

CPU scheduling and real-time principles

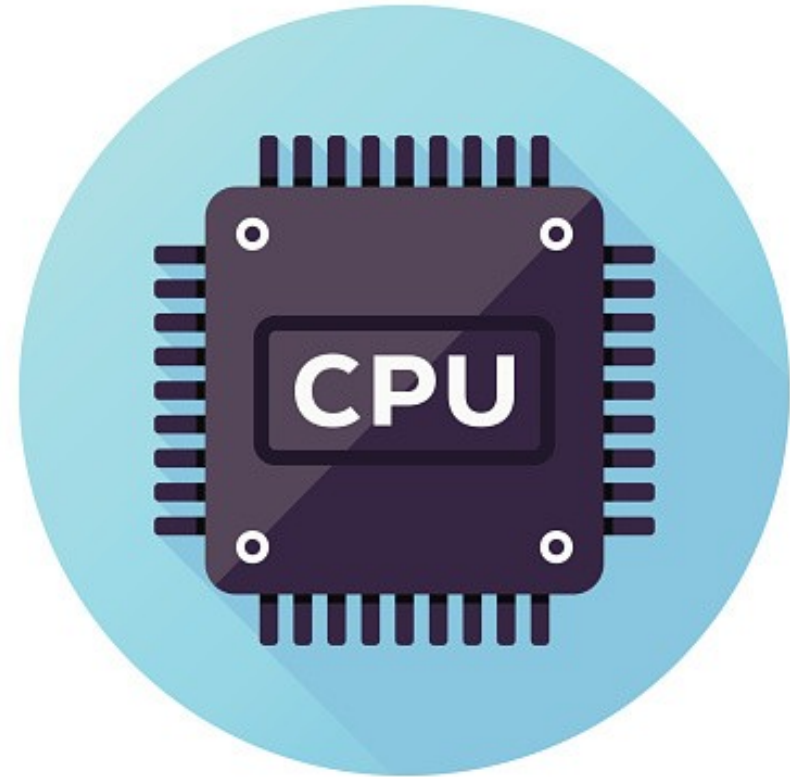
CSE513 · Week 2
Prof. Mehmet Göktürk

Sharing processors, meeting deadlines

Which ready task should execute next, and for how long?

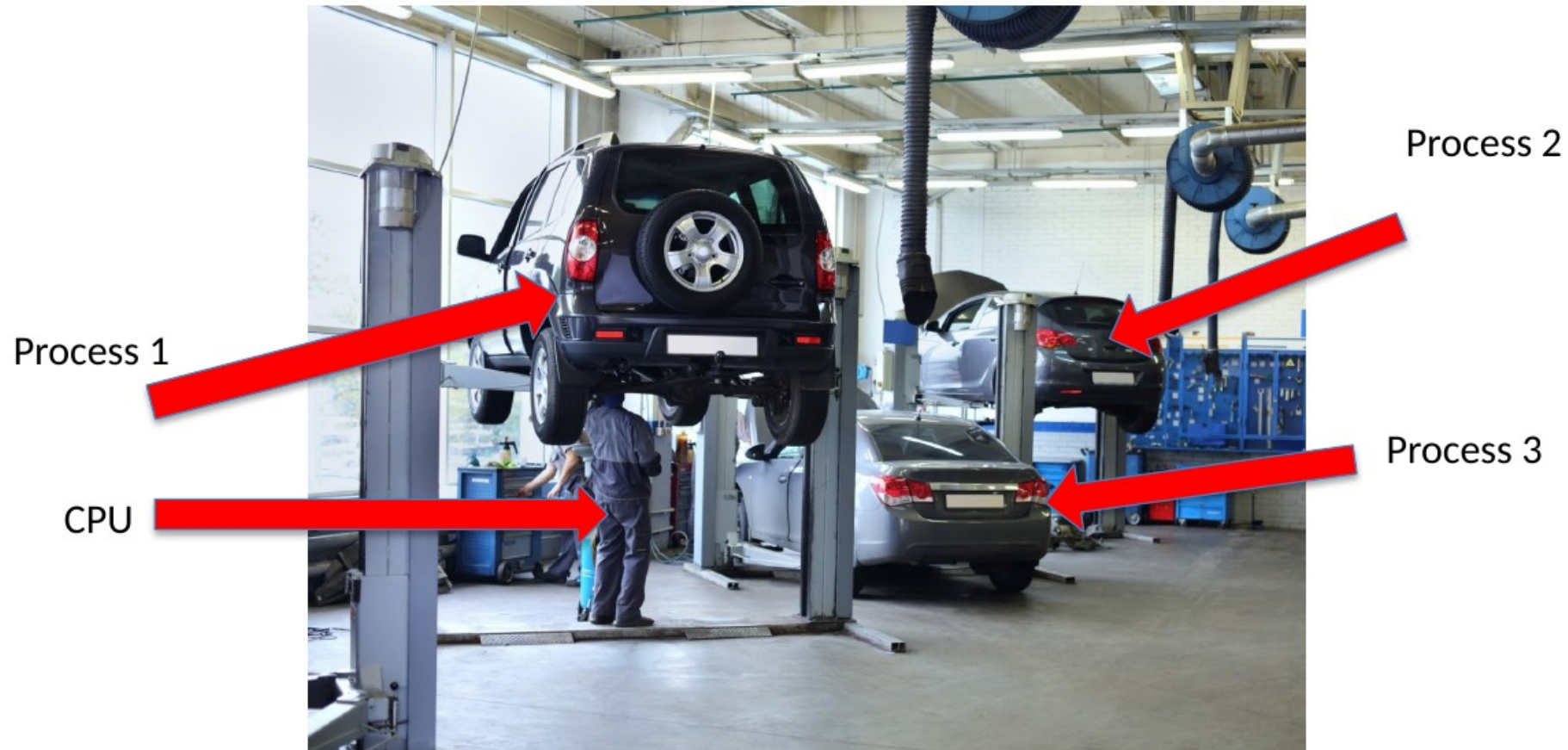
First half: multiprogramming, scheduling criteria and worked FCFS, SJF, SRTF and round-robin timelines.

Second half: priorities, multicore execution, real-time requirements, RM/EDF and Ubuntu observation.



Retained from M. Göktürk's scheduling slides

The mechanic shop analogy



Mechanic = CPU. Cars = processes. While one car waits for a part, another ready car can receive service.

Two mechanics and two execution resources

Two mechanics can make progress on different cars simultaneously. This adds parallel capacity.

One mechanic alternating among cars illustrates concurrency and multiprogramming. It does not become parallel execution on that one resource.

Shared tools and dependencies can still limit what two mechanics accomplish.



Retained from M. Göktürk's scheduling slides

Priority determination

Some work may receive priority because of urgency or policy. Arrival order is only one possible rule.

The photograph motivates a policy discussion. It does not establish that every emergency job must interrupt work immediately.

Ask what can be safely interrupted and what evidence defines urgency.



Retained from M. Göktürk's scheduling slides

Short tasks

A tyre change illustrates a relatively short service demand. Completing short jobs early can reduce average waiting.

We need the next CPU burst, not the total lifetime of an application. A long-lived interactive application can have many short bursts.



Retained from M. Göktürk's scheduling slides

Long tasks

An engine repair illustrates a larger service demand. Always favouring short work can postpone long work indefinitely.

A duration estimate can be wrong. Scheduling must account for what is known now, while preserving progress for other work.



Retained from M. Göktürk's scheduling slides

Ready queues and blocked work

A car ready for a mechanic corresponds to a runnable task awaiting CPU service.

A car waiting for an unavailable part corresponds to a blocked task. Giving it priority cannot supply the missing part.

The same task can alternate between these situations throughout its lifetime.



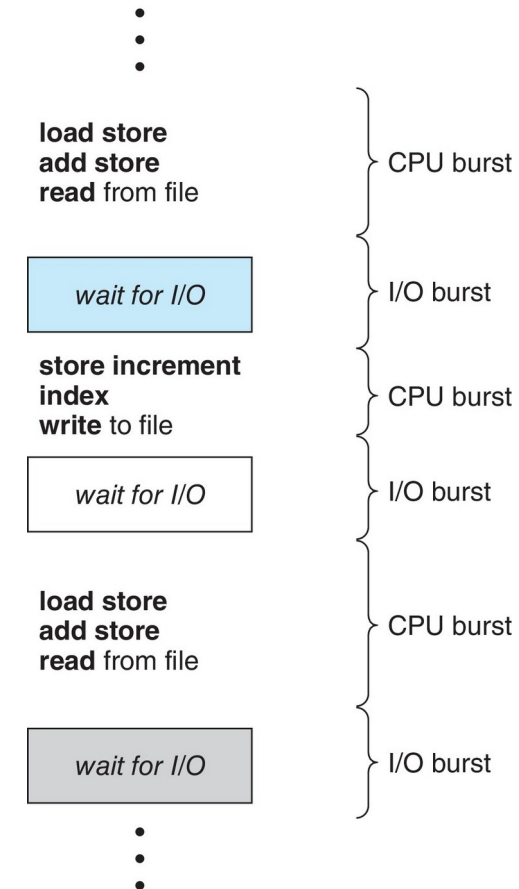
Retained from M. Göktürk's scheduling slides

CPU bursts and I/O waits

Execution alternates between CPU work and periods when an external operation must finish.

Multiprogramming overlaps one task's wait with another task's useful CPU work. It does not accelerate the waiting device.

A workload can be CPU-bound during one phase and I/O-bound during another.

