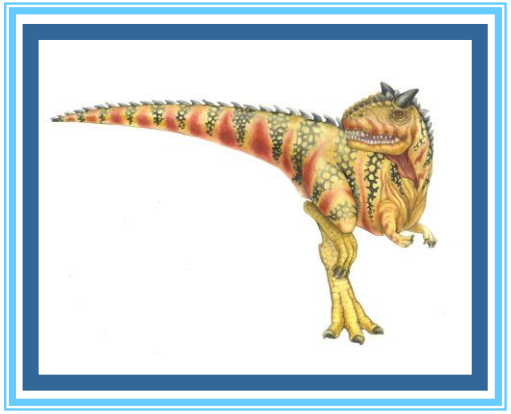
Xi'an Jiaotong-Liverpool University
西交利物浦大學

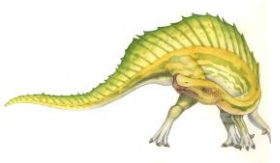
CPT104 - Operating Systems Concepts

CPU Scheduling II



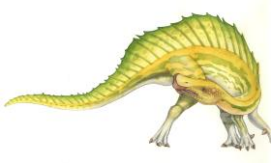
CPU Scheduling

- Thread Scheduling
- Multiple-Processor Scheduling
- Real-Time CPU Scheduling
- Algorithm Evaluation



CPU Scheduling II

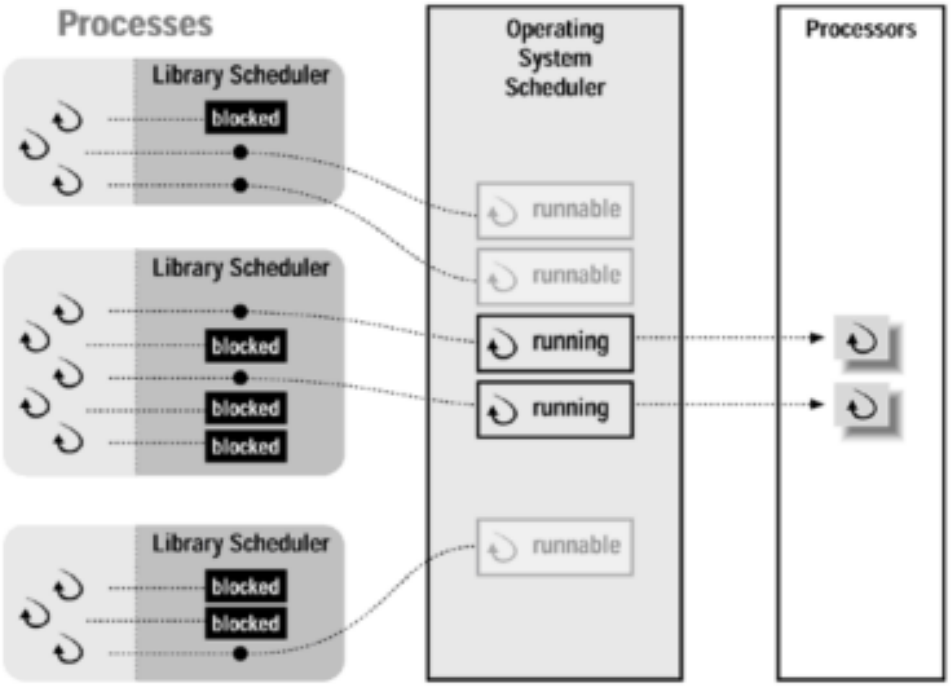
- **Thread Scheduling**
- Multiple-Processor Scheduling
- Real-Time CPU Scheduling
- Algorithm Evaluation



Lightweight Processes (LWP)

Lightweight processes is a schedulable entity which exist on a layer between *kernel-threads* and *user-threads*. It is managed by the kernel.

- One actual process may use multiple LWPs.
- Each LWP is bound to a kernel thread.
- User-level threads can be flexibly mapped to LWP.



Contention scope

- Distinction between user-level and kernel-level threads
- When threads supported, threads scheduled, not processes

The **contention scope** refers to the scope in which threads compete for the use of physical CPU(s).

There are two possible contention scopes:

- **Process Contention Scope PCS**, a.k.a *local contention scope*.
- **System Contention Scope SCS**, a.k.a *global contention scope.*

The ULT-KLT mapping is decided by the developer.

The thread models:

- Many to One model
- One to One model
- Many to Many model.

Thread Scheduling

The basic levels to schedule threads:

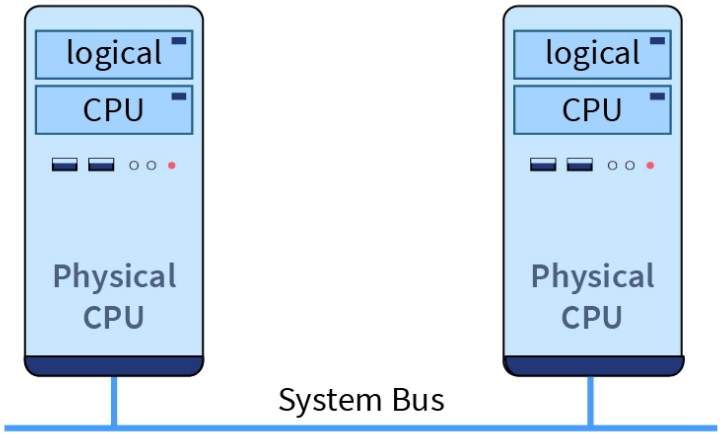
- **Process Contention Scope** (unbound threads) - competition for the CPU takes place among threads belonging to the same process. The thread library schedules the PCS thread to access the resources via available *Lightweight Processes* **LWP**s (priority as specified by the application developer during thread creation).
  - It used by **many-to-many** and **many-to-one** models.
- **System Contention Scope** (bound threads) - competition for the CPU takes place among all threads in the system. This scheme is used by the kernel to decide which kernel-level thread to schedule onto a CPU.
  - It used only by **one-to-one** model.

