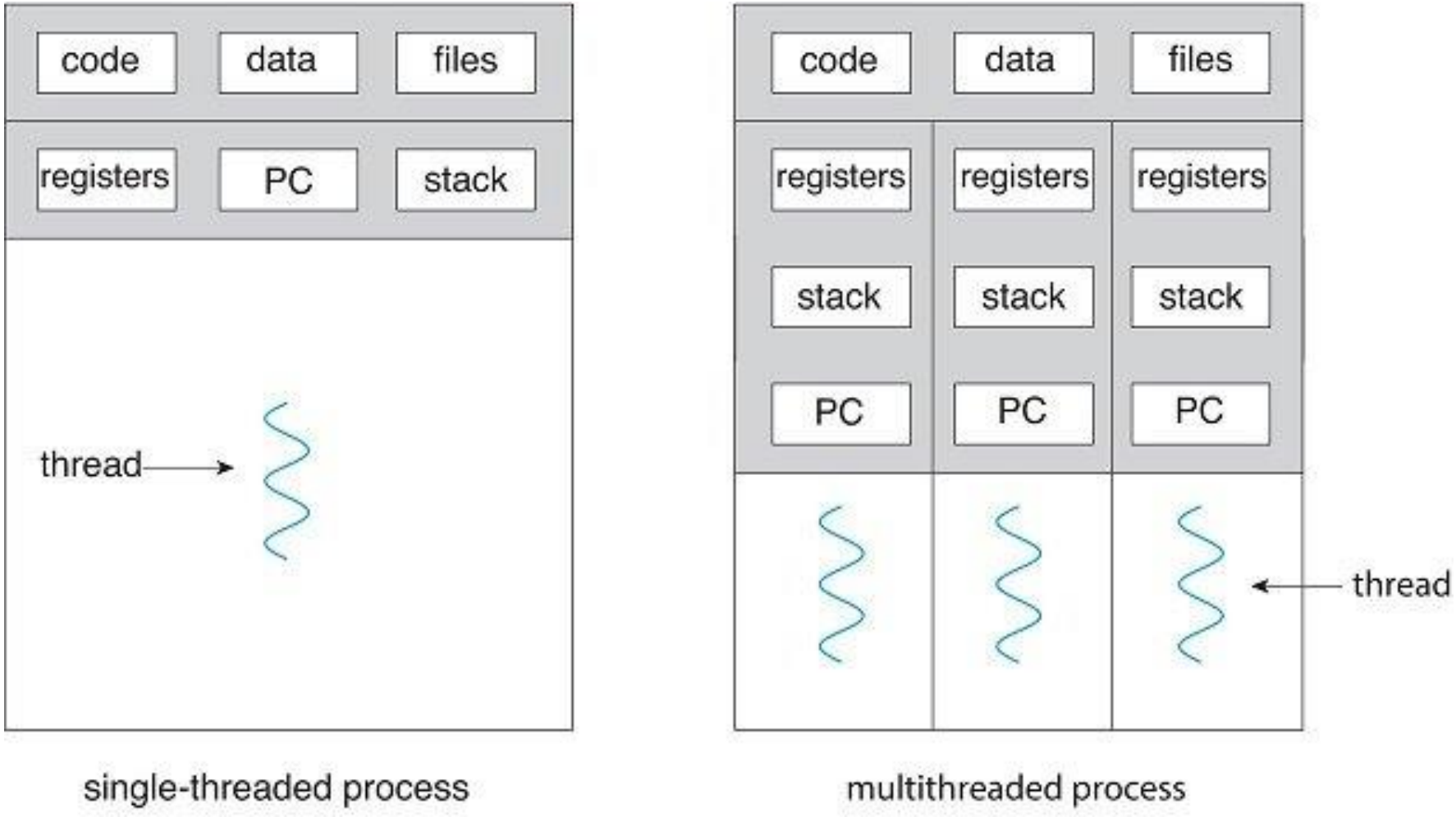
Computer and Operating Systems (COS)

Lecture 8

Overview of the contents

- **Threads & Concurrency**
  - **Multicore programming**
  - **Multithreading models**
- **CPU Scheduling**
  - **Scheduling criteria**
  - **Scheduling algorithms**
  - **Multiple-processor scheduling**
  - **Real-time CPU scheduling**
  - **Algorithm evalution**

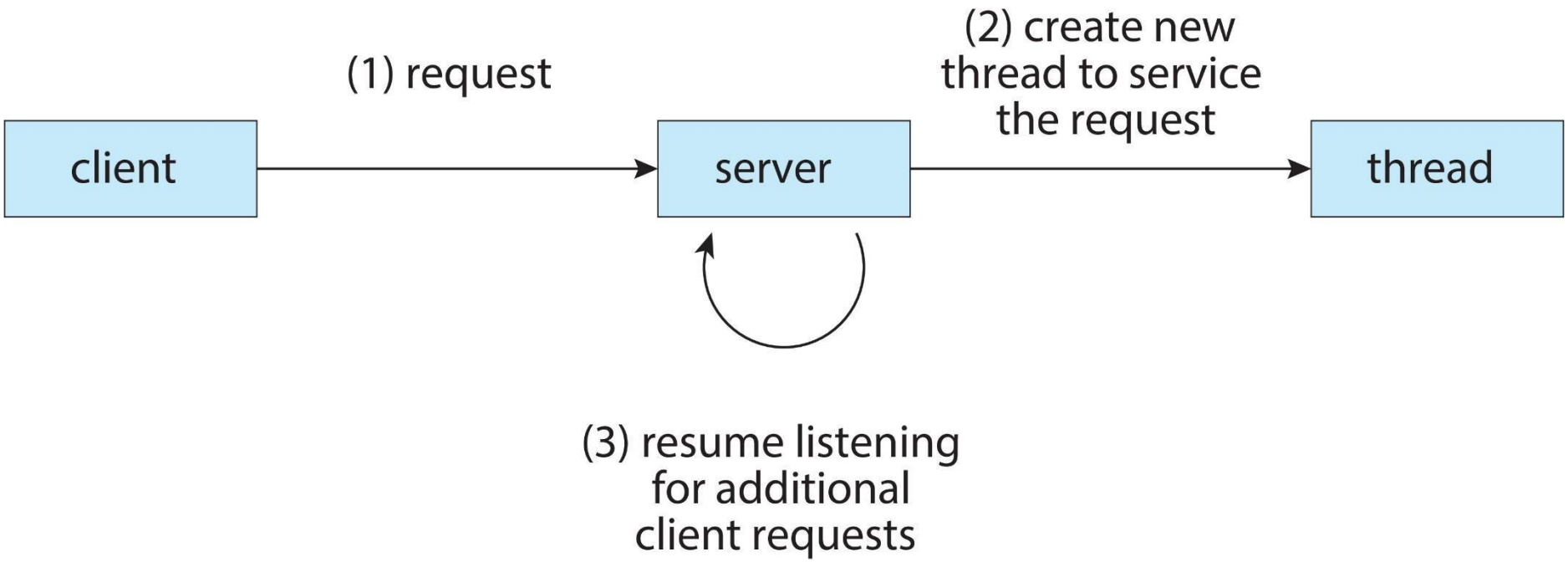
Multithreaded Programming



single-threaded process

multithreaded process

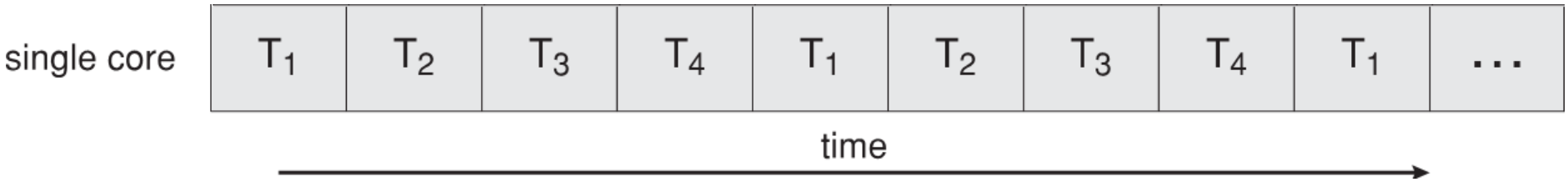
Multithreaded Server



Responsiveness:
Faster response/reaction
Resource sharing:
Provides a good opportunity for threads to collaborate on tasks.
Economy:
Since threads are in the same address space, it is more efficient to schedule them.
Scalability:
Threads can be run on different CPU cores in parallel.

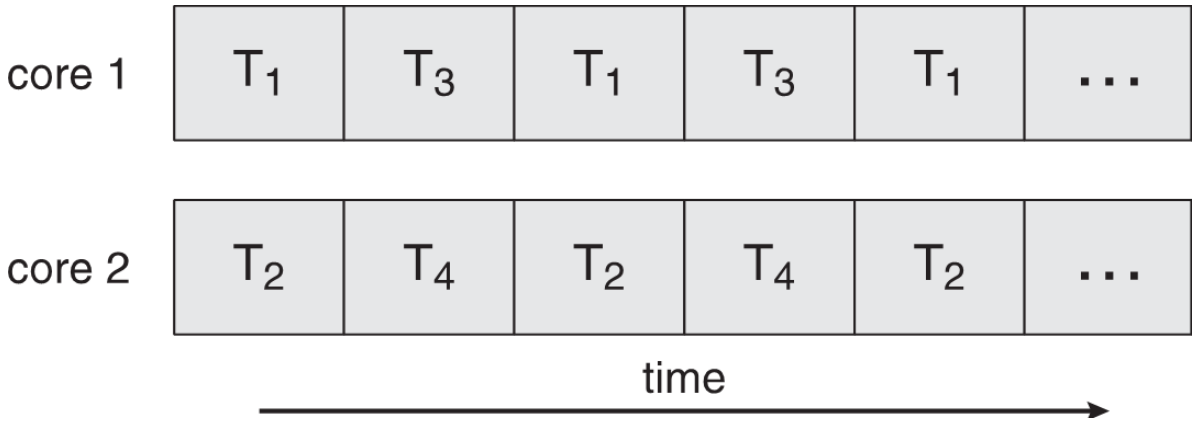
Concurrent execution on a single-core system - concurrency

rapidly switching between processes thereby allowing each process to make progress

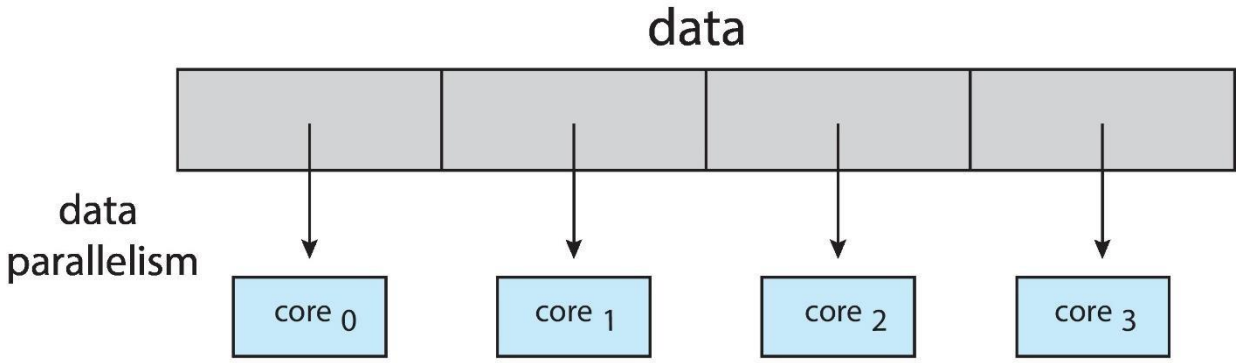


Parallel execution on a multicore system - Parallelism

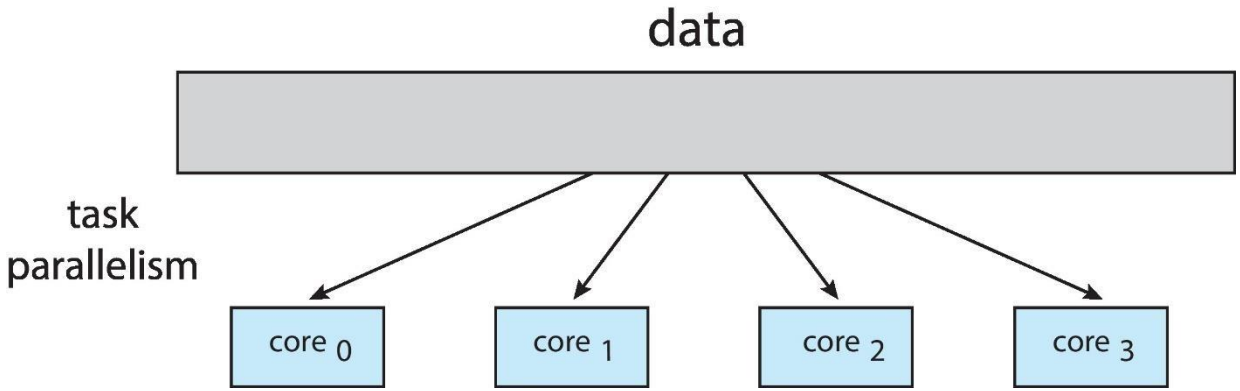
Multiple threads run in parallel on multiple cores



Two types of parallelism



Data parallelism: the calculation of a dataset is distributed over several cores. E.g. Arrays



Task parallelism tasks, not data, are distributed across multiple cores. Each task performs a unique operation.