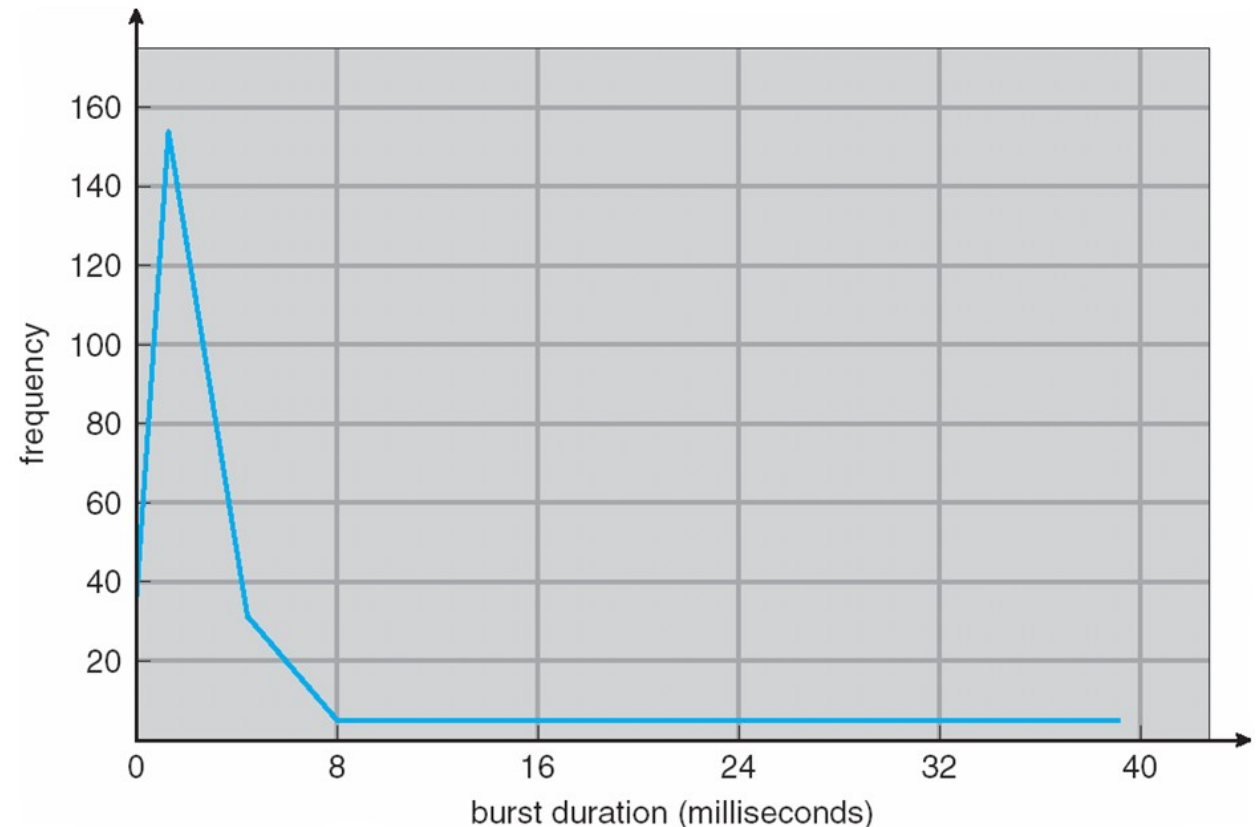


Burst lengths form a workload

The textbook histogram illustrates many short bursts and fewer long bursts. It is not a measurement of our Ubuntu machines.

The burst distribution affects whether favouring short jobs improves typical responsiveness.

Averages can hide a small number of tasks with very long waits.



Retained from M. Göktürk's scheduling slides

The scheduler chooses

The scheduler applies a policy to eligible work.
The next task need not be the one that arrived first.

The shop's service desk is the analogy: decide which job receives the available resource.

A queue can be represented by structures other than a literal FIFO list.



Retained from M. Göktürk's scheduling slides

The dispatcher transfers control

After selection, the dispatcher performs the transfer so the chosen task actually executes.

The mechanic receiving the selected job illustrates implementation of the decision.

Selection policy and context-switch mechanism answer different questions.



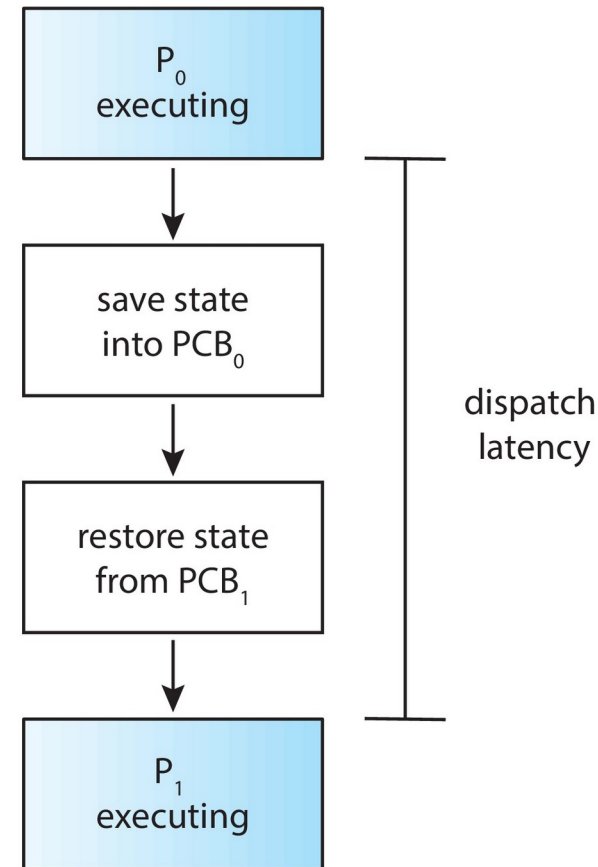
Retained from M. Göktürk's scheduling slides

Dispatch latency

Saved state allows one task to resume after another has run. Dispatching consumes time and can disturb useful cache state.

Dispatch latency is the delay in stopping one task and starting another.

Our first timelines assume zero overhead so we can isolate policy. Later examples restore a stated switch cost.



When can scheduling occur?

A running task blocks or terminates: another ready task can execute.

A timer expires or a task wakes up: the policy may choose to preempt the current task.

An interrupt is a scheduling opportunity, not proof that a different task ran.

Preemptive and nonpreemptive policies

Nonpreemptive execution retains the CPU until the current burst finishes or the task blocks.

Preemption can stop runnable work and return it to the ready queue. The task later resumes from its saved context.

Interruption can expose shared-state bugs. Synchronization is the next lecture's topic.

Utilization is busy time divided by elapsed time. Throughput is completed jobs per unit time.

Turnaround measures arrival to completion. Waiting accumulates time spent ready but not running.

Here, response time means arrival to first CPU service. An application's first visible response may occur later.

Fairness and deadline requirements cannot be inferred from a low mean waiting time alone.

For one CPU-only burst: turnaround $T = \text{completion } C - \text{arrival } A$.

Waiting $W = T - \text{burst } B$, and response $R = \text{first start } S - \text{arrival } A$.

For P arriving at 2, first running at 5 and finishing at 12 with $B = 4$: $T = 10$, $W = 6$, $R = 3$.

With I/O, turnaround also contains blocked time. Do not count that time as ready-queue waiting.

Assumptions for the first workload

One CPU, no I/O, no switching cost. All three jobs arrive at time 0.

CPU bursts: $P1 = 24$, $P2 = 3$, $P3 = 3$ time units.

Initial queue order is $P1$, $P2$, $P3$. Equal-priority ties follow that stated order.

Intervals use $[start, end)$: finishing at 24 frees the CPU for the next task at 24.

FCFS: the long job arrives first

P1 runs 0–24, P2 runs 24–27, P3 runs 27–30.

Waiting times in P1/P2/P3 order: 0, 24, 27. Mean waiting = $51 / 3 = 17$.

Completion times: 24, 27, 30. Mean turnaround = $81 / 3 = 27$.



FCFS: a different initial order

With the same burst lengths but queue P2, P3, P1, the short jobs finish first.

Waiting times in P1/P2/P3 order: 6, 0, 3. Mean waiting = $9 / 3 = 3$.

Mean turnaround = $(30 + 3 + 6) / 3 = 13$. Both orders finish all work at time 30.



The convoy effect

A long CPU burst can keep several short tasks waiting. Those tasks may be unable to issue their next I/O requests.

The single-burst example shows the initial queueing delay. A fuller CPU/I/O workload explains the resulting device utilization effects.

FCFS is simple, but arrival order can dominate the experience of short jobs.

Shortest job first

Nonpreemptive SJF selects the shortest next CPU burst among ready tasks.

For a fixed set of jobs available together, known burst lengths and zero overhead, it minimizes average waiting.

The claim depends on those assumptions. Future arrivals and unknown bursts complicate an online scheduler.

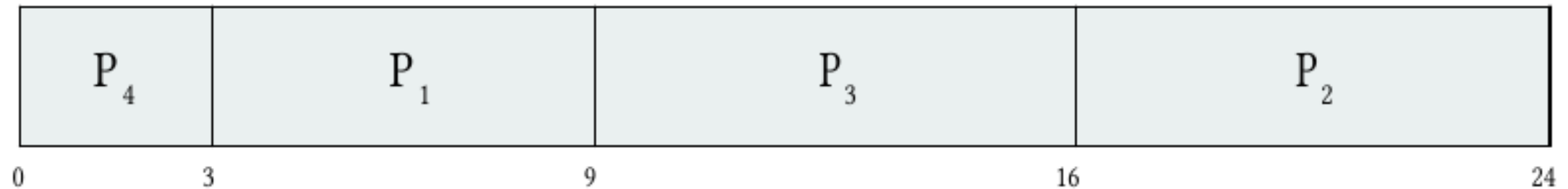
Repeated short arrivals can starve a long ready job.