CPU Scheduling

- Thread Scheduling
- **Multiple-Processor Scheduling**
- Real-Time CPU Scheduling
- Algorithm Evaluation

Approaches to Multiple-Processor Scheduling

- A *Multi-processor* is a system that has more than one processor but shares the same *memory*, *bus*, and *input/output devices*.
- CPU scheduling more complex when multiple CPUs are available
- We are focused on **Homogeneous processors** (processors are identical)



- **Asymmetric multiprocessing** – only one processor accesses the system data structures (type master-slave(s)).
- **Symmetric multiprocessing (SMP)** – each processor is *self-scheduling*.

Asymmetric multiprocessing

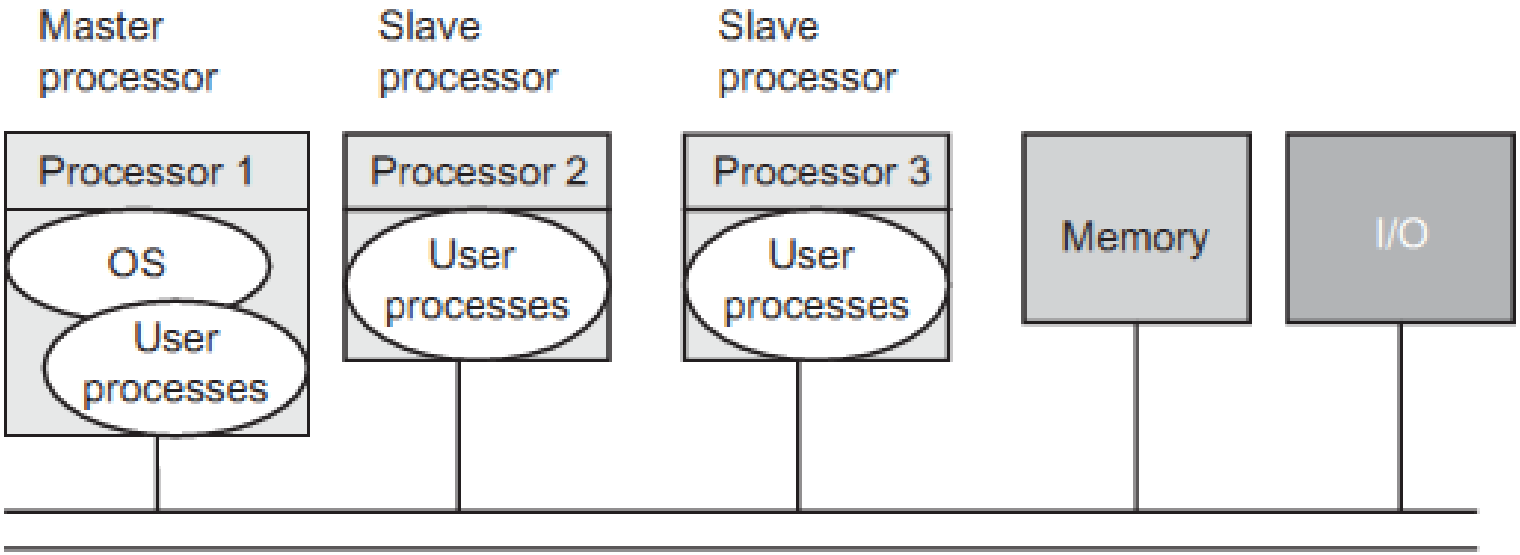
Master – Slave Configuration

One processor as **master** and other processors in the system as **slaves**.

The master processor runs the OS and processes while slave processors run the processes only.

The **process scheduling is performed by the master processor**.

The **parallel processing is possible** as a task can be broken down into sub-tasks and assigned to various processors.

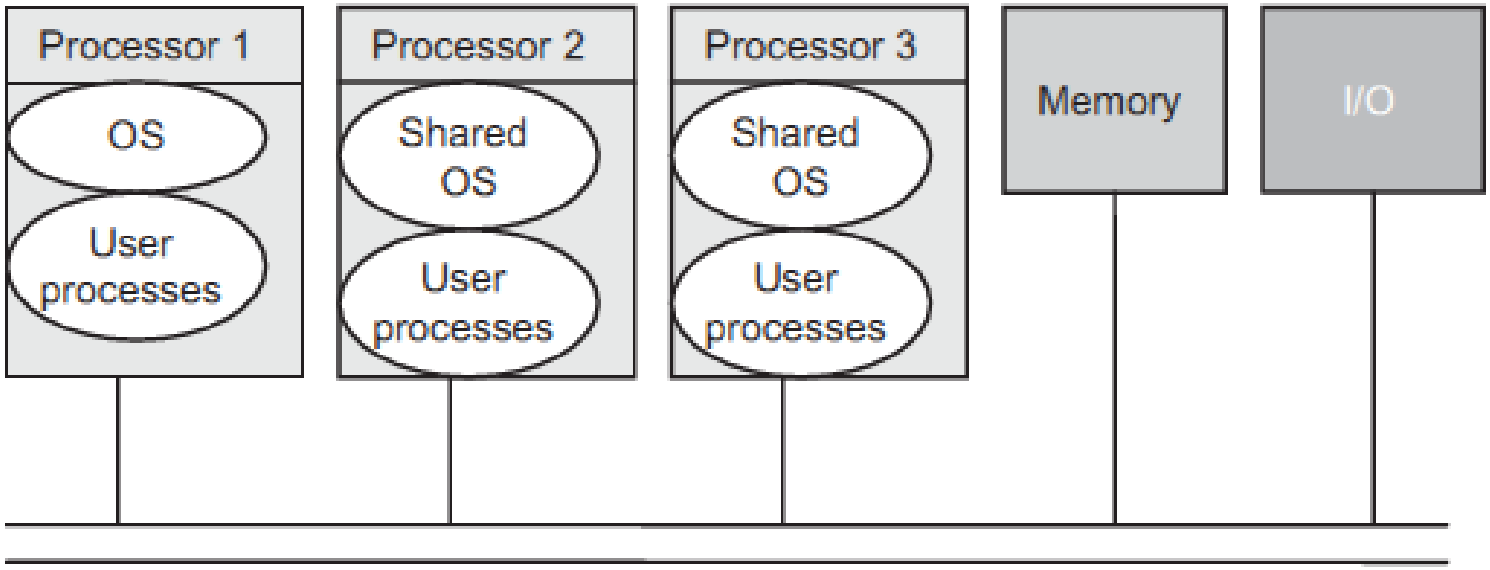


Symmetric Configuration SMP

Any processor can access any device and can handle any interrupts generated on it.

