery Scheduling", OSDI 1994.

The End

Chapter 5: CPU Scheduling

Previous: Chapter 4: Threads · Next: Chapter 6: Synchronization