Amdahl's Law

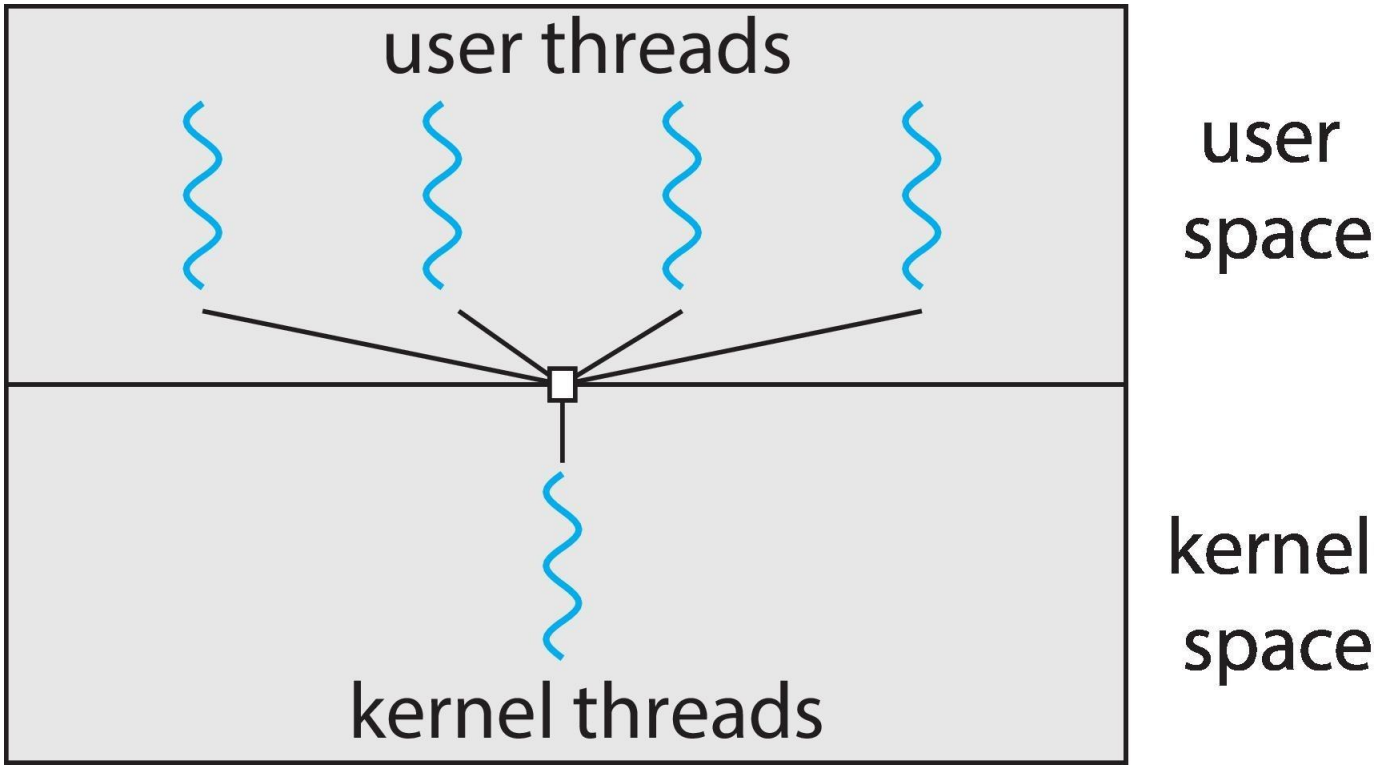
$$speedup \leq \frac{1}{S + \frac{(1-S)}{N}}$$

S is the portion of the application that must be performed serially on a system
(1-S) is is the portion of the application that can be performed in parallel on a system
N is the number of CPU cores

Challenges of multicore programming

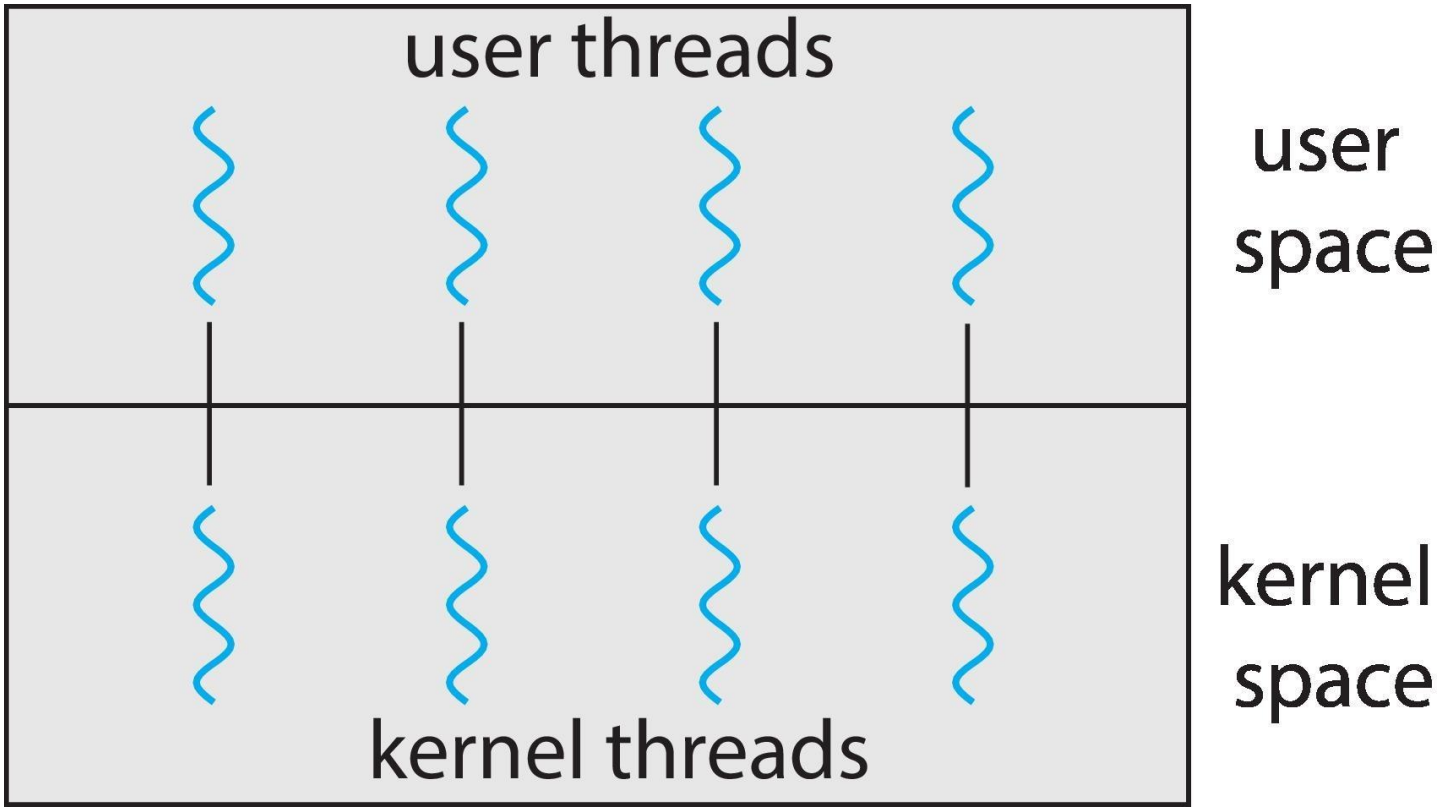
1) **Identifying tasks**. This involves examining applications to find areas that can be divided into separate, concurrent tasks.
2) **Balance**. While identifying tasks that can run in parallel, programmers must also ensure that the tasks perform equal work of equal value.
3) **Data splitting**. Just as applications are divided into separate tasks, the data accessed and manipulated by the tasks must be divided to run on separate cores.
4) **Data dependency**. The data accessed by the tasks must be examined for dependencies between two or more tasks. When one task depends on data from another, synchronization must be ensured.
5) **Testing and debugging**. concurrent programs are inherently more difficult than single-threaded applications in terms of testing and debugging.