SJF: a four-job timeline

All arrivals are 0. Bursts: $P_1 = 6$, $P_2 = 8$, $P_3 = 7$, $P_4 = 3$.

Order P_4, P_1, P_3, P_2 gives boundaries 0, 3, 9, 16, 24.

Waiting in $P_1/P_2/P_3/P_4$ order: 3, 16, 9, 0. Mean = $28 / 4 = 7$.



Why shorter work first reduces mean waiting

Consider adjacent jobs of lengths x and y , both ready, with $x \leq y$.

Running x before y adds x to the second job's wait. Reversing them adds y .

The pair finishes at the same time either way, so later jobs are unaffected. Choosing x first cannot increase total waiting.

Repeated exchanges give the SJF order for this simultaneous-arrival model.

Predicting the next CPU burst

Exponential averaging combines the observed burst with the previous estimate.

$$\tau(\text{next}) = \alpha \times \text{actual} + (1 - \alpha) \times \tau(\text{previous}).$$

With $\alpha = 0.5$ and estimate 10, an actual burst of 6 gives 8. A following burst of 4 gives 6.

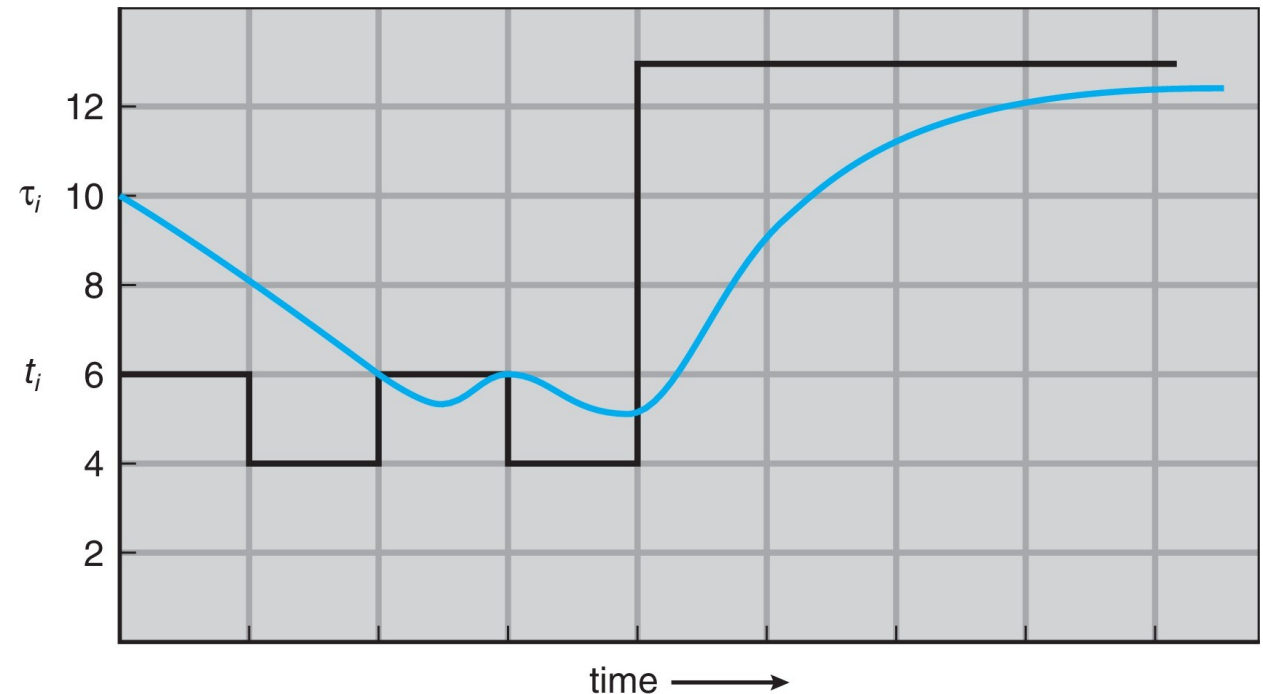
$\alpha = 0$ ignores new evidence. $\alpha = 1$ uses only the latest burst.

Prediction follows changing behaviour

The original graph shows the estimate reacting to a change in actual burst lengths.

A large α responds quickly but follows noise more closely. A small α smooths observations but adapts slowly.

An estimate is neither a known future burst nor a worst-case execution-time bound.



CPU burst (t_i)	6	4	6	4	13	13	13	...	
"guess" (τ_i)	10	8	6	6	5	9	11	12	...

Silberschatz, Galvin & Gagne · OSC 10e, Ch. 5

Shortest remaining time first

SRTF is the preemptive form: choose the ready job with the least remaining CPU demand.

At an arrival, compare the new burst with the current task's remaining work, not its original burst.

Workload: arrivals $P1/P2/P3/P4 = 0/1/2/3$, bursts = 8/4/9/5.

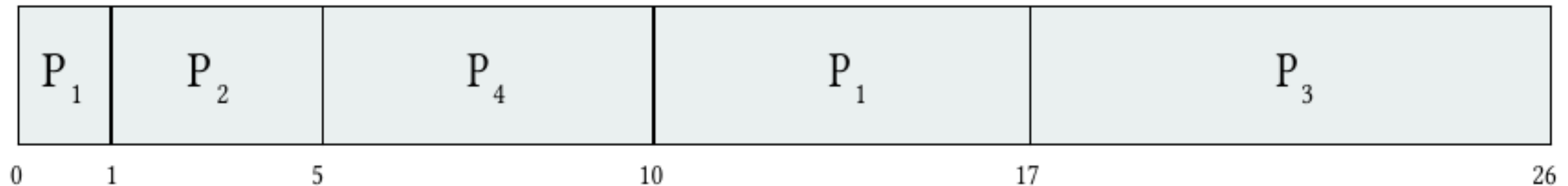
Keep the current task on an exact remaining-time tie. Assume zero switch cost.

SRTF: decisions at arrivals

At 1, P1 has 7 left and P2 needs 4, so P2 preempts. At 2 and 3, P2 still has the shortest remainder.

P2 finishes at 5. Then P4 runs to 10, P1 to 17, P3 to 26.

The chart preserves the initial P1 interval 0–1, before its later 10–17 interval.



SRTF: waiting and response are different

Completion $C = (17, 5, 26, 10)$, so turnaround $T = (17, 4, 24, 7)$.

Subtract bursts $(8, 4, 9, 5)$: waiting $W = (9, 0, 15, 2)$. Mean $W = 26 / 4 = 6.5$.

First starts $S = (0, 1, 17, 5)$: response $R = (0, 0, 15, 2)$. Mean $R = 4.25$.

P1 responds immediately yet waits 9 units later. One metric does not determine the other.

A runnable task receives at most quantum q , then returns to the queue tail if work remains.

A task that blocks or finishes early releases the CPU without consuming a full quantum.

For n continuously runnable equal peers, zero overhead and no outside arrivals, the gap between turns is at most $(n - 1)q$.

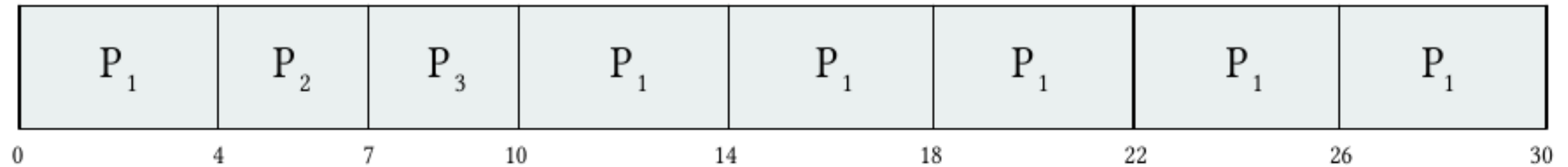
The bound changes when we add overhead, higher-priority work or a changing ready set.

Round robin: quantum 4

Reuse $P_1 = 24$, $P_2 = 3$, $P_3 = 3$, all ready at 0 in that order.

P_1 runs 0–4, P_2 4–7, P_3 7–10. Only P_1 remains and completes at 30.

Waiting = (6, 4, 7), mean $17 / 3 \approx 5.67$. Response = (0, 4, 7), mean $11 / 3 \approx 3.67$.

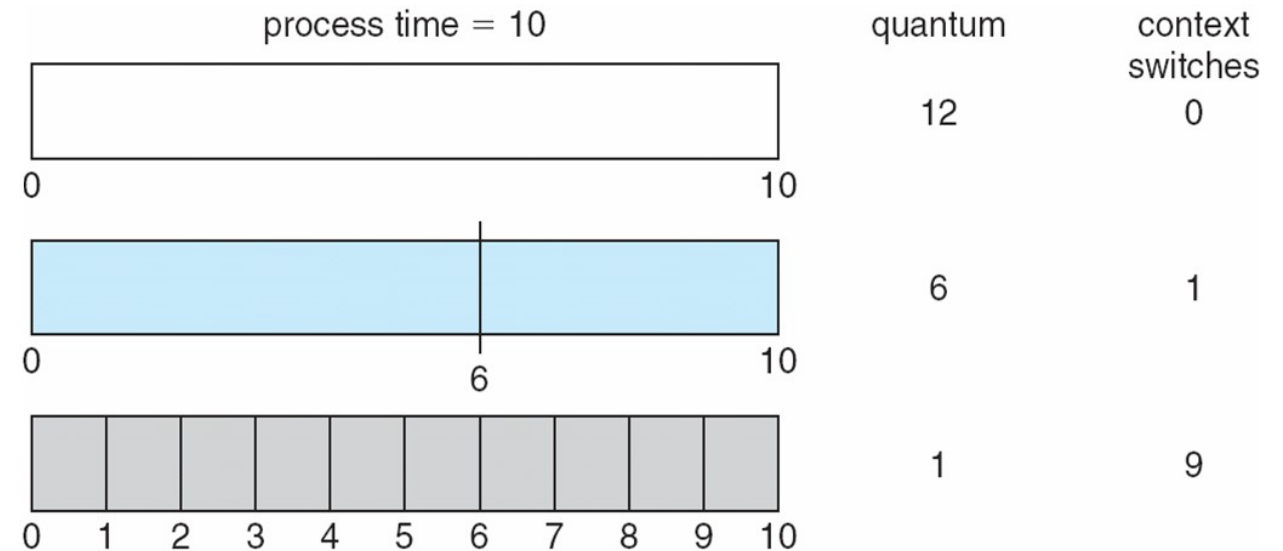


A smaller quantum has a cost

The original figure divides 10 units of useful work into smaller quanta. It counts potential preemption boundaries.