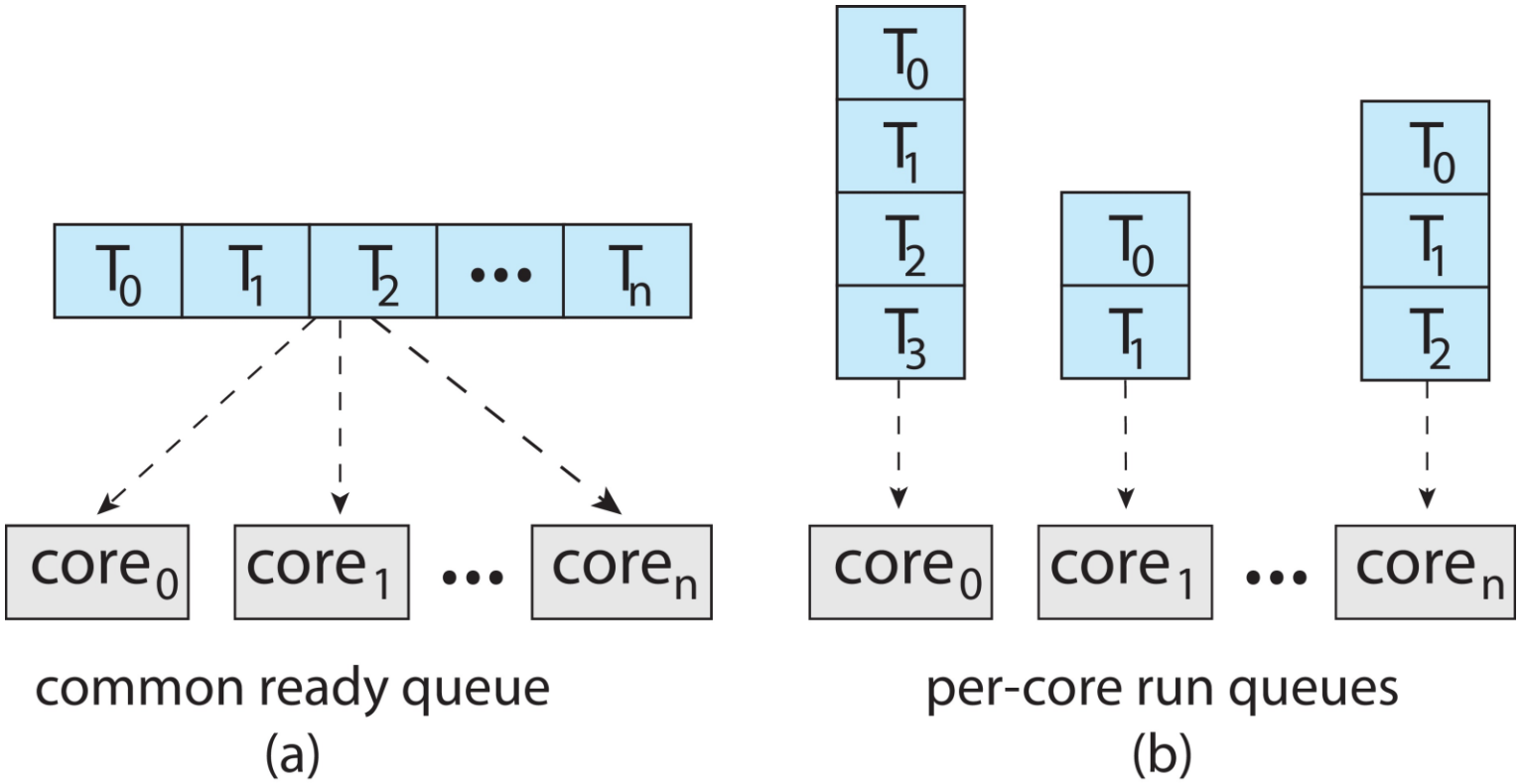
Mutual exclusion must be enforced such that only one processor is allowed to execute the OS at one time.

To prevent the concurrency of the processes many parts of the OS are independent of each other such as scheduler, file system call, etc.



Approaches to Symmetric Configuration

- all processes in common *ready queue*, or
- each of the processors has its own *private queue* of ready processes



Processor Affinity

Processor affinity is the ability to direct a specific task/process to use a specified core.

- *The idea behind*: When a process runs on a processor, the data accessed by the process most recently is populated in the *cache* memory of this processor. If the process is directed to always use the same core, it is possible that the process will run more efficiently because of the *cache* re-use.

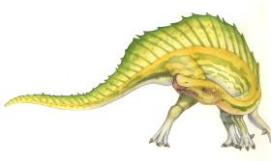
Note: If a process migrates from one CPU to another, the old instruction and address caches become invalid, and it will take time for caches on the new CPU to become 'populated'.

- **Soft affinity** – O.S. trying to keep a process running on the same processor but not guaranteeing it will do so.
- **Hard affinity** – O.S. allows a process to specify a subset of processors on which it may run.
  - **Linux** implement *soft affinity*, but they also provide system calls to support *hard affinity.*

Load Balancing

Load Balancing → a method of distributing work between the processors fairly in order to get optimal response time, resource utilization, and throughput.