Multithreading Models



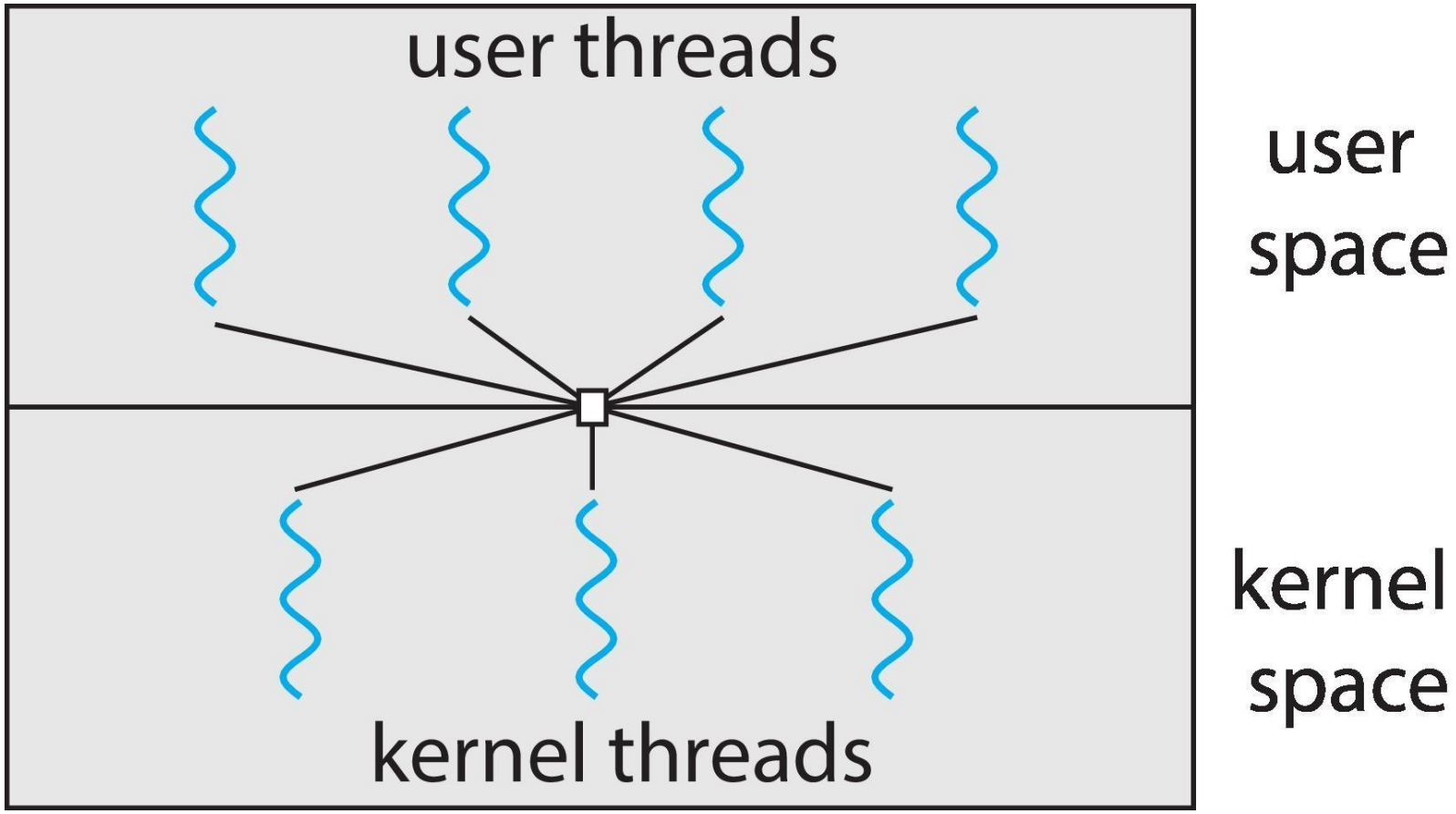
Many-to-one model

Multithreading Models



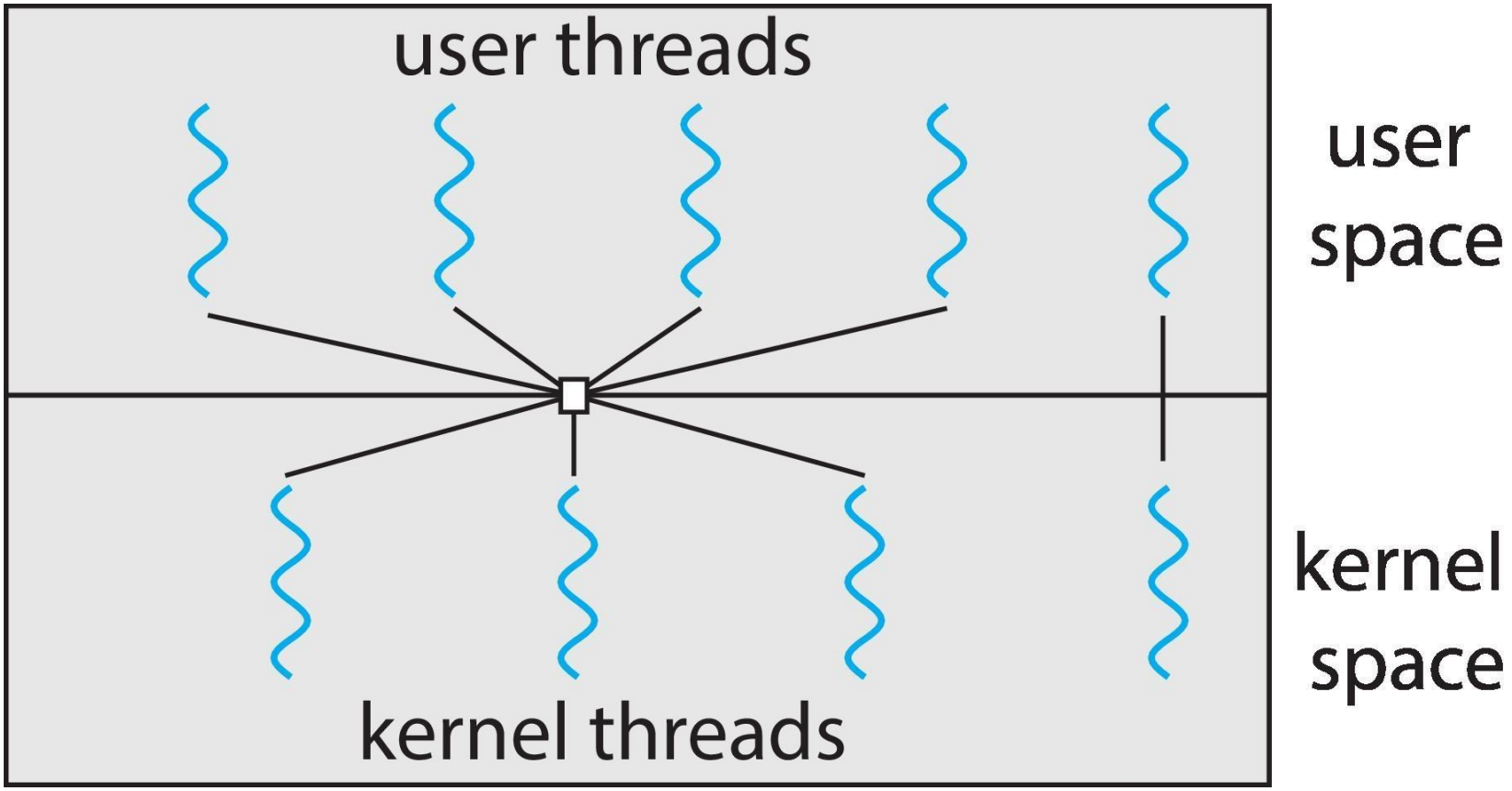
One-to-One model

Multithreading Models



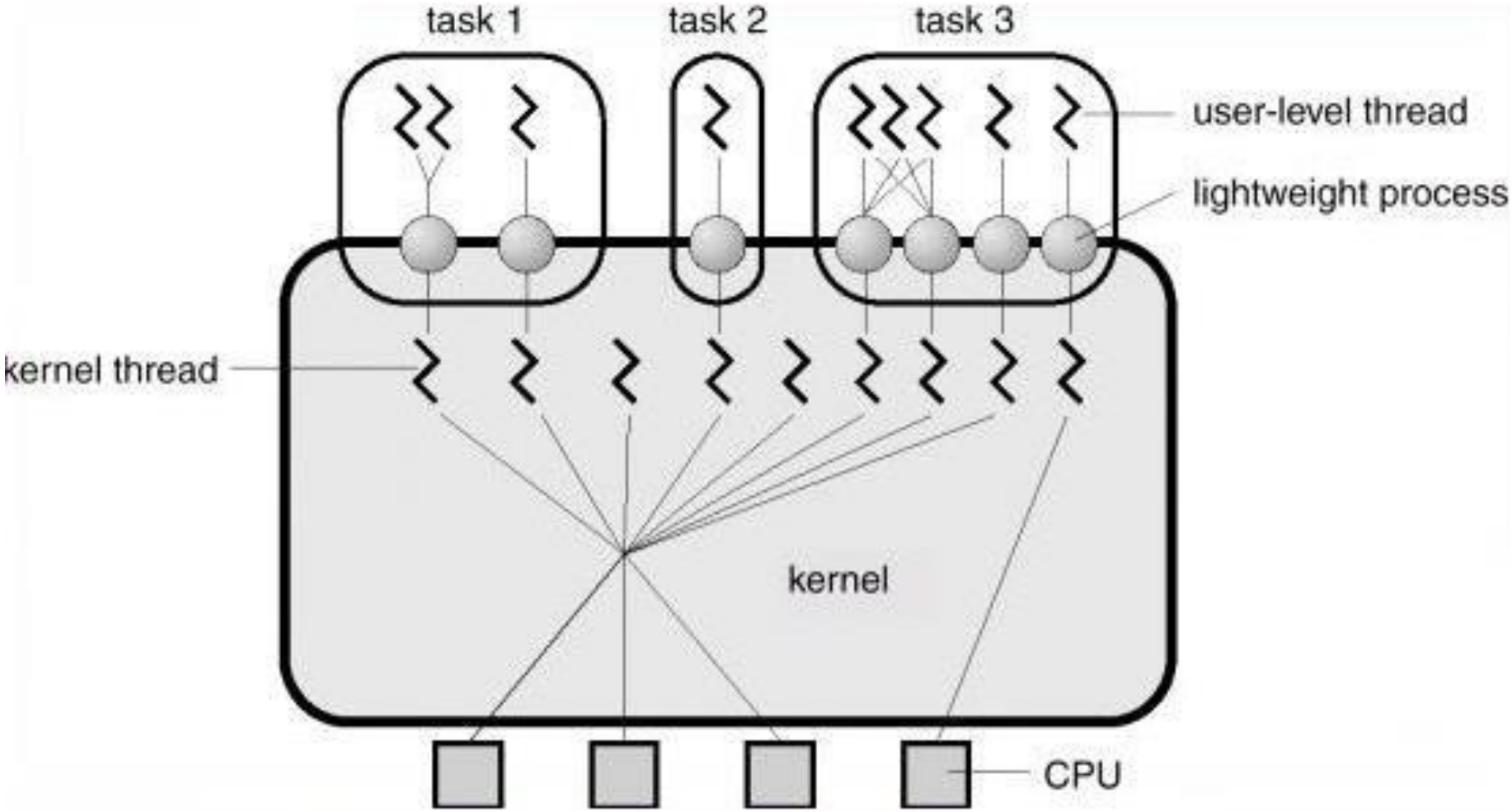
Many-to-Many model

Multithreading Models



Two-level model (besides Many-to-Many, a thread can be bound to a specific kernel thread)

Example: Solaris two-level model



Notice:

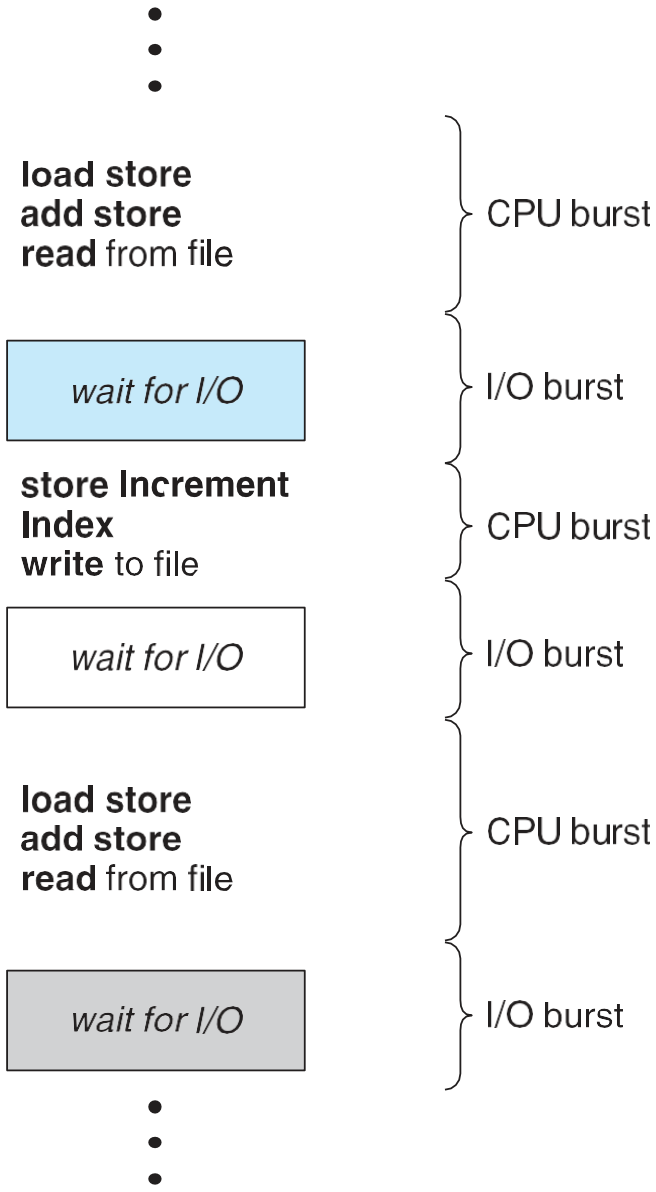
It is approx. 30 times faster to create a thread than a task (process)

It is approx. 5 times faster to schedule a thread than a task (process)