If no other task is runnable, quantum expiry need not cause a task-to-task switch.

For saturated alternating work with $q = 4$ ms and switch cost $s = 0.2$ ms, useful fraction $\approx q / (q + s) = 95.24\%$.



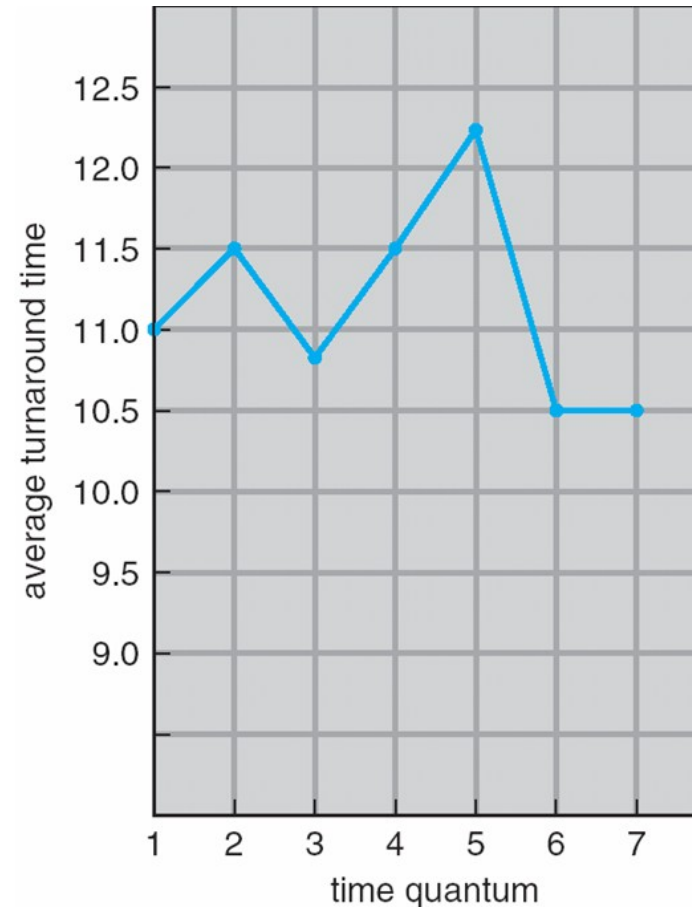
Retained from M. Göktürk's scheduling slides

Quantum and turnaround

This textbook workload has different average turnaround times for different quanta. The relation is not monotonic.

A very large quantum approaches FCFS. A tiny quantum can spend substantial time switching.

Choose a quantum using workload and latency goals. A textbook rule of thumb is not a universal Linux setting.



process	time
P_1	6
P_2	3
P_3	1
P_4	7

Retained from M. Göktürk's scheduling slides

Comparing policies on the same workload

For $P1/P2/P3 = 24/3/3$, all at 0: FCFS mean waiting = 17, mean response = 17.

SJF with tie order P2 before P3: both means = 3, but P1 waits until 6 before its first service.

RR, $q = 4$: mean waiting ≈ 5.67 , mean response ≈ 3.67 . Every job gets service by time 7.

All three finish at 30 under zero overhead. The preferred policy depends on the objective.

If P1 starts immediately and is later preempted, can its response be zero while its waiting is positive?
Yes.

Does 100% CPU utilization imply good responsiveness? No. One long job can keep the CPU busy while others wait.

When comparing algorithms, keep the workload and assumptions fixed. Mark every arrival, preemption and completion.

Priorities, multicore and deadlines

The first half allocated one CPU using arrival order, duration and time slices.

Now add importance, changing behaviour and multiple execution resources.

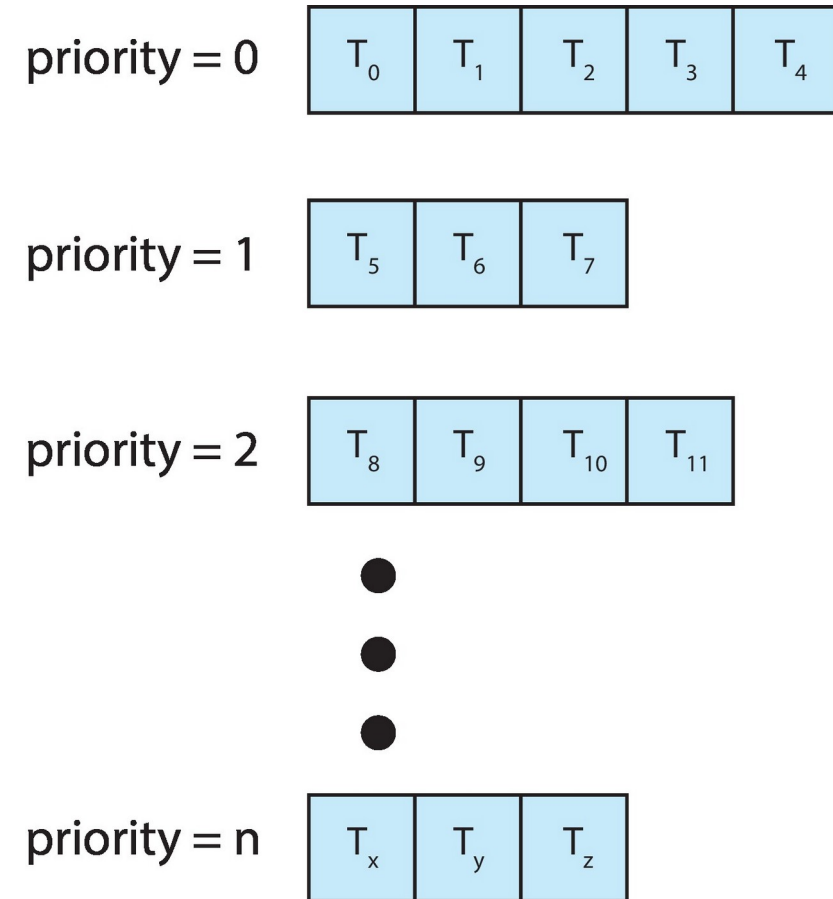
For real-time work, a completed result may be useful only if it arrives before its deadline.

Priority scheduling

Assign each ready task a priority. Select the highest-priority eligible task.

Our next textbook example uses smaller numbers for higher priorities. Linux real-time API priorities use the opposite numerical direction.

Priority policies can be preemptive or nonpreemptive. State how equal-priority tasks share the CPU.

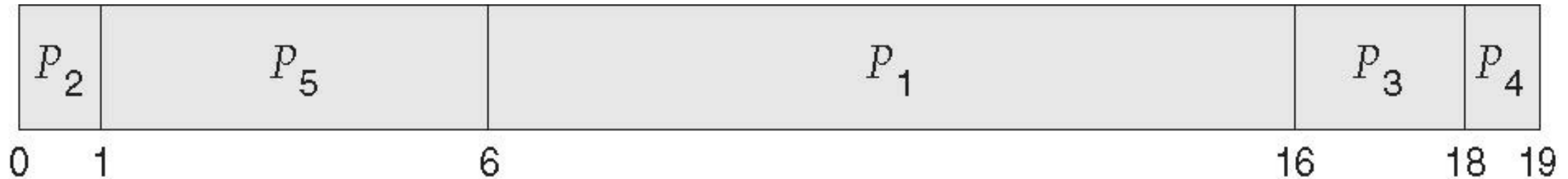


Priority scheduling: worked order

All arrive at 0. Bursts $P_1 \dots P_5 = (10, 1, 2, 1, 5)$, priorities = $(3, 1, 4, 5, 2)$. Smaller is higher.

Nonpreemptive order P_2, P_5, P_1, P_3, P_4 gives boundaries 0, 1, 6, 16, 18, 19.

Waiting in $P_1 \dots P_5$ order = $(6, 0, 16, 18, 1)$. Mean = $41 / 5 = 8.2$.



Retained from M. Göktürk's scheduling slides

An endless supply of higher-priority ready work can prevent a low-priority task from running. This is starvation.

Aging gradually improves the priority of waiting tasks. Its actual progress guarantee depends on the policy and workload.

