

Chapter 5: CPU Scheduling





Chapter 5: CPU Scheduling

- ❑ Basic Concepts
- ❑ Scheduling Criteria
- ❑ Scheduling Algorithms
- ❑ Multi-Processor Scheduling
- ❑ Real-Time CPU Scheduling
- ❑ Algorithm Evaluation





Objectives

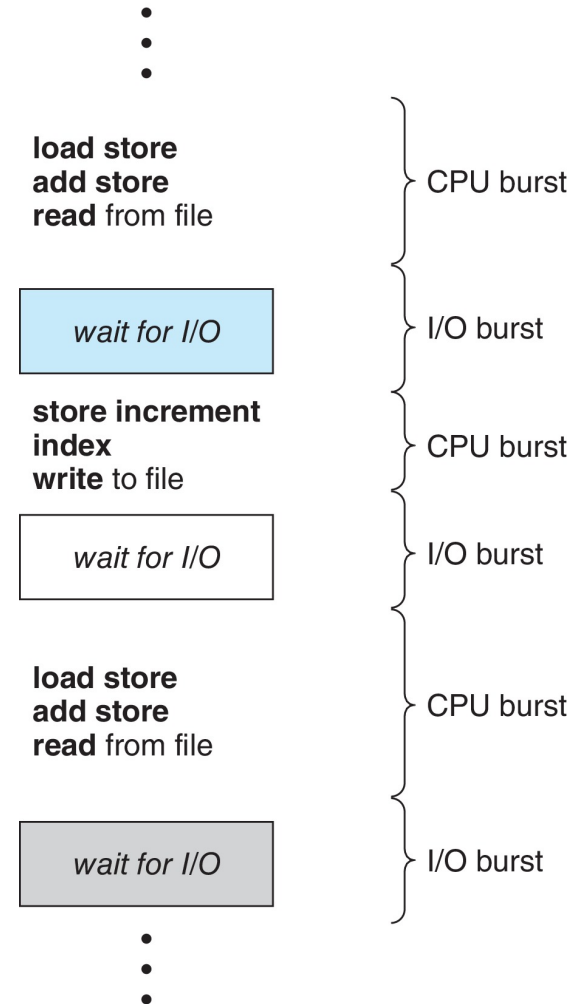
- Describe various CPU scheduling algorithms
- Assess CPU scheduling algorithms based on scheduling criteria
- Explain the issues related to multiprocessor and multicore scheduling
- Describe various real-time scheduling algorithms
- Apply modeling and simulations to evaluate CPU scheduling algorithms





Basic Concepts

- ❑ Maximum CPU utilization obtained with multiprogramming
- ❑ CPU–I/O Burst Cycle – Process execution consists of a **cycle** of CPU execution and I/O wait
- ❑ **CPU burst** followed by **I/O burst**
- ❑ CPU burst distribution is of main concern