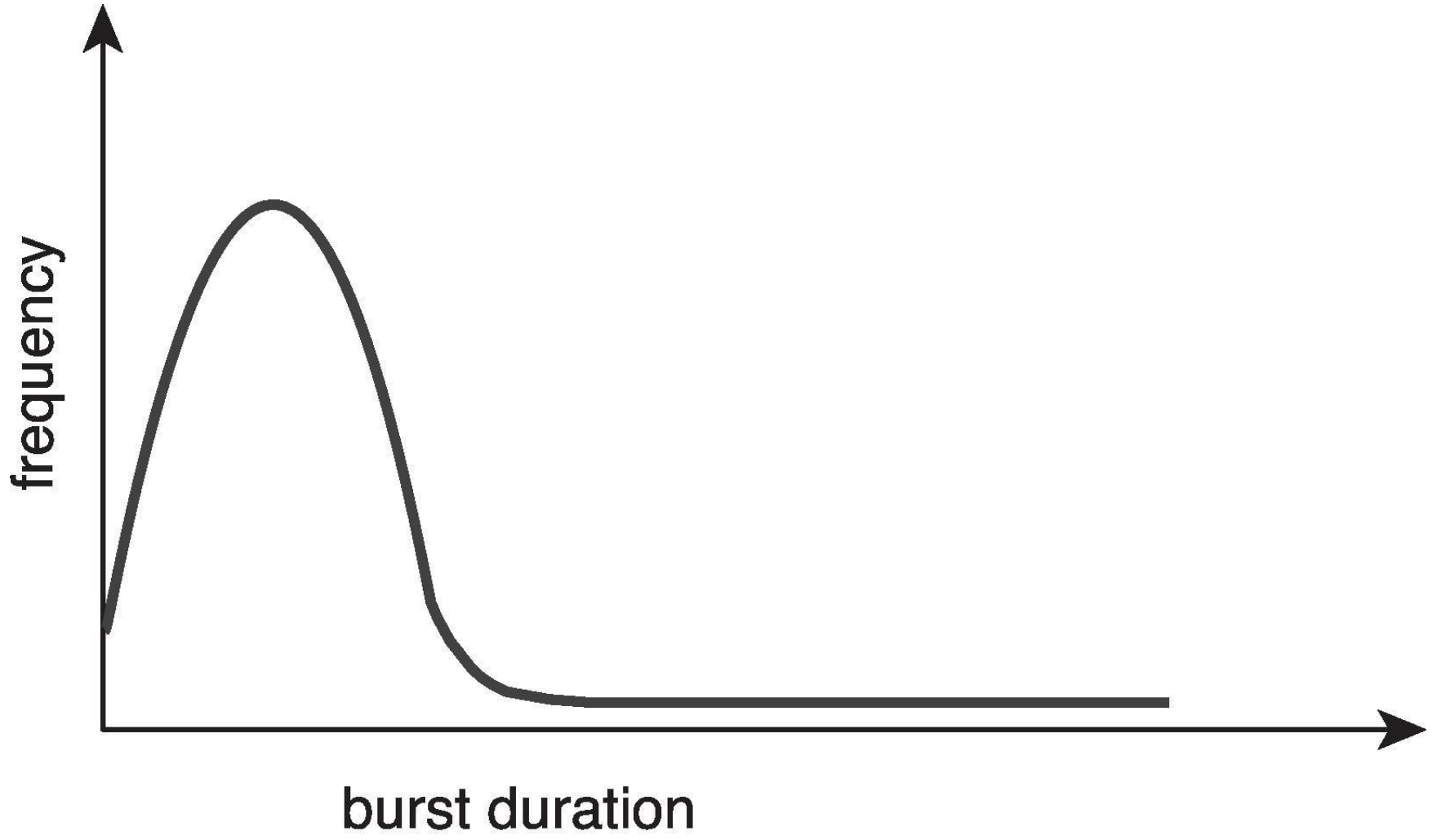
User threads are scheduled in the user-space by the application itself, support for creation, etc. can possibly obtained by the kernel through libraries in user-space

CPU Scheduling

Basic concept



CPU-burst histogram



Scheduling Parameters (Design Objectives)

When designing a scheduling algorithm, there can be different goals to aim for:

CPU utilization: it is important to make the best possible use of the CPU, i.e., high utilization. In real systems, the utilization will be between 40% - 90%.

Throughput: the number of user processes that are executed per unit of time, often sec. This is a measure of productivity.

Turnaround time: the lifespan of the individual user processes is measured from the time they are allocated resources and access is given to the system until their resources are released and they leave the system. That is, the sum of waiting times in all queues, CPU time and doing I/O.

Waiting time: only the sum of waiting times in the ready queue is measured during the lifetime of a process.

Response time: the time that elapses from a request being registered until a response is started is measured.

Variance in response time: The aim here is to have as little variance as possible in the response time. A large variance gives the user the impression that the system is unstable, although this is not the case.

First-Come, First-Served (FCFS) Scheduling

Process	Burst Time
P_1	24
P_2	3
P_3	3

Waiting time:

$P_1 = 0$
$P_2 = 24$
$P_3 = 27$

Average Waiting time:

$$P_A = \frac{0 + 24 + 27}{3} = 17$$

First-Come, First-Served (FCFS) Scheduling

If the three processes arrive in a different order…

Waiting time:

$P_1 = 6$
$P_2 = 0$
$P_3 = 3$

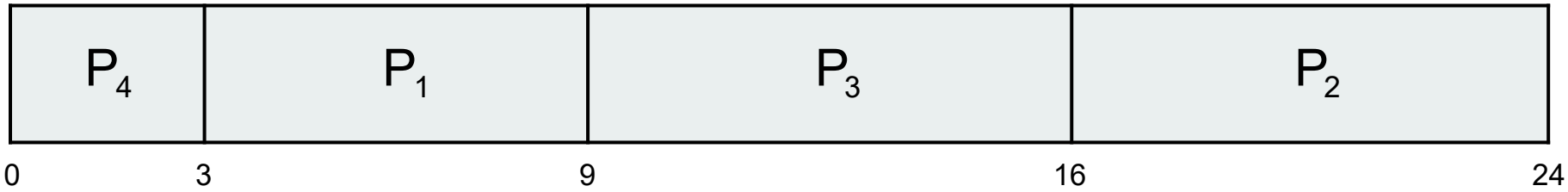
Average Wating time:

$$P_A = \frac{6 + 0 + 3}{3} = 3$$

Convoy effect: all the other processes wait for the one big process to get off the CPU.

Shortest-Job-First (SJF) Scheduling

Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3



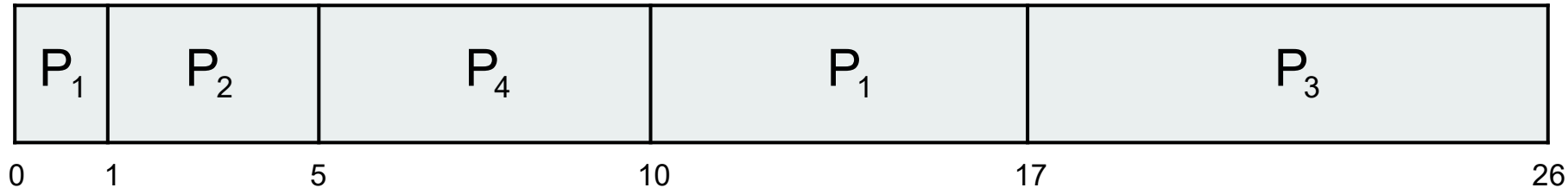
Average Waiting time:

$$P_A = \frac{3 + 16 + 9 + 0}{4} = 7$$

Non-Preemptive scheduling

Shortest-Job-First (SJF) Scheduling

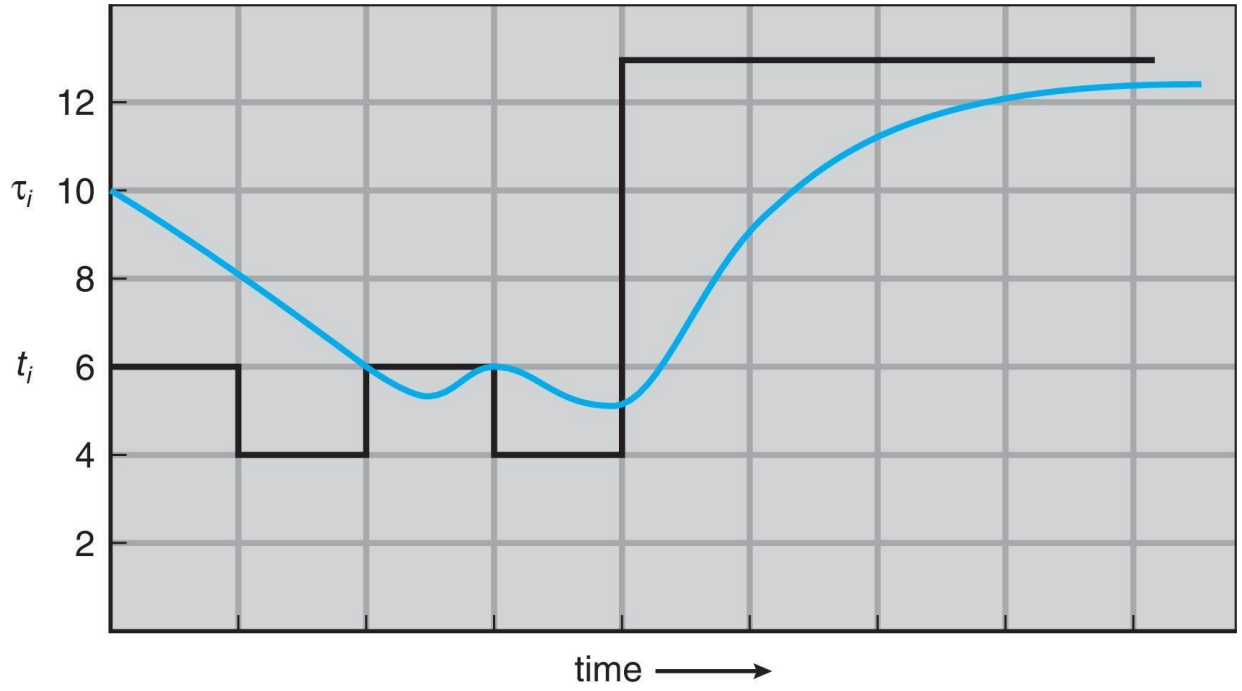
Process	*Arrival* Time	Burst Time
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5



Average Waiting time:

$$P_A = \frac{(10-0)+(1-1)+(17-2)+(5-3)}{4} = 6.75$$

Preemptive scheduling



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

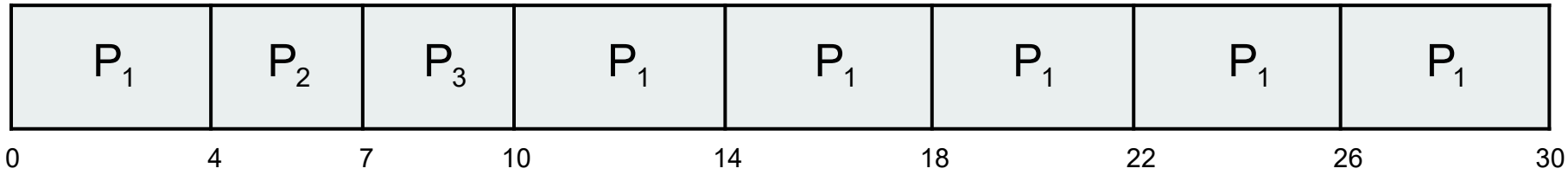
$$\tau_{n+1} = \alpha \cdot t_n + (1-\alpha) \cdot \alpha \cdot t_{n-1} + \cdots + (1-\alpha)^j \cdot \alpha \cdot t_{n-j} + \cdots + (1-\alpha)^{n+1} \cdot \tau_0$$

τ_0 is the start guess
α is the relative weight of recent and past history in the prediction.