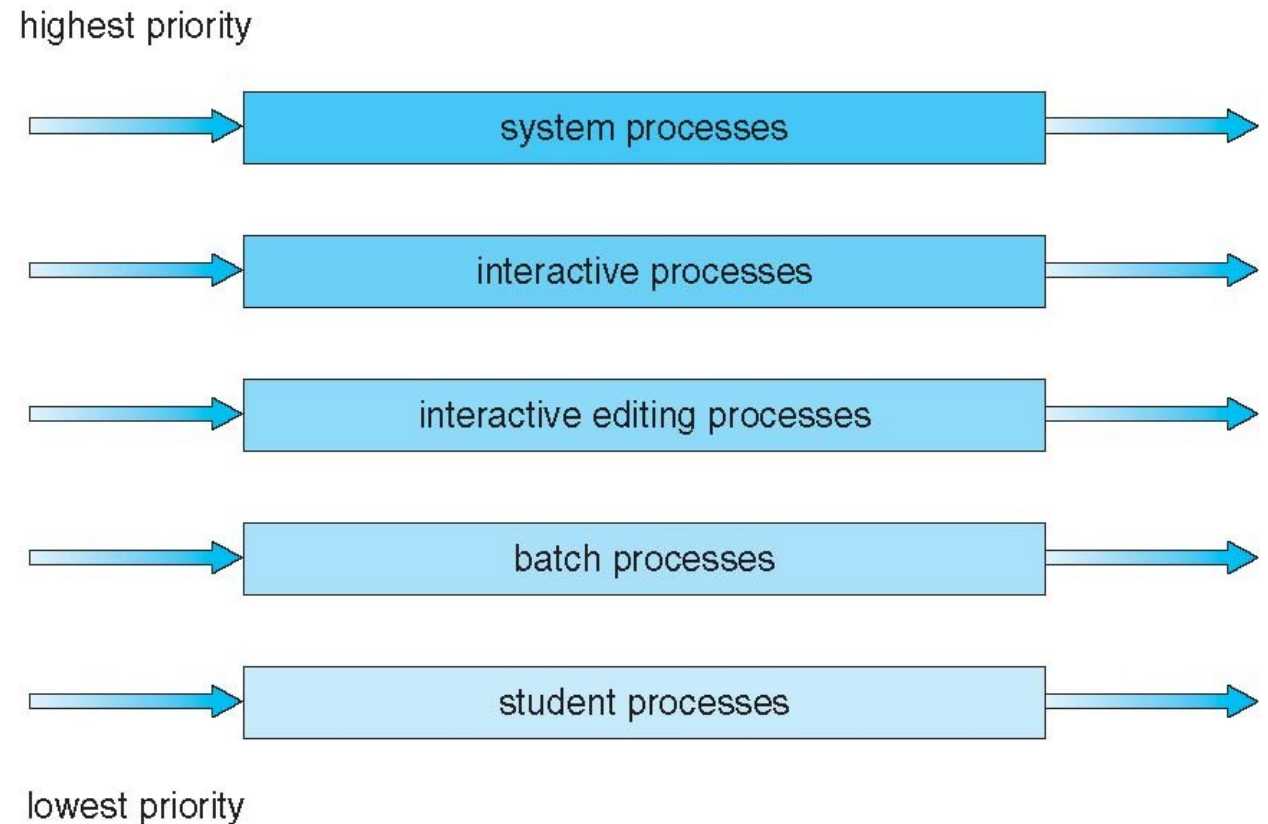
Round robin within one priority level does not stop starvation caused by higher-priority levels.

Separate queues and feedback

A multilevel queue assigns work to classes, then schedules within and between those queues.

Strict priority among queues can starve a lower queue even if each queue uses round robin internally.

A feedback design lets tasks move between queues as their observed behaviour changes.



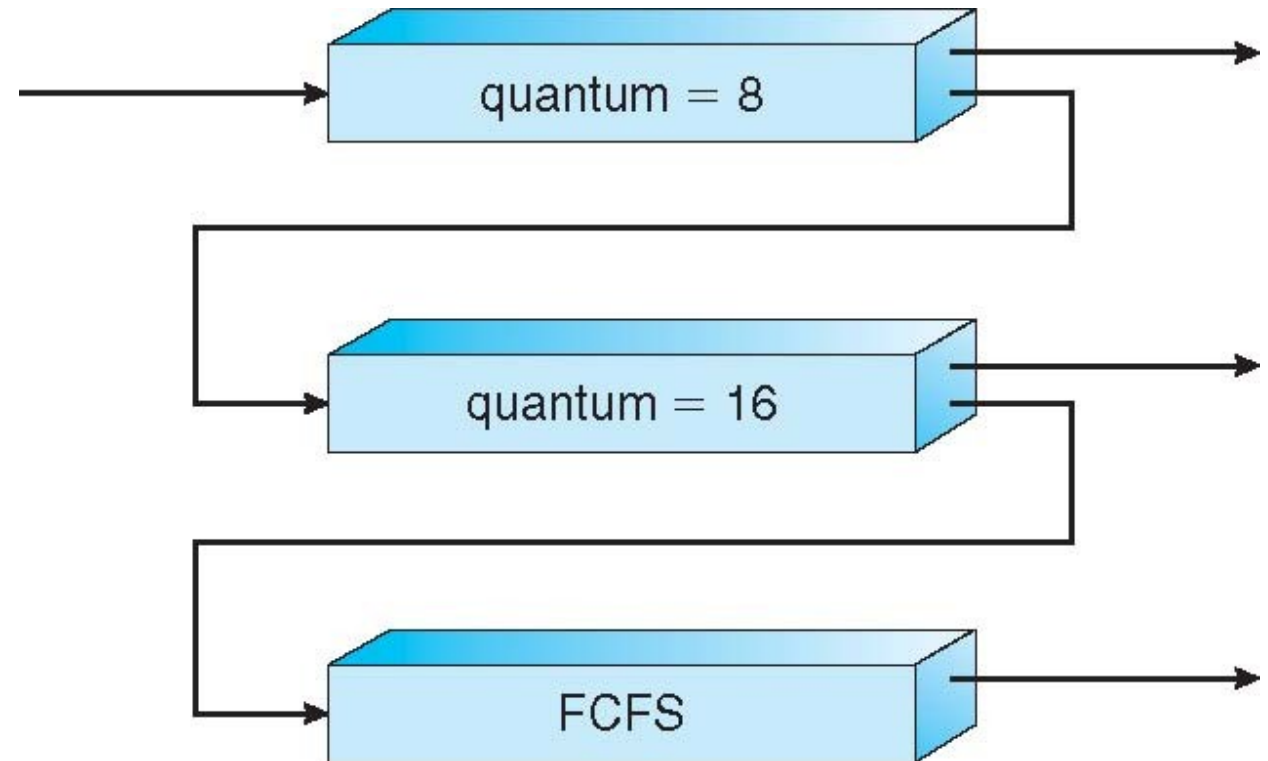
Retained from M. Göktürk's scheduling slides

Multilevel feedback queues

In this example, new work begins in Q0 with an 8-unit quantum. Full use without completion demotes it to Q1.

Q1 allows 16 units, then remaining work enters Q2 using FCFS. Higher queues take precedence.

This is a specified teaching policy. Promotion, I/O return and boost rules must also be defined in a complete implementation.



Retained from M. Göktürk's scheduling slides

Feedback: a worked CPU demand

A solitary job needs 30 CPU units. It consumes 8 in Q0, then 16 in Q1, leaving 6 in Q2.

If it is never interrupted by other work, it completes at 30. Queue changes alone do not add useful work.

For competing jobs, account for the total CPU allotment at a level. Otherwise a job may game the policy by yielding just before demotion.

Periodic priority boosts can help long-waiting work recover service.

What is scheduled?

The kernel schedules runnable execution contexts. In a system with kernel-supported threads, threads compete for CPU service.

A user-level runtime may make another scheduling decision inside the kernel thread's allocation.

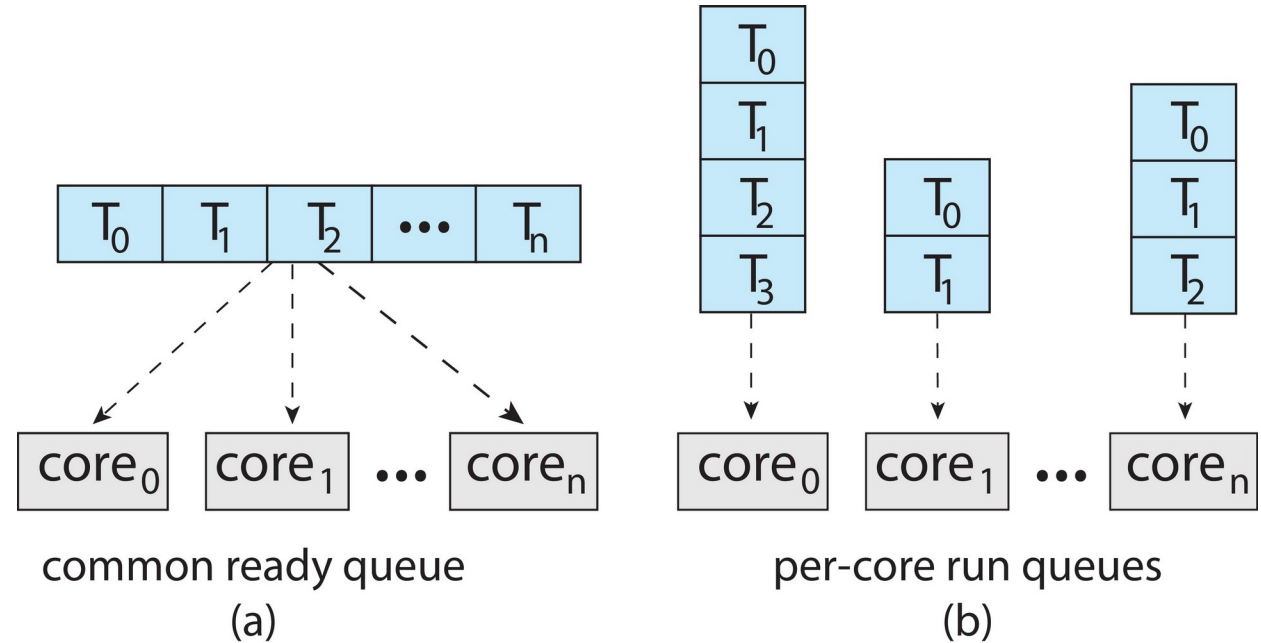
A process with several runnable threads can use several allowed logical CPUs. An individual thread executes on one logical CPU at a time.