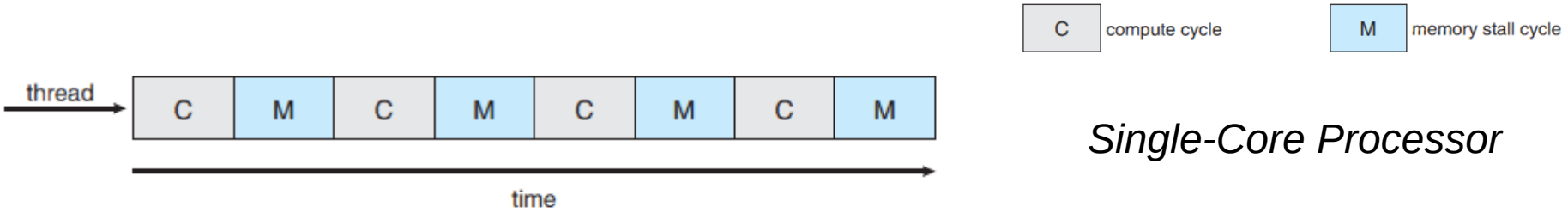
- **Push migration** = a task routinely checks (e.g., every 200 ms) the load on each processor. If the workload is unevenly distributed, moves (or *push*) processes to idle or less busy processor(s).
- **Pull migration** = an idle processor will extract the load from an overloaded processor itself (*pulls* a *waiting task from a busy processor*).



Multicore Processors

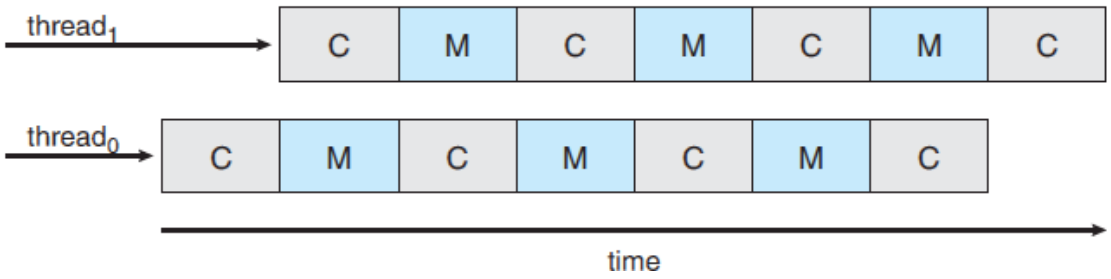
- A core executes one thread at a time.
- *Hyper Threading allows multiple threads to run on each core of CPU*

- Single-core processor spends time waiting for the data to become available (slowing or stopping of a process) = **Memory stall.**



Single-Core Processor

Solution!!! **Multicore processor**: a single computing component comprised of two or more CPUs called *cores* to run multiple threads concurrently. Each core has a register set thus appears to the operating system as a separate physical processor.



Multi-Core Processor

Multithreading

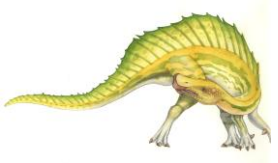
How do we execute multiple threads on same core?

Techniques for multithreading:

- **Coarse-grained multithreading** - switching between threads only when *one thread blocks* (long latency event such as a memory stall occurs).
- **Fine-grained multithreading** - instructions *"scheduling" among threads* obeys a Round Robin policy.

CPU Scheduling

- Thread Scheduling
- Multiple-Processor Scheduling
- **Real-Time CPU Scheduling**
- Algorithm Evaluation



Real-Time Operating Systems

A **real-time operating system** R.T.O.S. are **deadline driven.**

Examples: the patient monitoring in a hospital intensive-care unit, the autopilot in an aircraft, radar systems, robot control in an automated factory, etc.

- **Hard RTOS** – is one that **must meet its deadline**; otherwise, it will cause unacceptable damage or a fatal error to the system.
- **Soft RTOS** – an associated **deadline** that **is desirable but not necessary**; it still makes sense to schedule and complete the task even if it has passed its deadline.

Characteristics of a RTOS

The timing constraints are in forms of **period** and **deadline**.

The **period** is the amount of time between iterations of a regularly repeated task.

The **deadline** is a constraint of the maximum time limit within which the operation must be complete.

- **Aperiodic tasks** (*random time*) has **irregular arrival times** and either soft or hard deadlines.
- **Periodic tasks** (***repeated tasks***), the requirement may be stated as “once per period T” or “exactly T units apart.”

Issues in Real-time Scheduling

The major **challenges** for an RTOS is **to schedule the real-time tasks**.

Two types of **latencies may delay the processing** (performance):

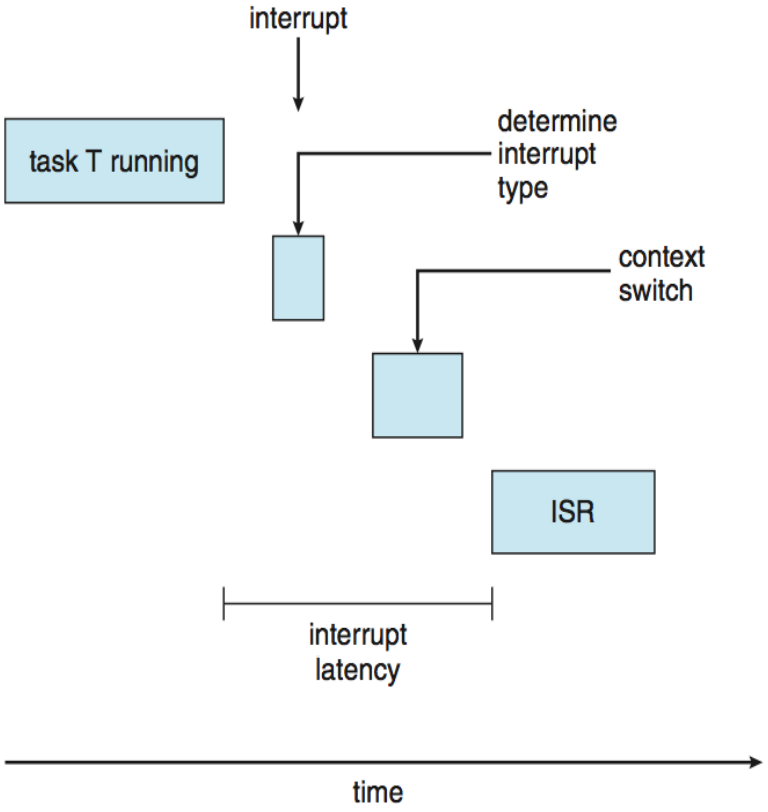
1. Interrupt latency *(*aka *interrupt response time)* - is the time elapsed between the **last instruction** executed on the current interrupted task and **start of the interrupt handler**.