You can predict something about the near future if you know the present and the past (from the present to the past, smaller weights will be used)

Round-Robin (RR) Scheduling

Process	Burst Time
P_1	24
P_2	3
P_3	3



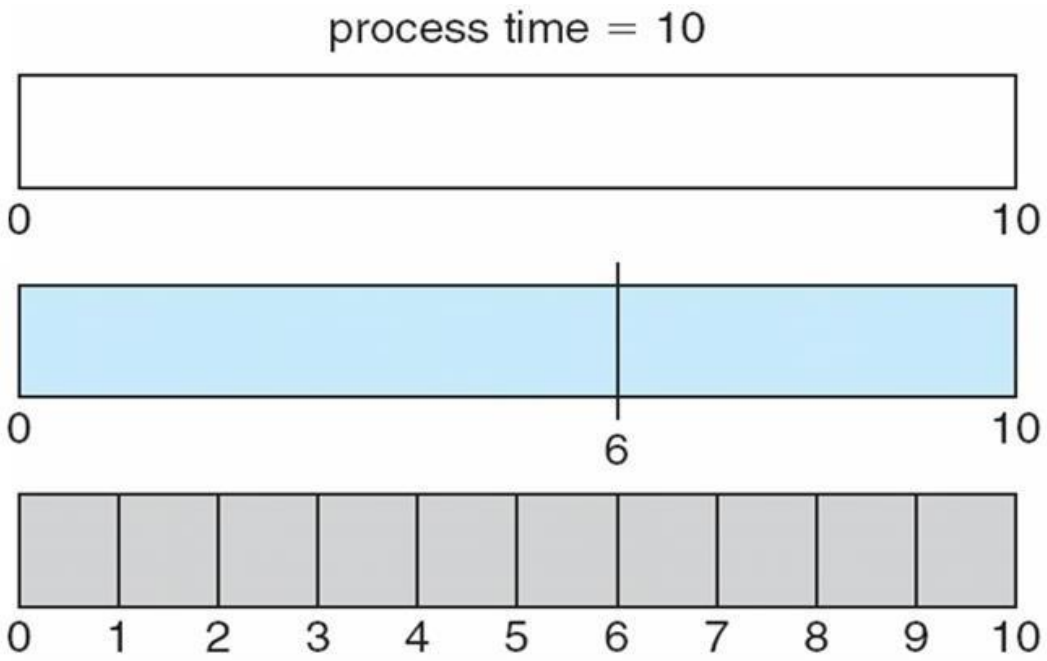
Time quantum or time slice = 4

Average Waiting time:

$$P_A = \frac{(10-4)+4+7}{3} = 5.66$$

If time slice goes towards infinity, then this algorithm becomes FCFS

Effect of the size of time slice on context switches

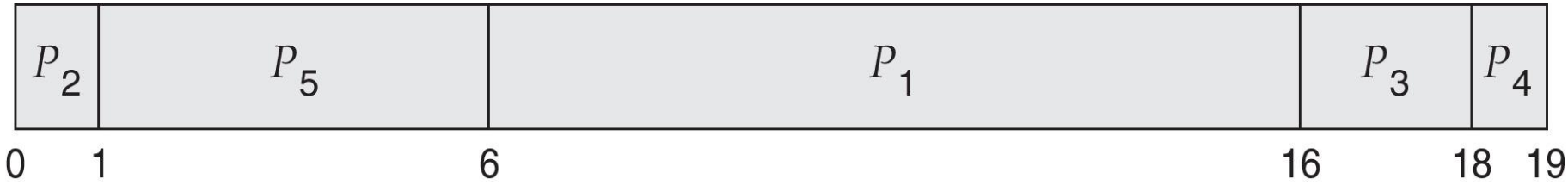


quantum	context switches
12	0
6	1
1	9

A good rule of thumb is that 80% of CPU bursts should be shorter than the time quantum.

Priority Scheduling

Process	Burst Time	Priority
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

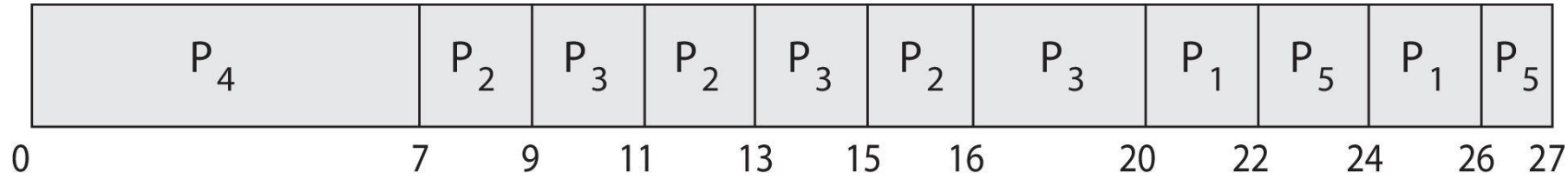


Average Wating time: 8.2

Be aware that starvation problems can occur here!
(aging is a solution to this)

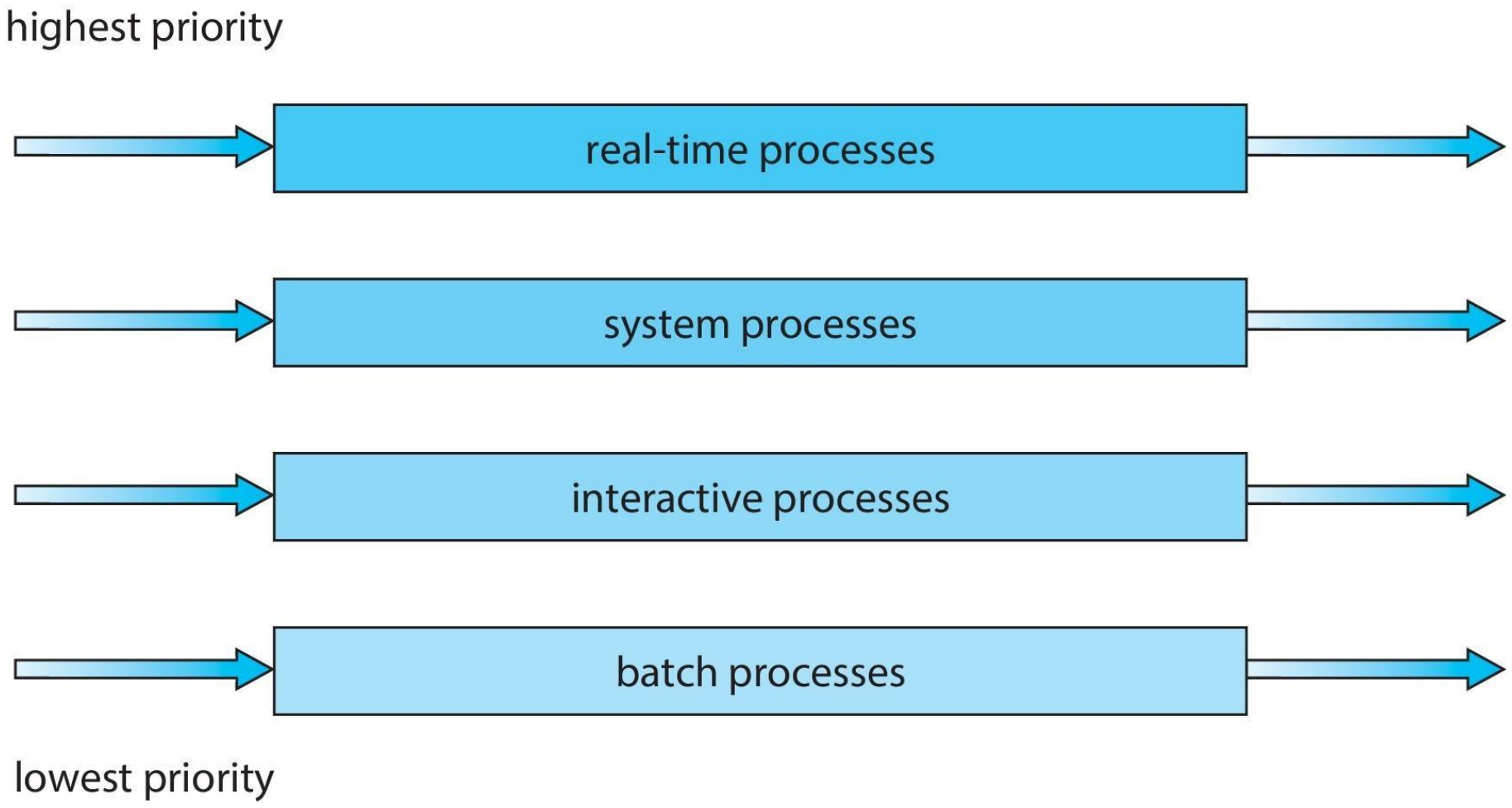
Priority Scheduling with Round-Robin

Process	Burst Time	Priority
P_1	4	3
P_2	5	2
P_3	8	2
P_4	7	1
P_5	3	3



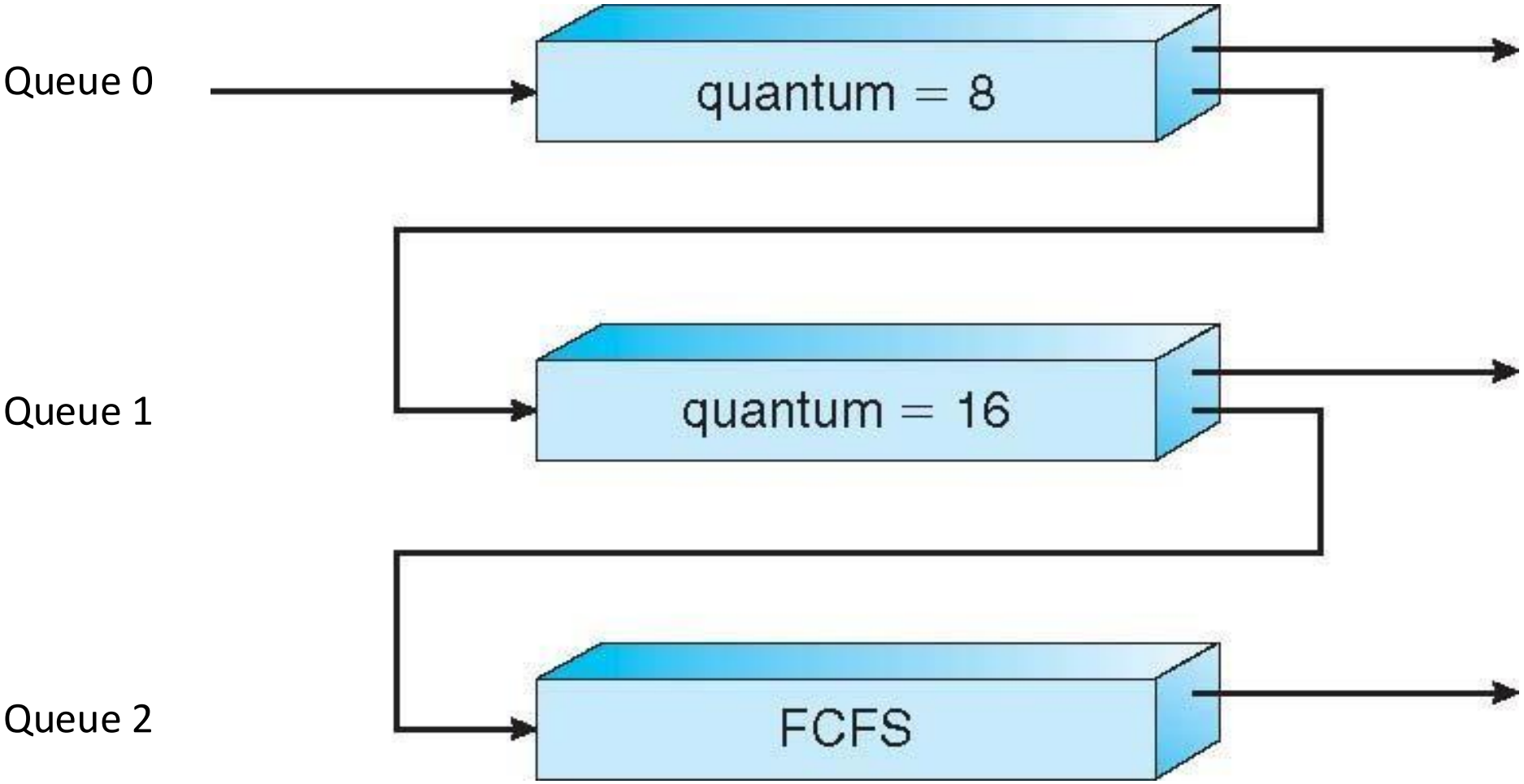
First, priorities are used to select the processes. But if several processes have the same priority, then Round-Robin scheduling is used.