One shared queue or several queues

A common ready queue makes global selection conceptually simple but can create contention.

Per-CPU queues reduce shared access and help locality, but can become imbalanced.

Load balancing moves work. Affinity tries to preserve the benefit of previously warmed caches.

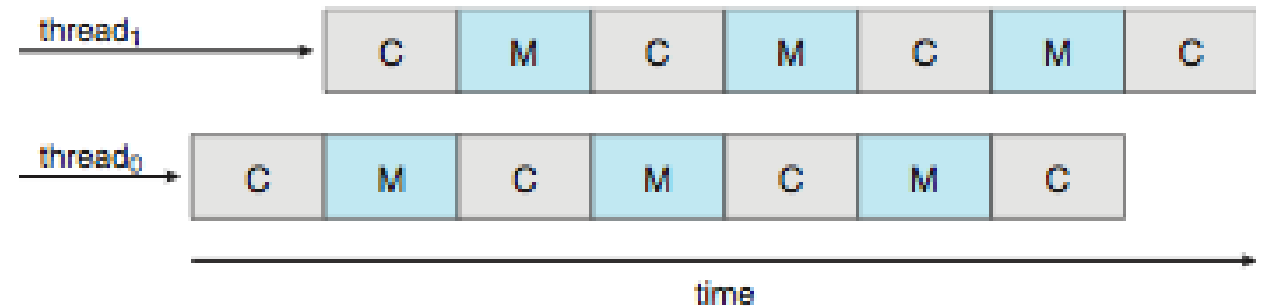


Core count and hardware threads

A physical core and an OS-visible logical CPU are different units. Hardware threads on one core share execution resources.

The original compute/stall diagram motivates overlapping work when one stream cannot proceed.

Two logical CPUs on one core do not guarantee the throughput of two independent physical cores.



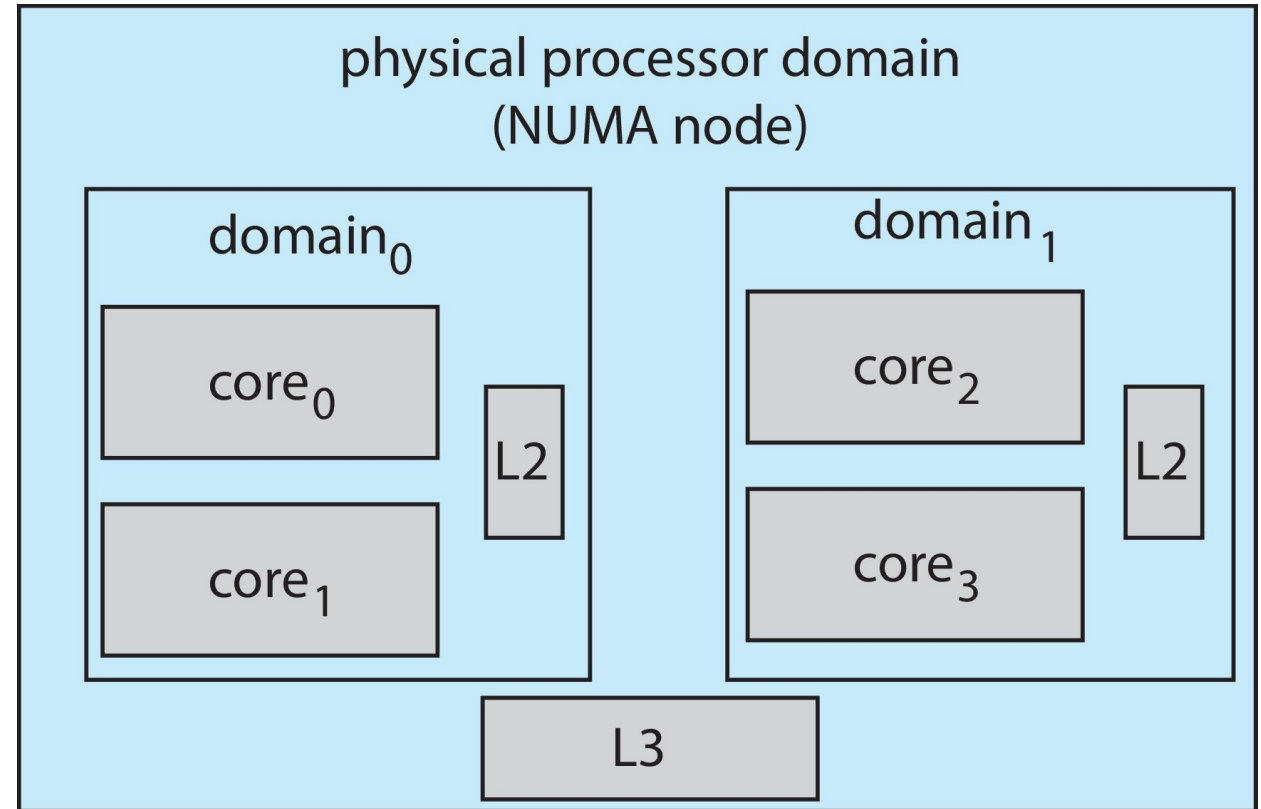
Retained from M. Göktürk's scheduling slides

Affinity, NUMA and migration

Moving a thread can improve queue balance while losing cache locality. Memory placement matters on a NUMA machine.

Scheduling domains group CPUs for balancing according to shared hardware and locality. This figure is a conceptual example.

Affinity constrains eligible CPUs. It does not reserve a CPU or remove interference from other work.



A hard real-time requirement makes deadline satisfaction part of correctness under stated operating assumptions.

Soft real-time work can tolerate some lateness with degraded quality, such as a dropped media frame.

A fast average execution time does not establish a worst-case deadline guarantee.

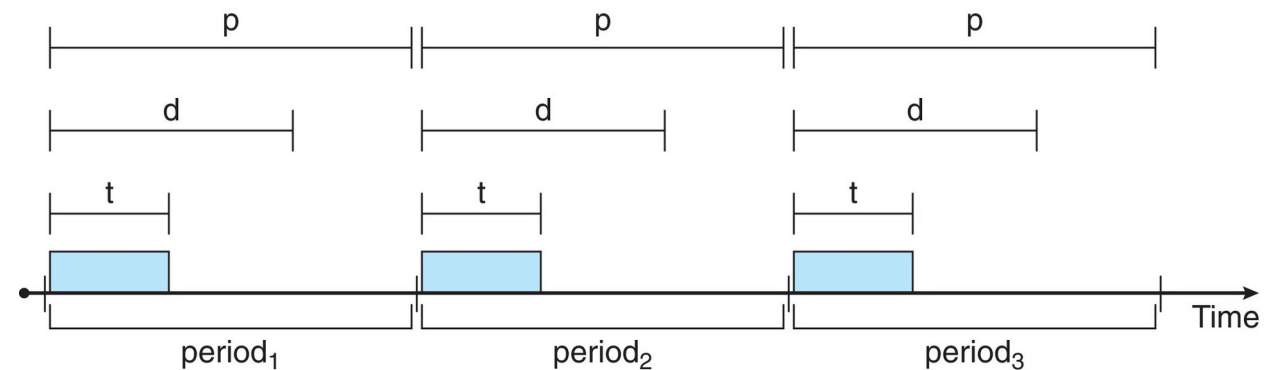
Priority alone cannot account for all blocking, interrupt delays and execution-time variation.

Release, execution demand and deadline

Each job has release r , execution demand C and absolute deadline $d = r + D$. A periodic task releases jobs every period T .

The original figure labels execution t , relative deadline d and period p . Our notation uses C , D and T to avoid ambiguity.

A task describes a repeating source of jobs. A job is one released instance.