An interrupt is a signal emitted by a device attached to a computer or from a program within the computer. It requires the operating system to stop and figure out what to do next.

2. Dispatch latency – time it takes for the dispatcher to **stop** one process and **start** another running. *To keep dispatch latency low is to provide preemptive kernels.*

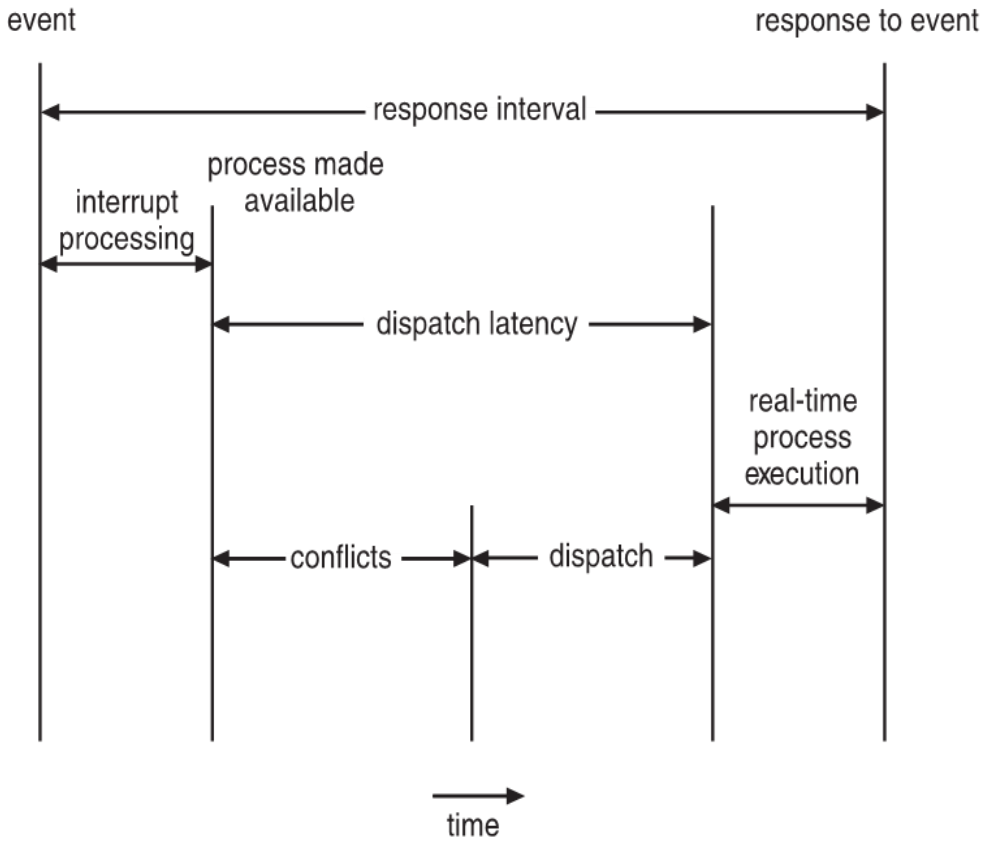
Real-Time Operating Systems

Interrupt latency



ISR = interrupt service routine

Dispatch latency



Real-Time CPU Scheduling

The RTOS schedules all tasks according to the deadline information and ensures that all deadlines are met.

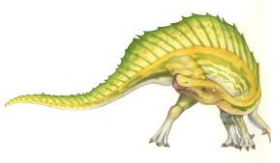
Static scheduling. A **schedule is prepared before execution** of tasks/processes.

Priority-based scheduling. The **priority assigned to tasks** depends on how quickly a task has to respond to the event.

Dynamic scheduling. There is complete knowledge of tasks set, but new arrivals are not known. Therefore, **the schedule changes over the time**.

Real-Time CPU Scheduling

- Rate-Monotonic Scheduling
  - ➢ Missed Deadlines with Rate Monotonic Scheduling
- Earliest-Deadline-First Scheduling
- Proportional Share Scheduling



Characteristics of processes

Processes are considered **periodic** (repeated tasks).

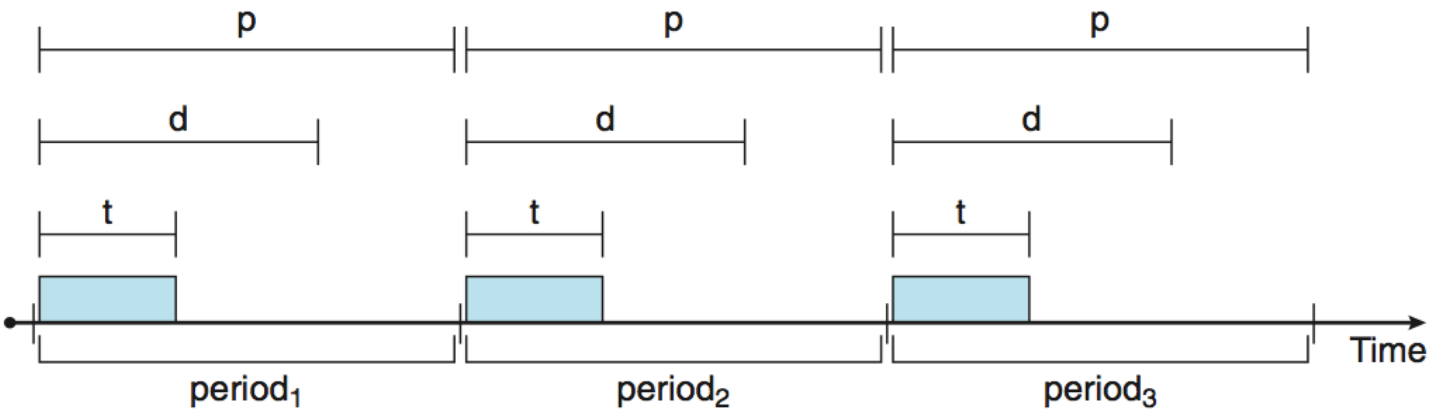
A periodic process has:

- **processing time t**,
- **deadline d** by which it **must be serviced by the CPU**, and
- **period p**.

$$0 \leq t \leq d \leq p$$

Rate of a periodic task is $1/p$

A process may have to announce its deadline requirements to the scheduler.