Multilevel Queue Scheduling



e.g., Windows (32 different queues)

Multilevel Feedback Queue Scheduling



Multi-processor scheduling

- CPU scheduling obviously becomes more complex when there are multiple CPUs in the system.

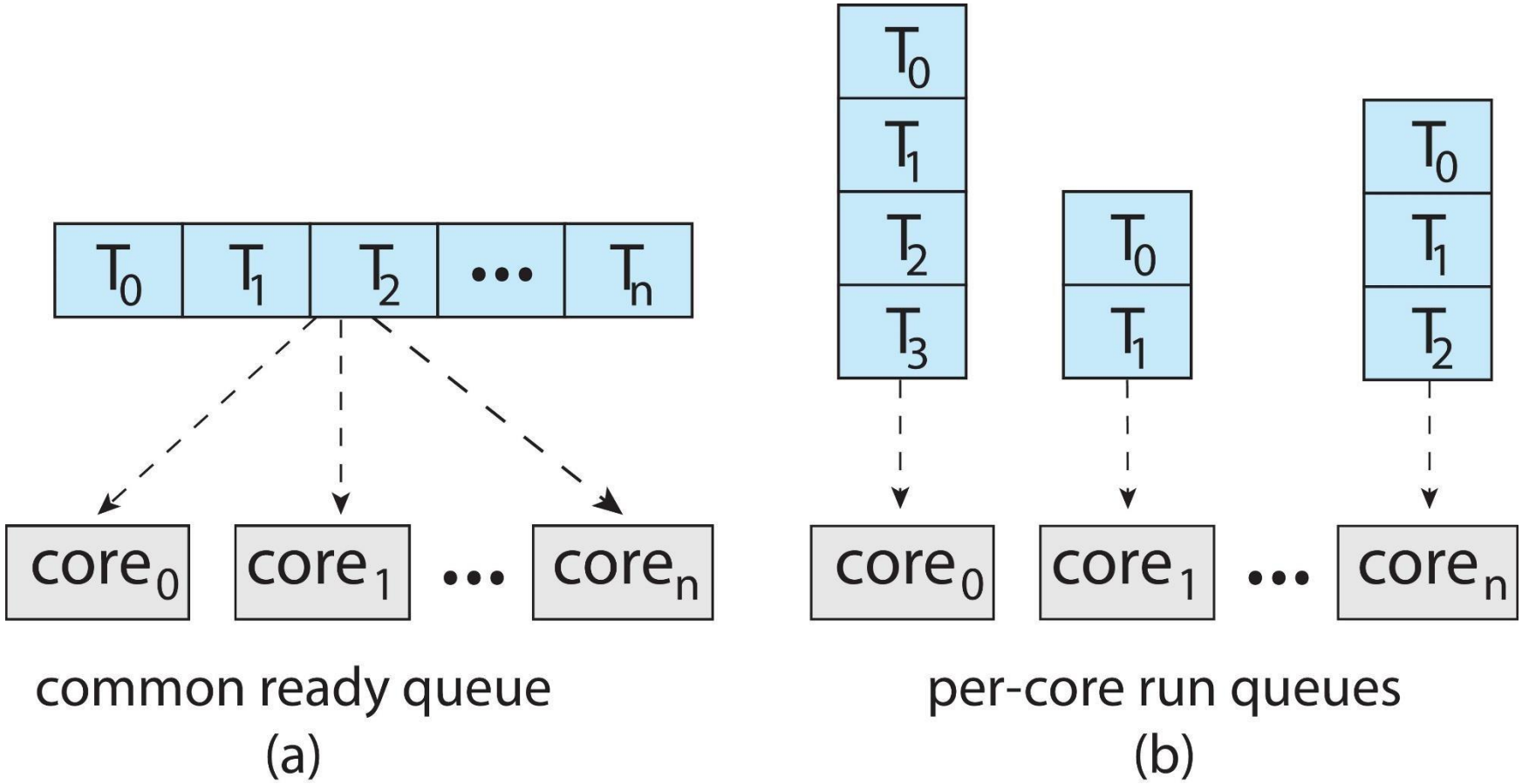
Two approaches

- **Asymmetric multiprocessing -** Here it is a processor that does the scheduling, I/O processing, etc., i.e., system administration. Other processors are responsible for only user code
- **Symmetric multiprocessing (SMP) -** Each processor is self-scheduling but can either have a common ready queue or its own private ready queue.
  To achieve balanced workload in such systems, two approaches can be used:
  - **Push migration** - here an administrative process periodically checks the load on each processor and evenly distribute the load by moving or pushing threads from overloaded to idle processors.
  - **Pull migration** - here idle processors pull processes from the ready queues of busy processors.
  - combination of both strategies often occurs in real systems

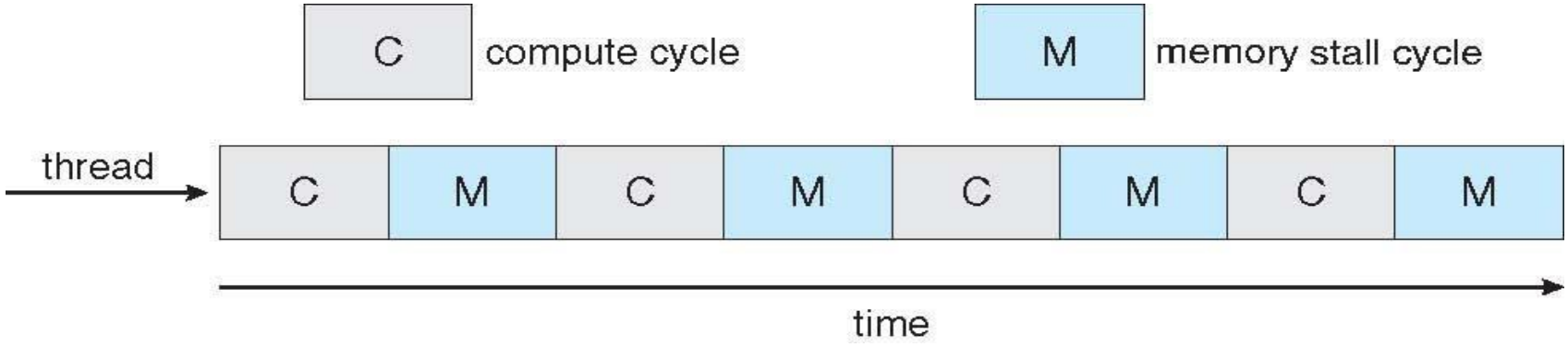
Processor Affinity – a process has an affinity for the processor on which it is currently running, as it is costly to move it to another processor (due to cache, etc.).

- **Soft connection** – here you try to keep the process on a processor, but no guarantee.
- **Hard connection** – here the process is guaranteed to run on the designated processor.
- A variant allows the process to specify a set of processors on which it can run.

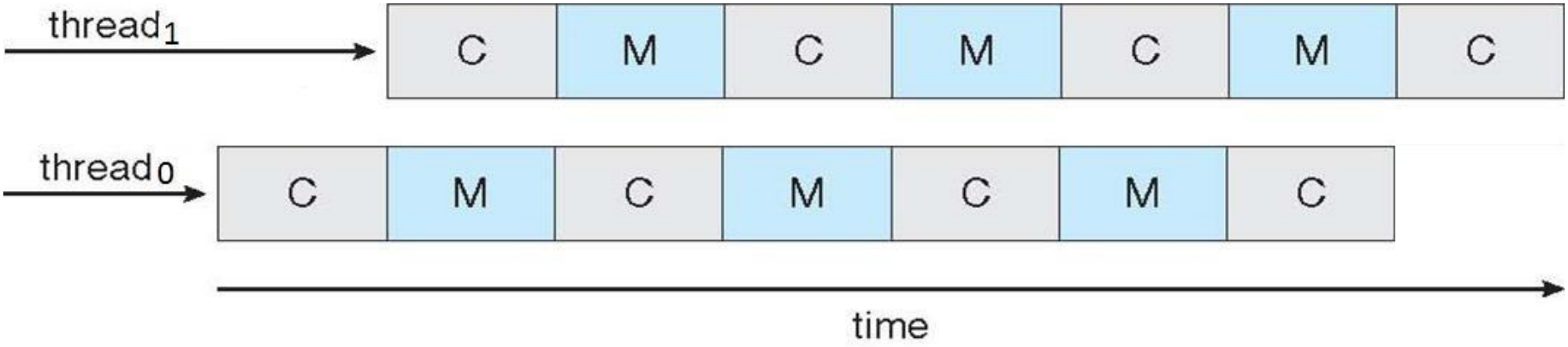
Multi-processor scheduling (SMP)