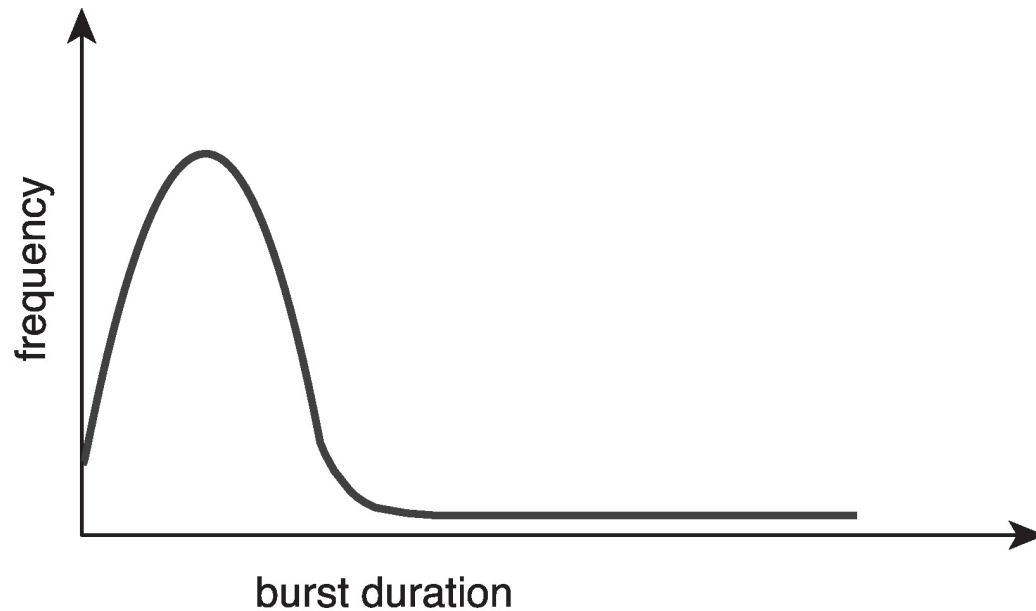




Histogram of CPU-burst Times

Large number of short bursts

Small number of longer bursts





CPU Scheduler

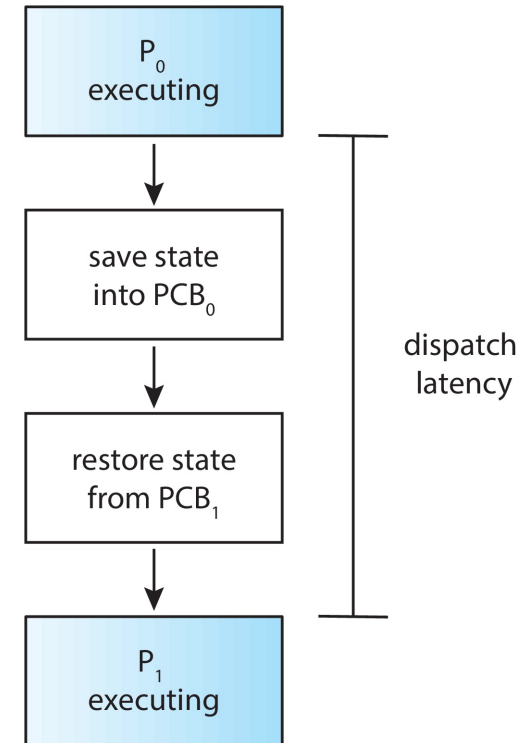
- The **CPU scheduler** selects from among the processes in ready queue, and allocates a CPU core to one of them
 - Queue may be ordered in various ways
- We have two types of scheduling: **non-preemptive** and **preemptive**





Dispatcher

- ❑ Dispatcher module gives control of the CPU to the process selected by the short-term scheduler; this involves:
 - ❑ switching context
 - ❑ switching to user mode
- ❑ **Dispatch latency** – time it takes for the dispatcher to stop one process and start another running





Scheduling Criteria

- ❑ **CPU utilization** – keep the CPU as busy as possible
- ❑ **Throughput** – # of processes that complete their execution per time unit
- ❑ **Turnaround time** – amount of time to execute a particular process
- ❑ **Waiting time** – amount of time a process has been waiting in the ready queue
- ❑ **Response time** – amount of time it takes from when a request was submitted until the first response is produced, not output (for time-sharing environment)





Scheduling Algorithm Optimization Criteria

- ❑ Max CPU utilization
- ❑ Max throughput
- ❑ Min turnaround time
- ❑ Min waiting time
- ❑ Min response time





First- Come, First-Served (FCFS) Scheduling

| <u>Process</u> | <u>Burst Time</u> |
|----------------|-------------------|
| P_1 | 24 |
| P_2 | 3 |
| P_3 | 3 |

- Suppose that the processes arrive in the order: P_1, P_2, P_3
The Gantt Chart for the schedule is:



- Waiting time for $P_1 = 0$; $P_2 = 24$; $P_3 = 27$
- Average waiting time: $(0 + 24 + 27)/3 = 17$





FCFS Scheduling (Cont.)

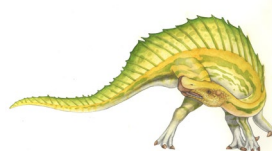
Suppose that the processes arrive in the order:

$$P_2, P_3, P_1$$

- The Gantt chart for the schedule is:



- Waiting time for $P_1 = 6$; $P_2 = 0$; $P_3 = 3$
- Average waiting time: $(6 + 0 + 3)/3 = 3$
- Much better than previous case
- **Convoy effect** - short process behind long process
 - Consider one CPU-bound and many I/O-bound processes





Shortest-Job-First (SJF) Scheduling

- Associate with each process the length of its next CPU burst
 - Use these lengths to schedule the process with the shortest time
- SJF is optimal – gives minimum average waiting time for a given set of processes
 - The difficulty is knowing the length of the next CPU request
 - Could ask the user