Rate Monotonic Scheduling (RMS)

It is a **static priority-based preemptive scheduling algorithm.**

The task with the shortest period will always preempt the executing task.

The higher the frequency (1/period) of a task, the higher is its priority

The CPU utilization of a process Pi

$\mathbf{t_i}$ = the execution time

$\mathbf{p_i}$ = the period of process

$$CPU\ utilization = \frac{t_i}{p_i}$$

To meet all deadlines in the system, the following must be satisfied:

$$\sum_i \frac{t_i}{p_i} \leq 1$$

Rate Monotonic Scheduling (RMS)

The shortest period = the highest priority;

Each process requires to complete its burst time by the start of its next period.

P1 is high priority than P2.

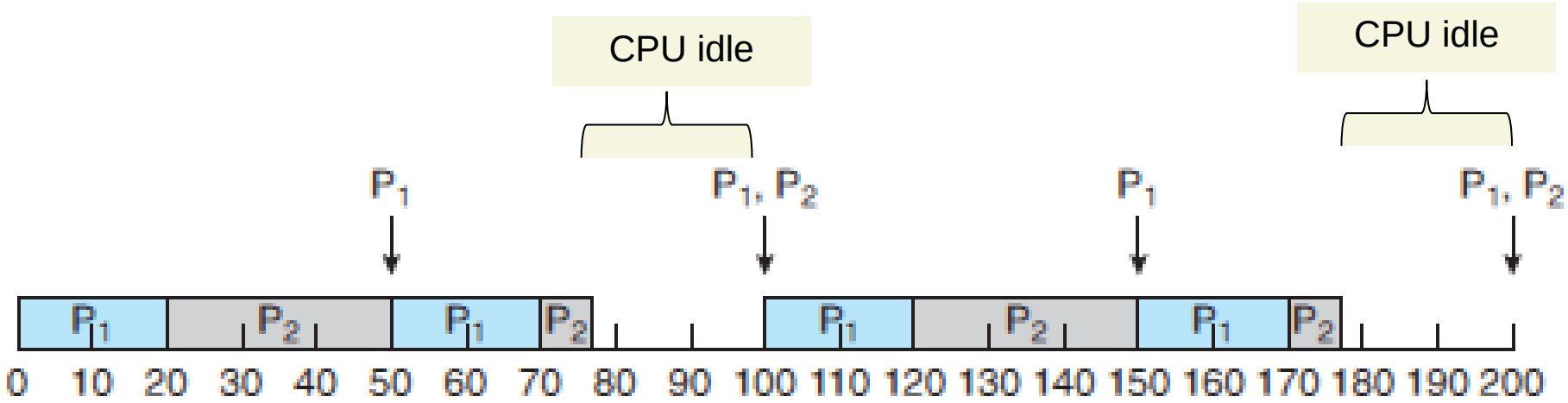
P1 $p_1 = 50$, $t_1 = 20$ — the CPU utilization of P1 = 20/50 = 0.4

P2 $p_2 = 100$, $t_2 = 35$. — the CPU utilization of P2 = 35/100 = 0.35

$\mathbf{t_i}$ = the execution/burst time

$\mathbf{p_i}$ = the period of process

The total CPU utilization – 75%



Rate Monotonic Scheduling (RMS)

- **Assume P2 is high priority than P1.**

Each process requires to complete its burst time by the start of its next period.

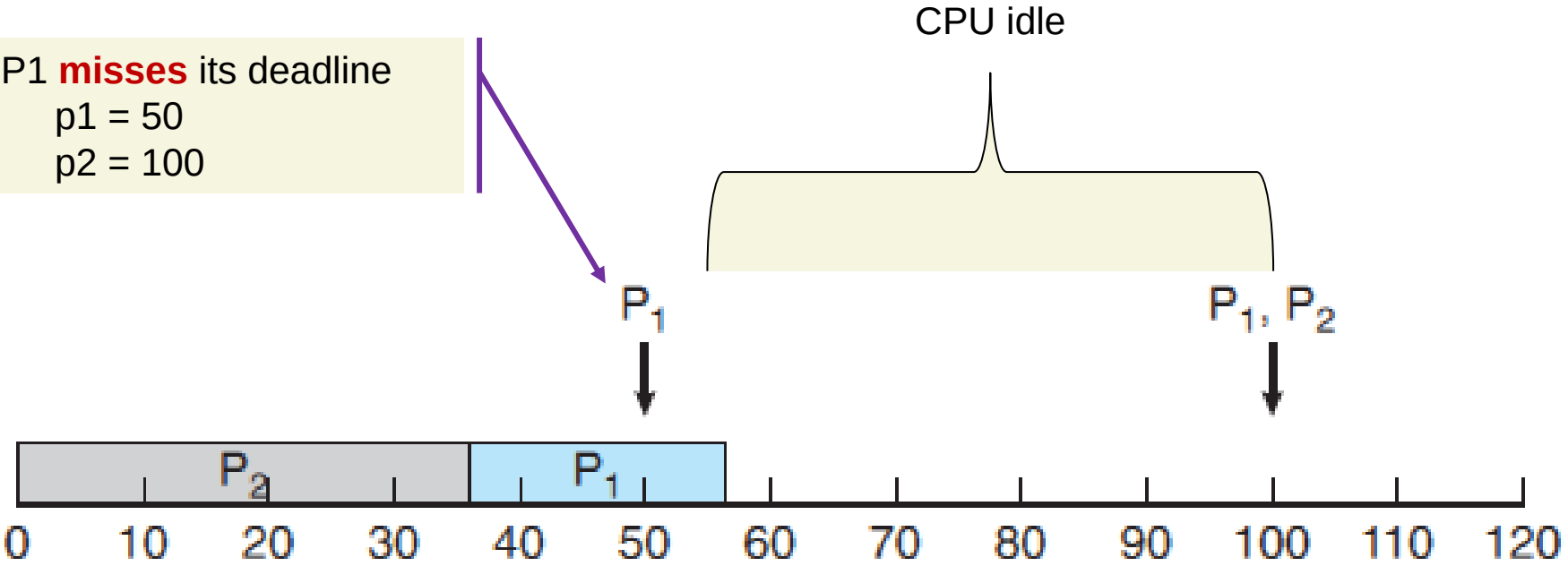
P1 $p_1 = 50$, $t_1 = 20$

P2 $p_2 = 100$, $t_2 = 35$.