Memory stall

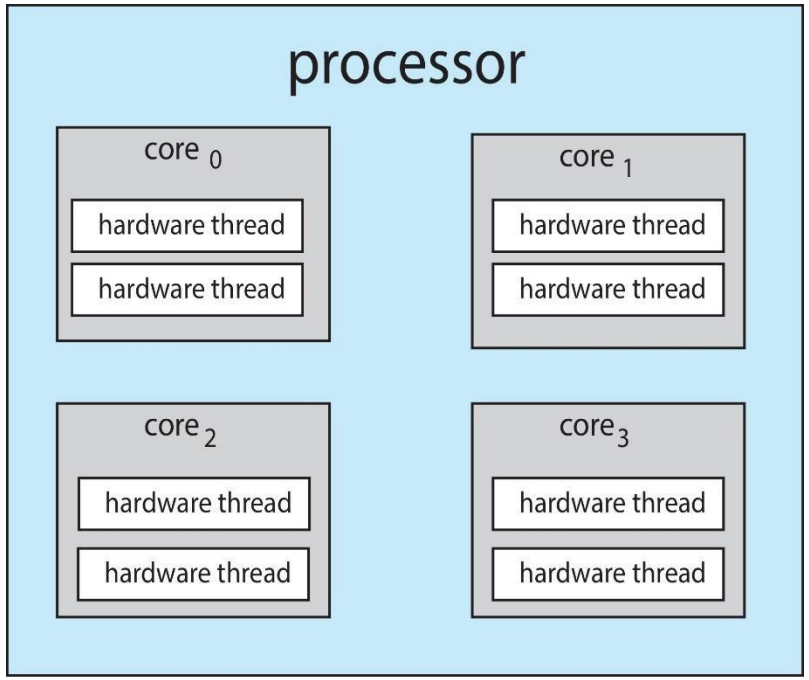


Multithreaded multicore system

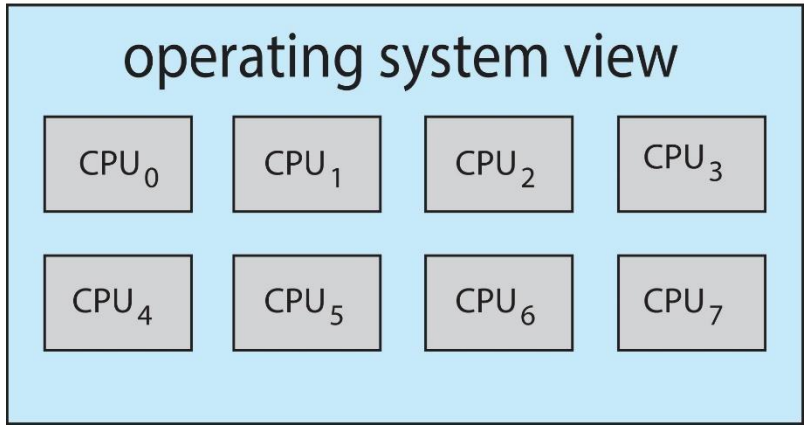


Multithread Multicore system

Physical



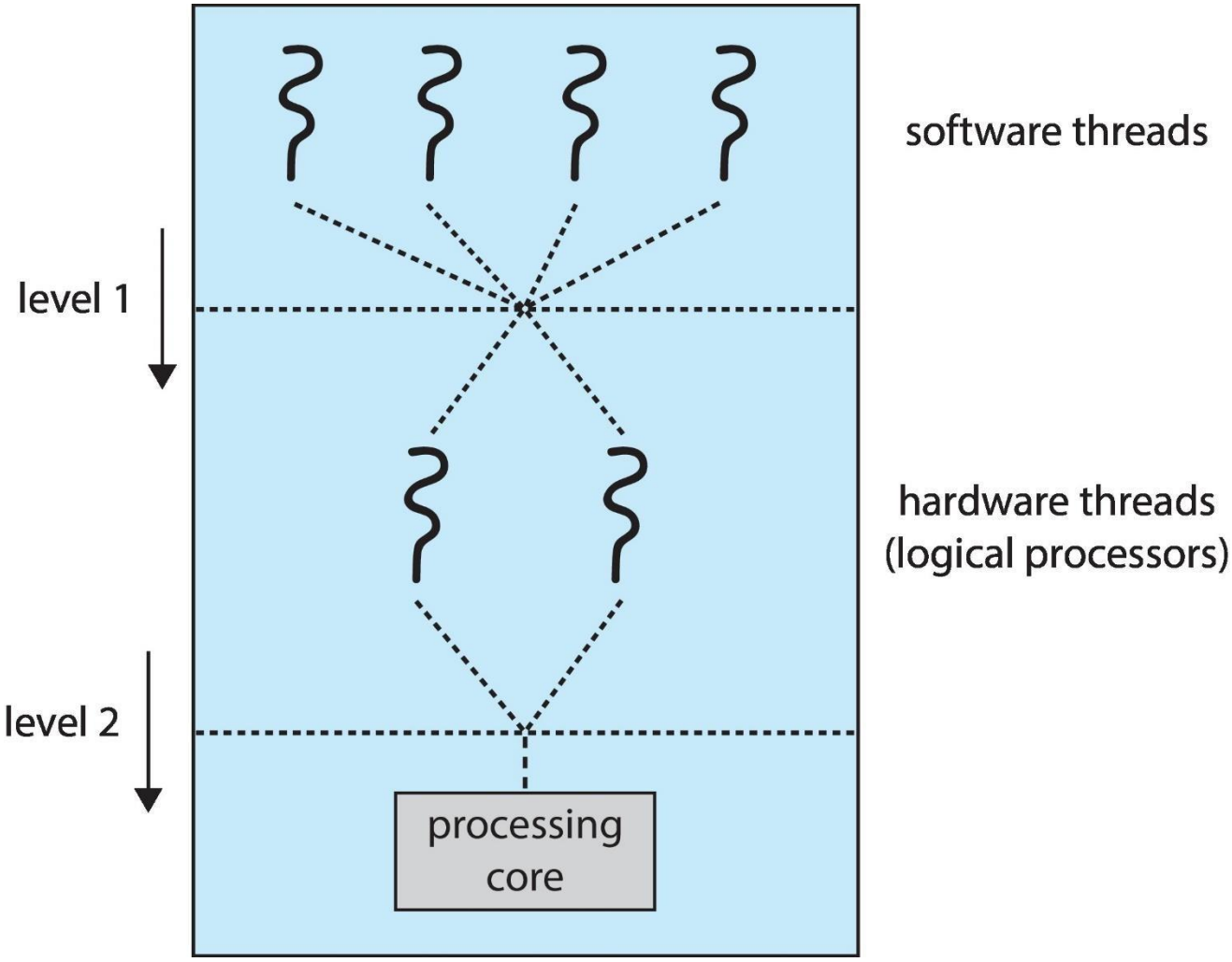
Logical



Example:

- **UltraSPARC T3**: It has **16 physical cores**, and **8 hardware threads per core**.
- **Intel Itanium**: It has **2 physical cores** (dual-core), and **2 hardware threads per core**.

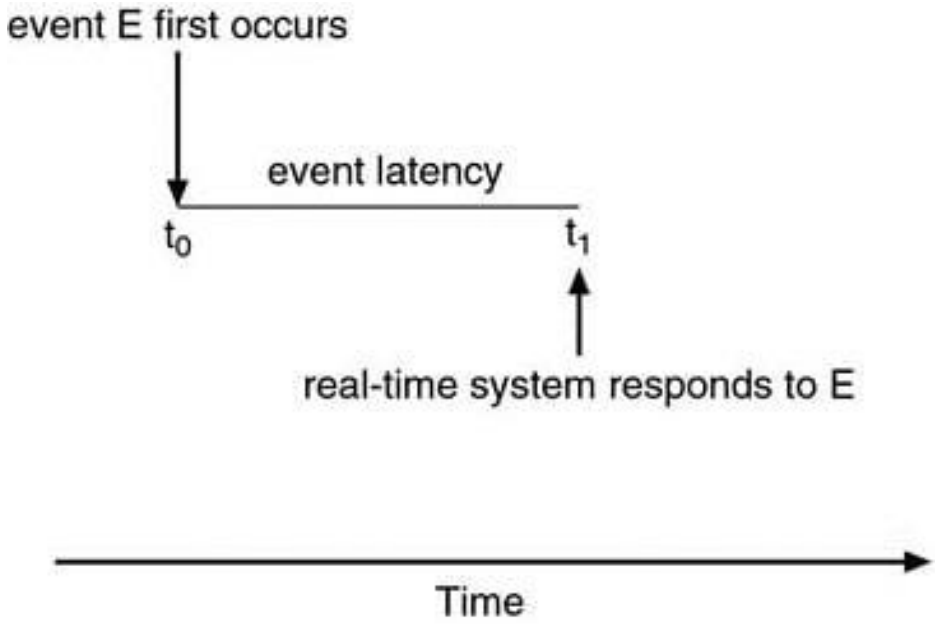
Two levels of scheduling



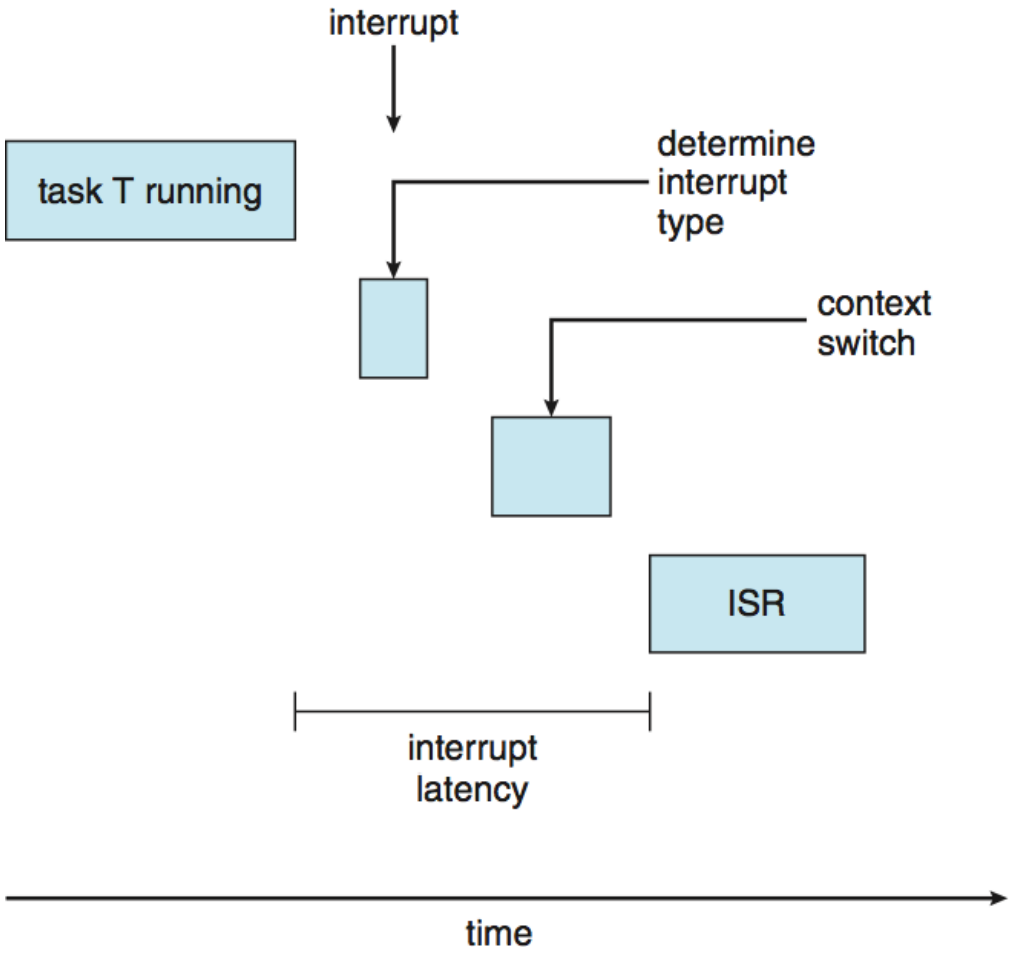
- Software threads are scheduled by software (kernel threads of OS and user threads of processes, via thread library)
- A hardware thread is the physical place where a software thread runs
- The physical CPU determines which hardware thread to run.

Real-time CPU Scheduling

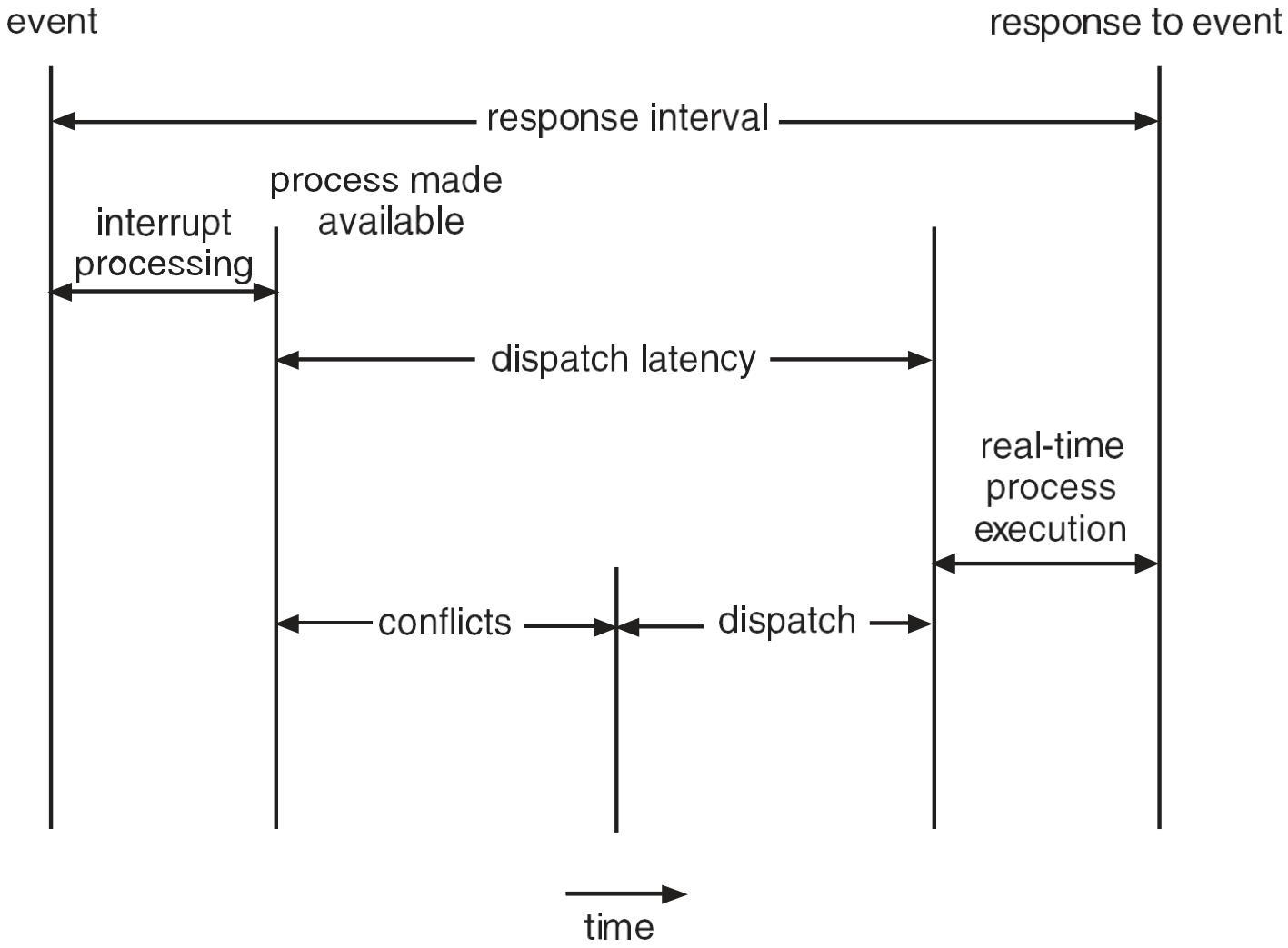
- **Soft real-time system**: provide no guarantee as to when a critical real-time process will be scheduled. They guarantee only that the process will be given preference over noncritical processes.
- **Hard real-time system**: have stricter requirements. A task must be serviced by its deadline; service after the deadline has expired is the same as no service at all.