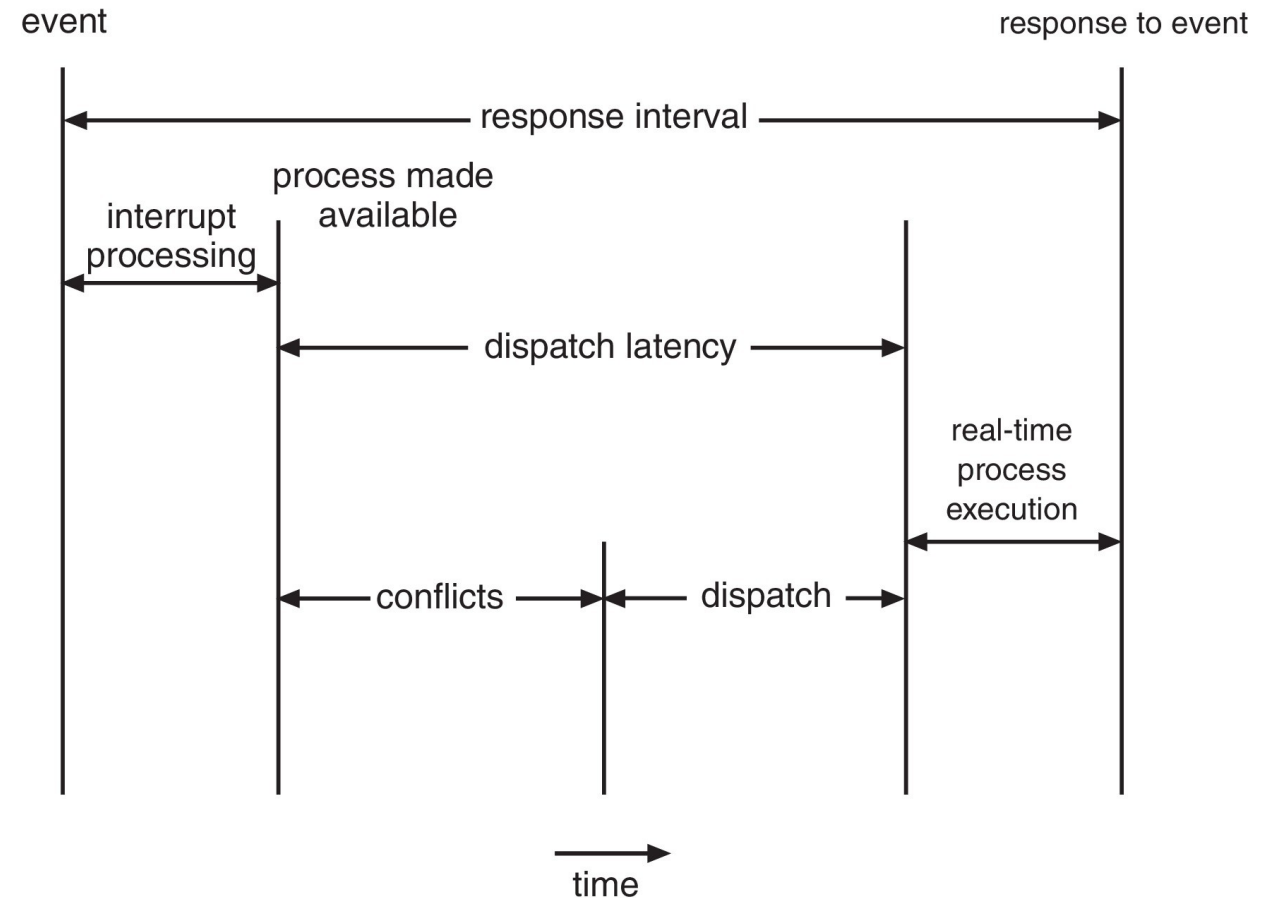


Latency before useful execution

An external event can incur interrupt latency, handler work and dispatch delay before its task runs.

A highest-priority ready task may still wait for a nonpreemptible section or a resource held elsewhere.

Deadline reasoning needs bounded execution and interference, not only a scheduler's selection rule.



The model for RM and EDF examples

One processor, independent periodic tasks, all initially released at 0. Relative deadlines equal periods: $D = T$.

Execution demands are known worst-case bounds. Jobs are fully preemptible, with zero blocking and zero scheduling overhead.

Utilization $U = \sum (C_i / T_i)$. We use milliseconds throughout the following examples.

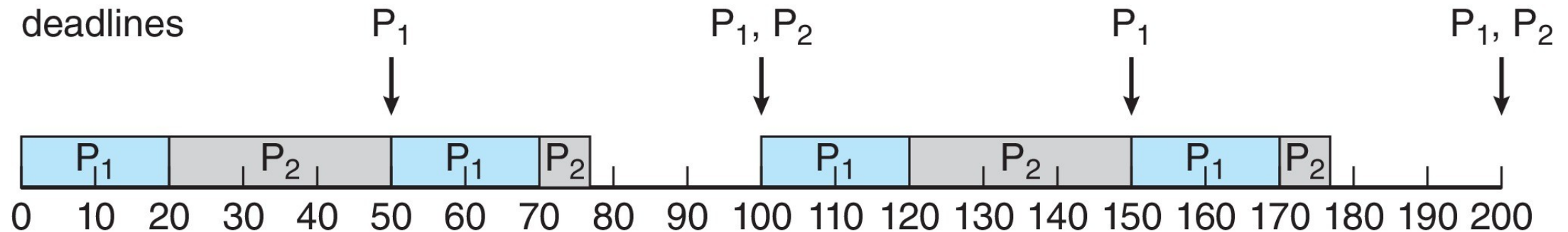
These assumptions support the mathematical results. They are not a description of every deployed real-time system.

Rate-monotonic priorities

RM assigns a fixed priority by period: shorter period means higher priority. The priority remains fixed across successive jobs.

Example A: P1 has $C_1 = 20$, $T_1 = 50$. P2 has $C_2 = 35$, $T_2 = 100$.

P1 runs 0–20, P2 20–50, P1 50–70, P2 70–75. Both first jobs meet their deadlines.



RM: a sufficient utilization bound

For n independent periodic tasks with $D = T$ under our model, $U \leq n(2^{1/n} - 1)$ guarantees RM schedulability.

For $n = 2$, the bound is about 0.8284. Example A has $U = 20/50 + 35/100 = 0.75$, so it passes.

Above the bound does not mean failure. The test is sufficient, not necessary.

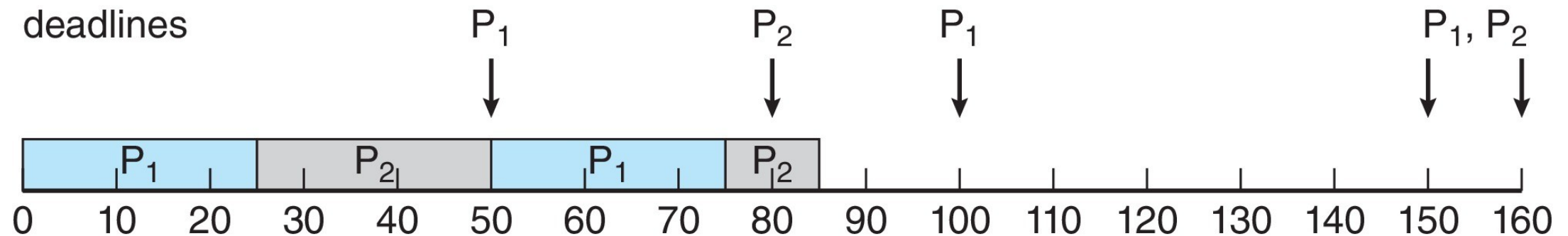
A harmonic set such as $(C, T) = (1, 2), (2, 4)$ can use $U = 1$ and still meet all RM deadlines.

RM: utilization below 1 can still fail

Example B: $P_1 = (25, 50)$, $P_2 = (35, 80)$, with $D = T$. $U = 0.5 + 0.4375 = 0.9375$.

RM runs P_1 0–25 and P_2 25–50. The next P_1 job preempts P_2 and runs 50–75.

P_2 still needs 10 at time 75, but its deadline is 80. It completes at 85 and misses by 5.



Response-time analysis for the missed job

Here R means release-to-completion time, not first service. For P2: $R(\text{next}) = C2 + \text{ceil}(R / T1) \times C1$.

Starting at $R0 = 35$: $R1 = 35 + 25 = 60$, then $R2 = 35 + 2 \times 25 = 85$. The next iteration remains 85.

$R = 85$ exceeds $D2 = 80$. This confirms the interference visible in the RM timeline.

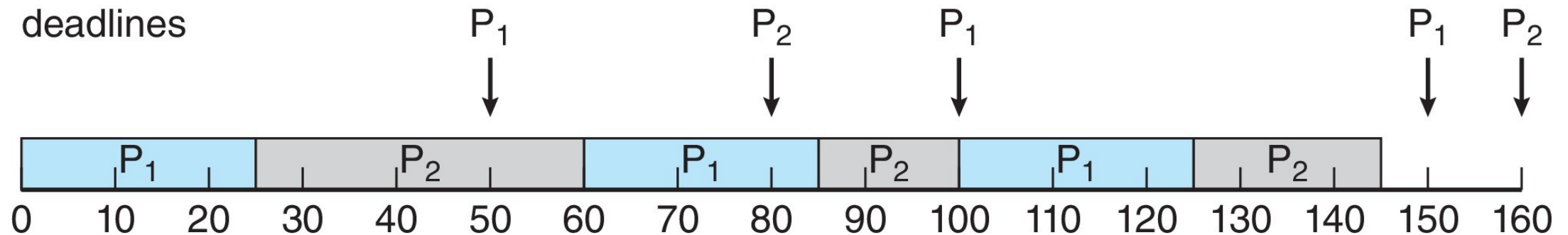
This simple recurrence assumes our independent, preemptible task model and excludes blocking and overhead.

Earliest deadline first

EDF chooses the ready job with the earliest absolute deadline. Priorities can change as jobs arrive and deadlines approach.

At time 50 in Example B, unfinished P2 has deadline 80 and the new P1 job has deadline 100.

EDF keeps P2 running until 60, then runs P1 60–85. P2 meets its deadline.



EDF: utilization and its limits

Under the single-processor, independent, fully preemptible $D = T$ model, $U \leq 1$ is the exact utilization condition for EDF feasibility.

Example B has $U = 0.9375$, so EDF can schedule it even though RM misses a deadline.

If $D < T$, aggregate utilization alone is insufficient. Work can be concentrated into shorter deadline windows.

Blocking, overhead, release jitter and multiple processors require a richer analysis.

EDF: tracing beyond the first deadline

For Example B: P2 runs 85–100, pauses for P1 100–125, then completes its second job at 145.

P1 next runs 150–175. P2 runs 175–200. At 200, its deadline 240 precedes P1's deadline 250.

P2 finishes at 210, then P1 completes at 235. Continue by comparing absolute deadlines at every event.

A finite trace illustrates decisions. The feasibility theorem provides the guarantee under the stated model.

Blocking and priority inversion

Suppose high-priority H waits for a lock held by low-priority L. Medium-priority M can preempt L and prolong H's wait.