$\mathbf{t_i}$ = the execution/burst time

$\mathbf{p_i}$ = the period of process

P1 **misses** its deadline
p1 = 50
p2 = 100



Missing Deadlines with RMS

A set of processes that cannot be scheduled using the RMS.

P1: $p_1 = 50$, $t_1 = 25$.

P2: $p_2 = 80$, $t_2 = 35$.

process **P1 is high priority** (p1 < p2).

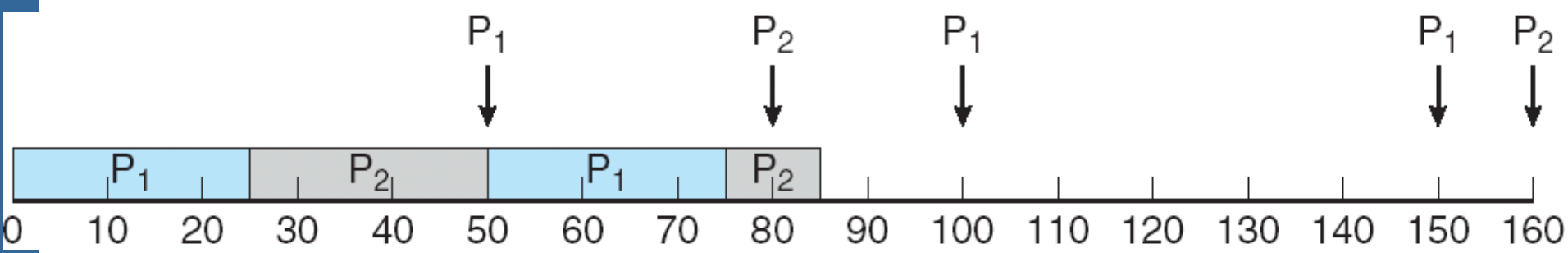
t_i = the execution/burst time

p_i = the period of process

The total CPU utilization = 0.94

P2 still **has** 10 ms **to run**

P2 **misses** its deadline.
p1 = 100,
p2 = 80.



Earliest-Deadline-First Scheduling

The scheduling criterion is **based on the deadline of the processes**.

It can be used for both ***static*** and ***dynamic real-time scheduling***.

The tasks or processes do not need to be periodic.

Any ***executing task can be preempted*** if any other periodic instance with an earlier deadline is ready (*dynamic real-time scheduling**)***

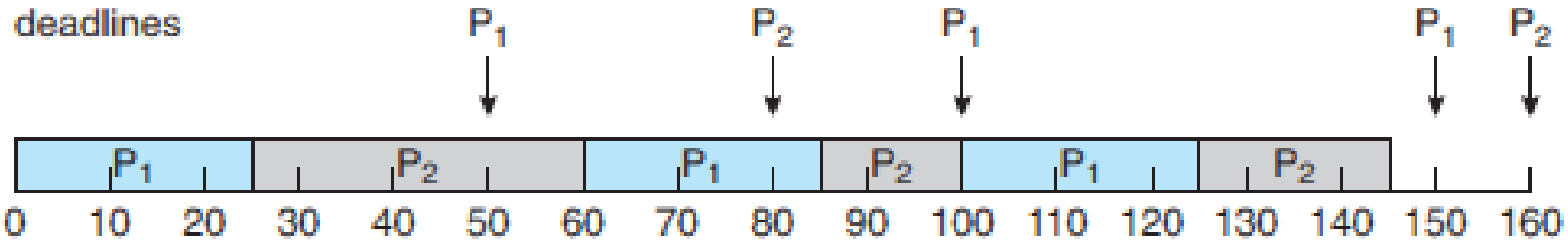
the earlier is the deadline = the higher is the priority;

P1: $(p_1)\ d_1 = 50,\ t_1 = 25.$

P2: $(p_2)\ d_2 = 80,\ t_2 = 35.$

$\mathbf{t_i}$ = the execution/burst time

$\mathbf{(p_i)\ d_i}$ = the deadline of process



Proportional Share Scheduling

Scheduling that pre-allocates certain amount of CPU time to each of the processes.

Fair-share scheduler

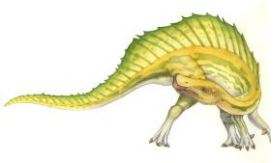
- Guarantee that each process obtain a certain percentage of CPU time
- Not optimized for turnaround or response time

- ***T*** shares are allocated among all processes in the system
- An application receives ***N*** shares where ***N < T***
- This ensures each application will receive ***N / T*** of the total processor time

Example:
T = 100 shares is to be divided among three processes, A, B, and C.
A is assigned 50 shares, B is assigned 15 shares, and C is assigned 20 shares.

CPU Scheduling

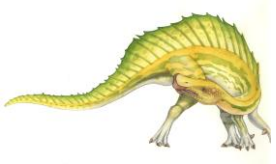
- Thread Scheduling
- Multiple-Processor Scheduling
- Real-Time CPU Scheduling
- **Algorithm Evaluation**



Scheduling Algorithm Evaluation

How do we select a CPU-scheduling algorithm for a particular system?

- **Deterministic evaluation**
- **Queueing Models**
- **Simulations**

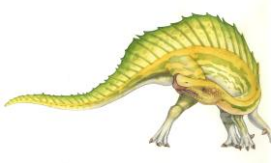


1. Deterministic Modeling

- *What algorithm can provide the minimum average waiting time?*
- Takes a particular predetermined workload and defines the performance of each algorithm for that workload.