Real-time CPU Scheduling



Real-time CPU Scheduling



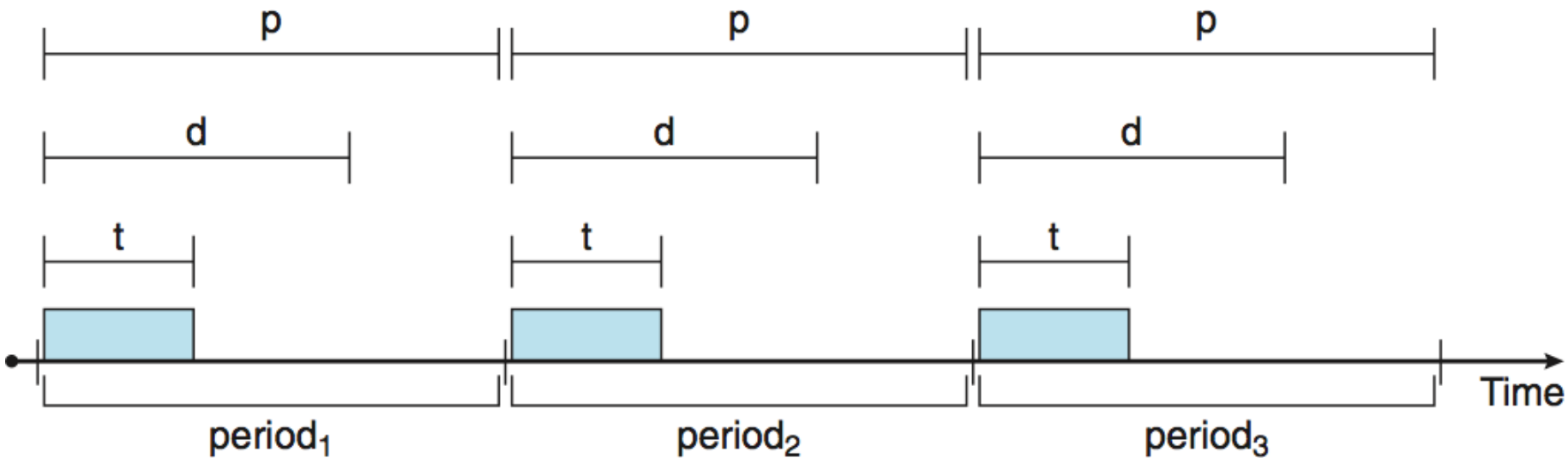
In the conflict phase

1. Preemption/interruption of any process running in the kernel.
2. Release resources from low-priority processes if they are to be used by high-priority processes

Priority-Based Real-Time Scheduling

Priority-based scheduler only guarantees soft real-time functionality

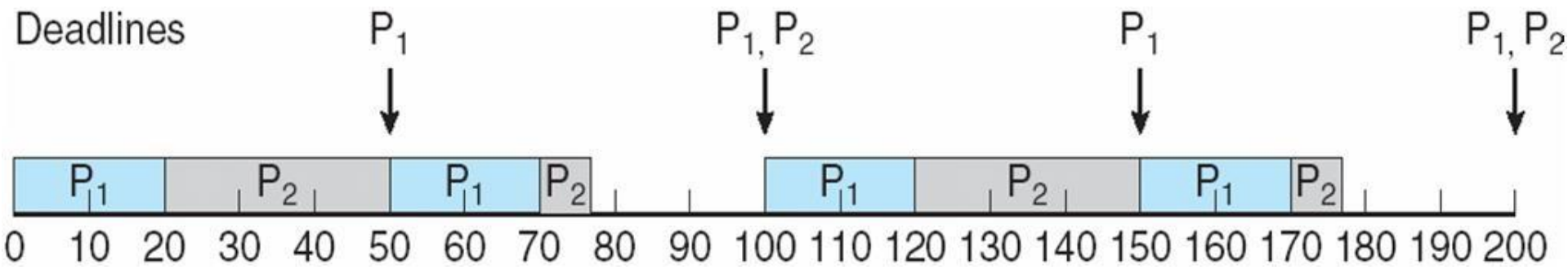
Hard real-time scheduling require additional scheduling features to be able to guarantee that deadline requirements are met.



Real-time processes are often characterized by:

- have a fixed period time **p**
- a fixed processing time on the CPU **t**
- a deadline **d**, by which it must be serviced by the CPU
- the following inequality applies: $0 \leq t \leq d \leq p$

Rate-Monotonic Real-Time Scheduling



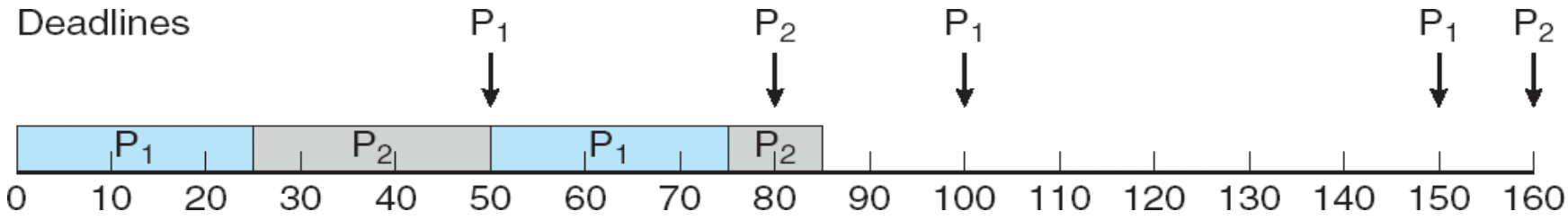
Shorter period time = higher priority
Longer period time = lower priority
P1 (50) therefore has a higher priority than P2 (100)

$$\text{CPU Ulization} = \left(\frac{t_1}{p_1}\right) + \left(\frac{t_2}{p_2}\right) = \left(\frac{20}{50}\right) + \left(\frac{35}{100}\right) = 75\%$$

Rate-Monotonic is considered optimal in the sense that*:*
If processes cannot be scheduled with this algorithm, then they cannot be scheduled by any static priority-based algorithms.

Rate Monotonic Real Time Scheduling

deadline not met



$$\text{CPU Ulization} = \left(\frac{t_1}{p_1}\right) + \left(\frac{t_2}{p_2}\right) = \left(\frac{25}{50}\right) + \left(\frac{35}{80}\right) \approx 94\%$$

This seems to be possible but actually not (P2 does not meet its deadline!)

This is because the worst-case CPU utilization follows this formula for N processes:

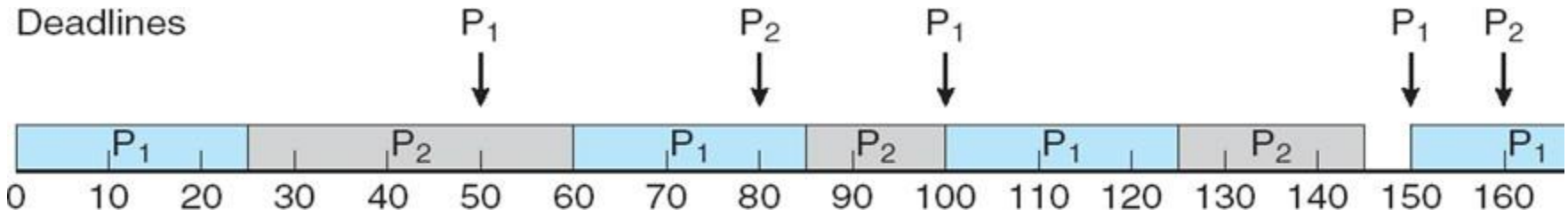
$$\text{CPU Ulization}_{worst\ case} = N \cdot \left(2^{\frac{1}{N}} - 1\right)$$

In the example above:

$$\text{CPU Ulization}_{worst\ case} = 2 \cdot \left(2^{\frac{1}{2}} - 1\right) \approx 83\%$$

Earliest-Deadline-First (EDF) Real-Time Scheduling

Here, the priority is assigned dynamically so that the earliest **next** deadline gives the highest priority.



same scenario as before (where it failed):

period	CPU burst
$P_1 = 50$	$t_1 = 25$
$P_2 = 80$	$t_2 = 35$

P1 does not interrupt P2 at 50 even if a new period starts. This is because P2 at this point has a higher priority than P1, as it has an earlier deadline at 80.

As we can see, P1 interrupts the process P2 at 100, as a new period starts here. P1 has the highest priority, as its next deadline is at 150 while the next deadline of P2 is at 160.

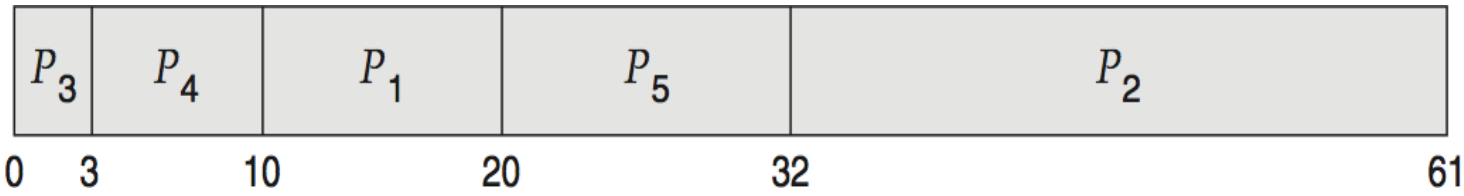
This algorithm does not require periodic processes with constant CPU burst times. It is considered theoretically optimal to achieve 100% CPU utilization, but impossible in practice due to cost of context switch and other overheads.

Evaluation of CPU Scheduling Algorithms

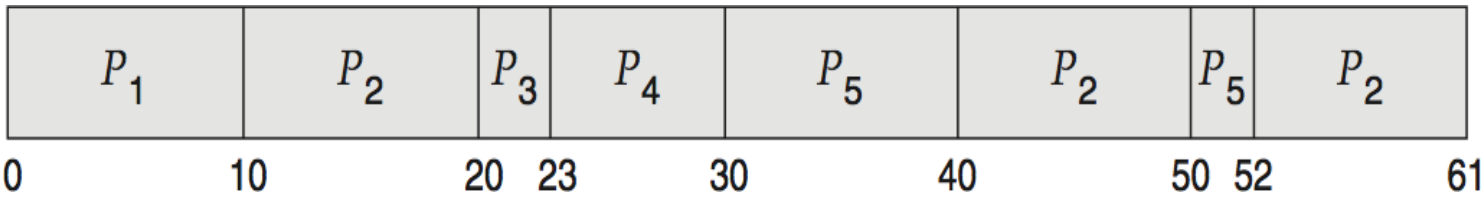
Deterministic modeling

FCFS = 28 ms

Non-preemptive SJF = 13 ms



RR = 23 ms



Five processes arrive at time 0 in the order given with the lengths of the CPU burst of 10, 29, 3, 7 and 12, respectively.

Evaluation of CPU Scheduling Algorithms

Simulations