H's priority alone cannot make the lock available. The dependency passes through L.

Priority inheritance can temporarily raise the owner's priority. Full bounds still depend on locking rules and critical-section lengths.

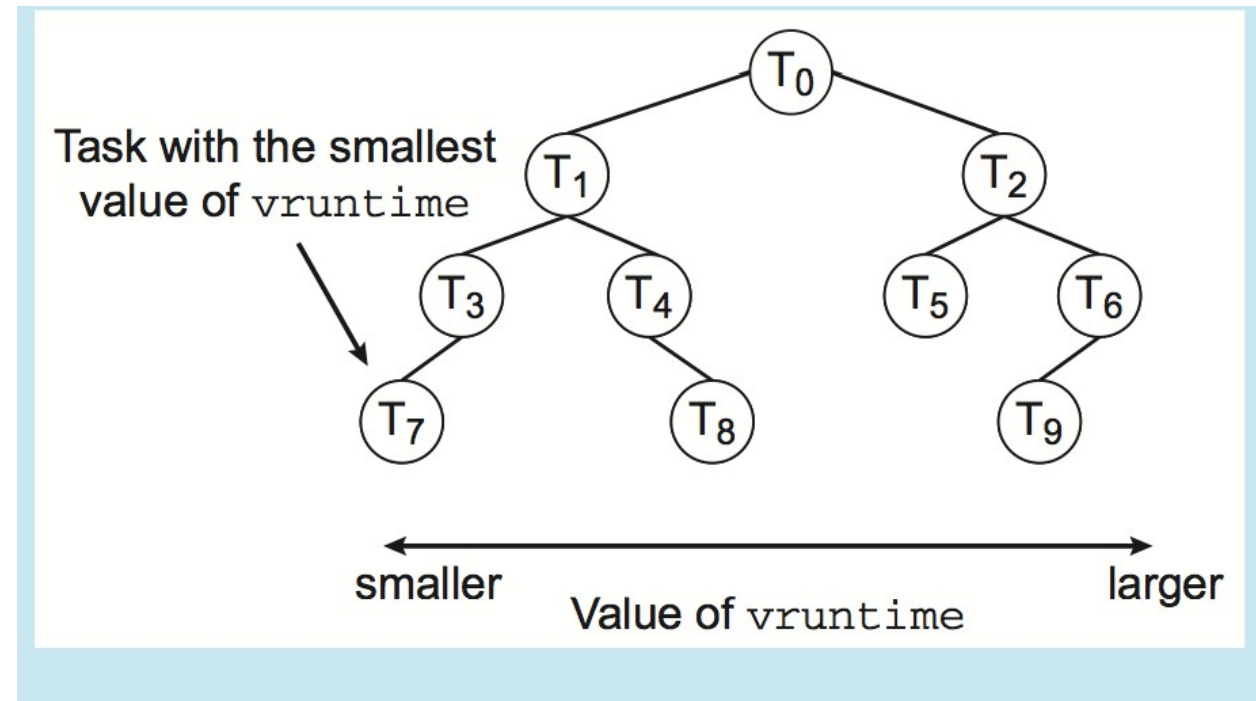
Week 3 develops the synchronization mechanisms behind this situation.

Linux fair scheduling: historical context

The older CFS illustration orders tasks using virtual runtime to explain proportional fairness. Treat this as historical background.

Current kernel documentation describes the transition toward EEVDF beginning with Linux 6.6.

The installed kernel version and distribution configuration matter when explaining an actual Ubuntu observation.



Virtual deadlines and real deadlines

EEVDF selects eligible fair-class work using virtual deadlines and service accounting. These are fairness/latency mechanisms.

Linux SCHED_DEADLINE is a separate policy using EDF with bandwidth reservation machinery.

An EEVDF virtual deadline is not an application's promise to finish by a wall-clock deadline.

Nice values for normal scheduling range from -20 to $+19$. They influence relative service, not a hard completion deadline.

SCHED_FIFO and SCHED_RR use static real-time priorities. RR time-slices peers at the same priority.

Our observation activity reads policy and affinity. It does not require enabling privileged real-time workloads.

Ubuntu observation: record the environment

Record `uname -r`, `lscpu` and the installed versions of the tools. In a VM, also record the virtual CPU allocation.

For your shell, inspect: `ps -o pid,cls,ni,psr,comm -p "$$"`

Read its policy with `chrt -p "$$"` and allowed CPUs with `taskset -pc "$$"`.

PSR is a sampled CPU identifier, not proof of continuous execution or an exclusive reservation.

The reading supplies a bounded CPU-work program and a same-CPU comparison using ordinary nice values 0 and 10.

Predict how contention changes elapsed time while CPU demand stays similar. Record actual CPU time, elapsed time and repeated runs.

A single result cannot identify the scheduling algorithm. VM load, affinity and other runnable work affect the observation.

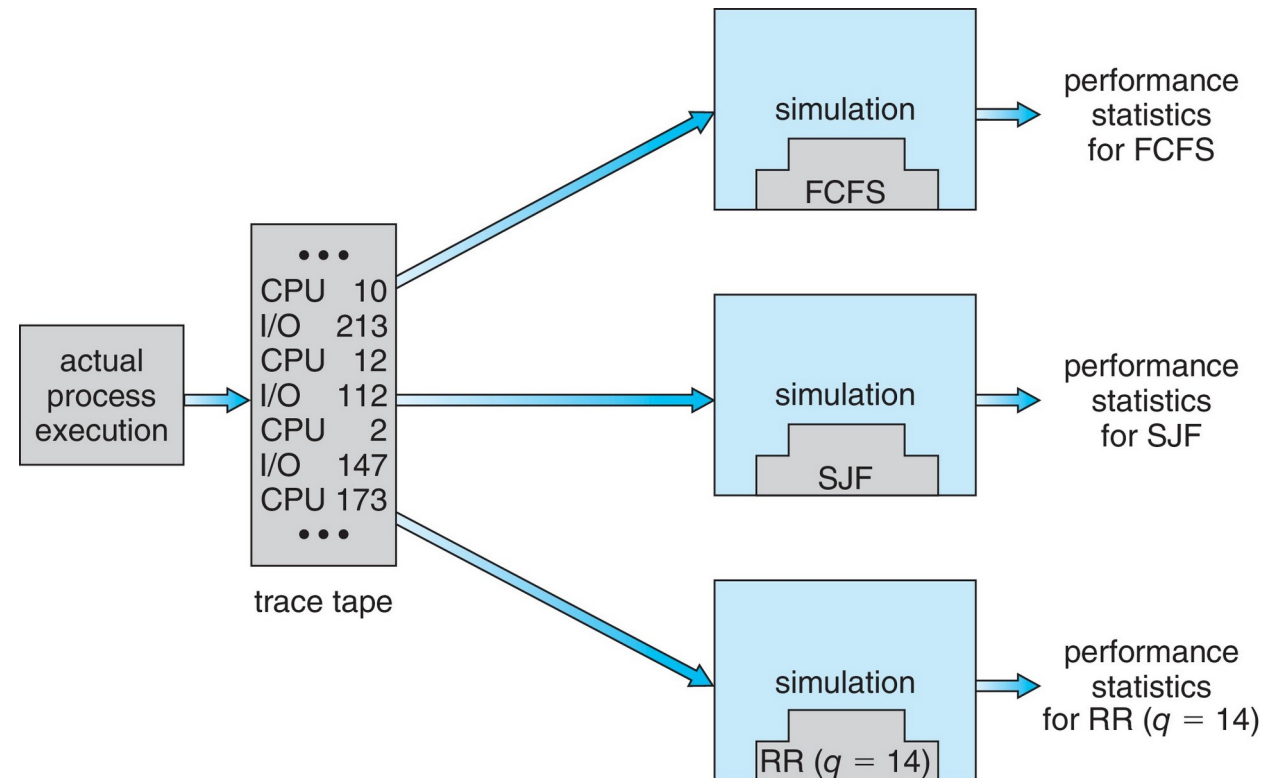
This is optional practice with AI assistance permitted. Assignment deadlines and submission rules are separate.

Evaluating a scheduling policy

Deterministic timelines answer a precise question for a stated workload. They are easy to check but narrow in scope.

A simulation varies arrivals and burst distributions. Its output depends on the model and input traces.

Measurements include real overhead and interference but require reproducible conditions. Compare more than the mean.



Little's law and measurement boundaries

For a stable queue with consistent boundaries, average number waiting $L_q = \text{arrival rate } \lambda \times \text{average queue wait } W_q$.

If $\lambda = 7$ jobs/s and $L_q = 14$ jobs, $W_q = 2$ s. Units expose the meaning of the result.

If the count includes running work, use time in the system rather than queue-only waiting.

An overloaded queue that grows without bound does not provide the steady-state averages this calculation assumes.

Worked review: identify the hidden assumption

“SJF is always best.” Best according to which criterion, with what arrival information and burst knowledge?

“ $U < 1$ proves RM deadlines.” False: Example B has $U = 0.9375$ and misses at 80.

“Pinning reserves a CPU.” False: affinity restricts placement but permits other work on the same CPU.

“EDF and EEVDF mean the same guarantee.” False: application deadlines and virtual service deadlines serve different purposes.

Rebuild each timeline before checking its answer. Keep arrivals, remaining demand, queue order and deadlines visible.

OSC 10e, Chapter 5: §§5.1–5.3 for policies, §5.5 for multicore, §5.6 for real time, §5.8 for evaluation.

Read the Week 2 illustrated chapter for complete calculations, practice and worked answers. Next: shared state and synchronization.

Adapted from Silberschatz, Galvin and Gagne's official 10e slides and Mehmet Göktürk's scheduling slides. Figures retain their source credits.