- Consider 5 processes arriving at time 0:

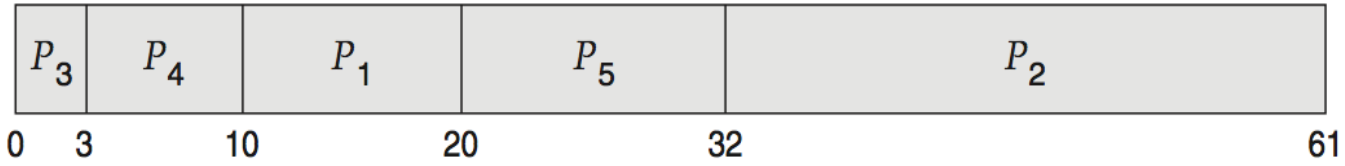
Process	Burst Time
P_1	10
P_2	29
P_3	3
P_4	7
P_5	12



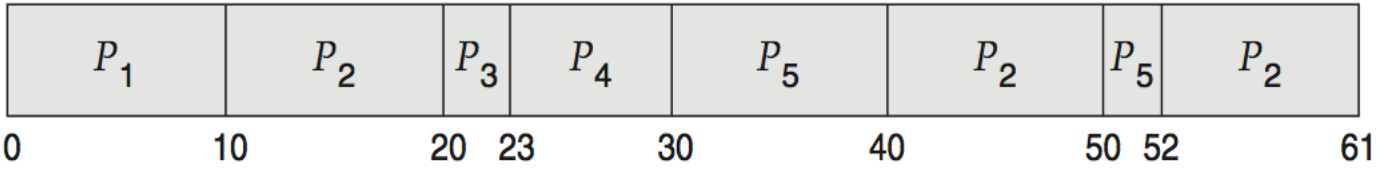
1. Deterministic Modeling

- **FCFS** is 28ms:

- Non-preemptive **SFJ** is 13ms:



- **RR** is 23ms:



2. Queueing Models

- **Little's formula** – "*the average number of processes in a system (over some interval) is equal to their average arrival rate, multiplied by their average time in the system*."
- processes leaving queue must equal processes arriving, thus:

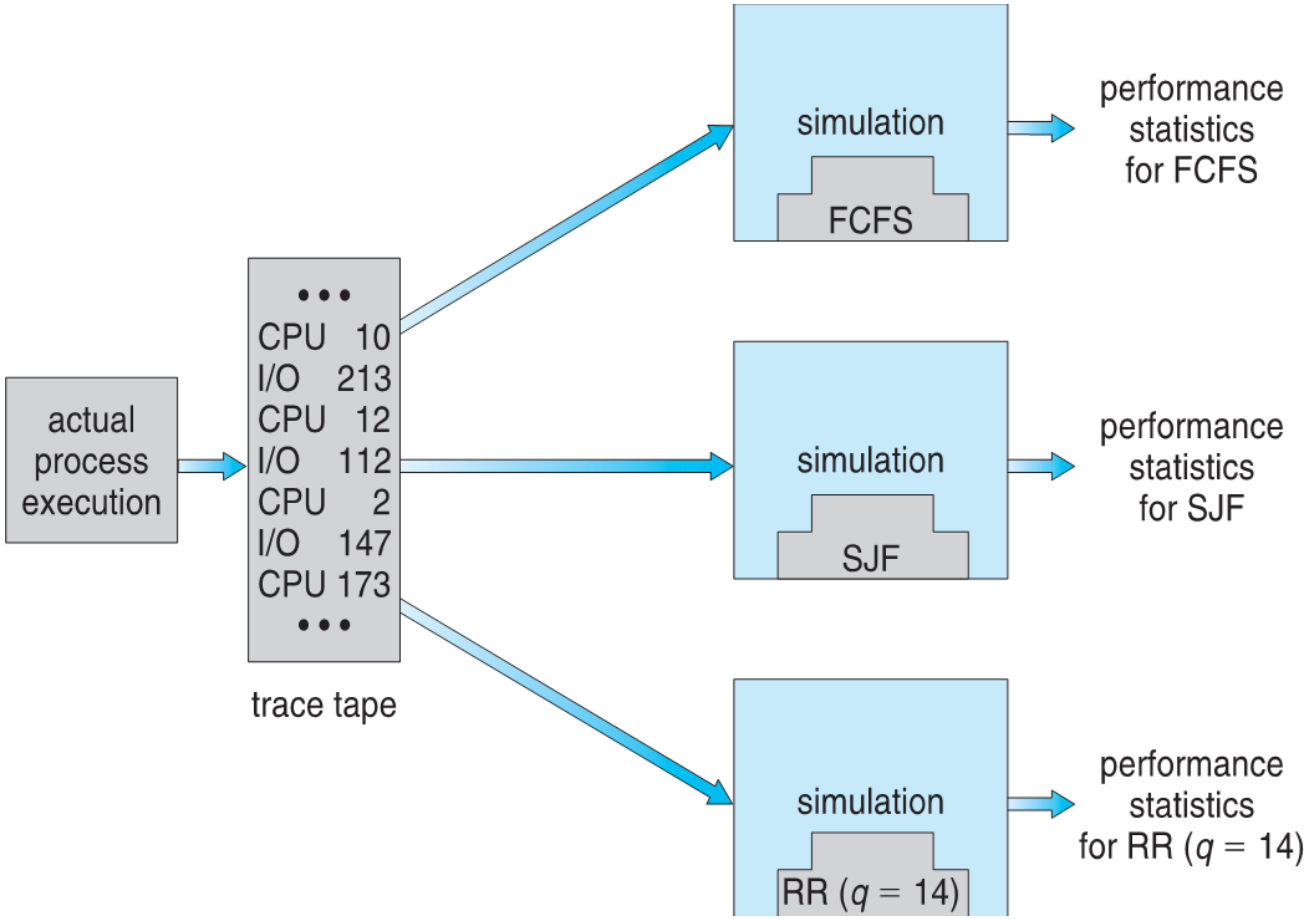
$$n = \lambda \times W$$

n = average queue length
W = average waiting time in queue
λ = average arrival rate into queue

Example, if on average *7 processes arrive per second*, and normally 14 processes in queue, then average wait time per process = 2 seconds (n=14, λ =7)

3. Simulation

Uses data collected (**trace tapes)** from real processes on real machines and is fed into the simulation.



End of Lecture

Summary

- Thread Scheduling
- Multiple-Processor Scheduling
- Real-Time CPU Scheduling
- Algorithm Evaluation

Reading

- The module textbook 9th edition - **Chapter 6**