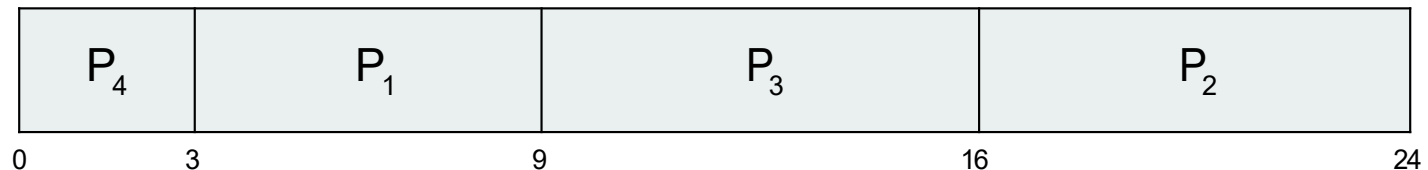




Example of SJF

| <u>Process</u> | <u>Burst Time</u> |
|----------------|-------------------|
| P_1 | 6 |
| P_2 | 8 |
| P_3 | 7 |
| P_4 | 3 |

□ SJF scheduling chart



□ Average waiting time = $(3 + 16 + 9 + 0) / 4 = 7$





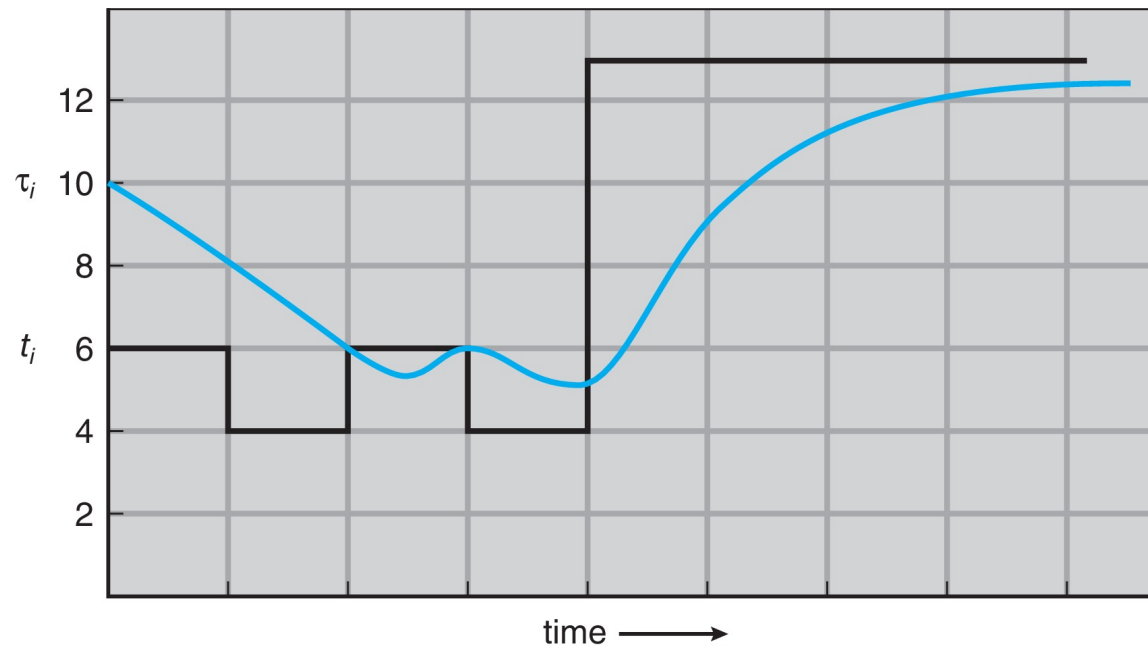
Determining Length of Next CPU Burst

- Can only estimate the length – should be similar to the previous one
 - Then pick process with shortest predicted next CPU burst
- Can be done by using the length of previous CPU bursts, using exponential averaging
 1. t_n = actual length of n^{th} CPU burst
 2. τ_{n+1} = predicted value for the next CPU burst
 3. $\alpha, 0 \leq \alpha \leq 1$
 4. Define: $\tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n$.
- Commonly, α set to $\frac{1}{2}$
- Preemptive version called **shortest-remaining-time-first**





Prediction of the Length of the Next CPU Burst



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





Examples of Exponential Averaging

- $\alpha = 0$
 - $\tau_{n+1} = \tau_n$
 - Recent history does not count
- $\alpha = 1$
 - $\tau_{n+1} = \alpha t_n$
 - Only the actual last CPU burst counts
- If we expand the formula, we get:

$$\begin{aligned}\tau_{n+1} = & \alpha t_n + (1 - \alpha)\alpha t_{n-1} + \dots \\ & + (1 - \alpha)^j \alpha t_{n-j} + \dots \\ & + (1 - \alpha)^{n+1} \tau_0\end{aligned}$$

- Since both α and $(1 - \alpha)$ are less than or equal to 1, each successive term has less weight than its predecessor



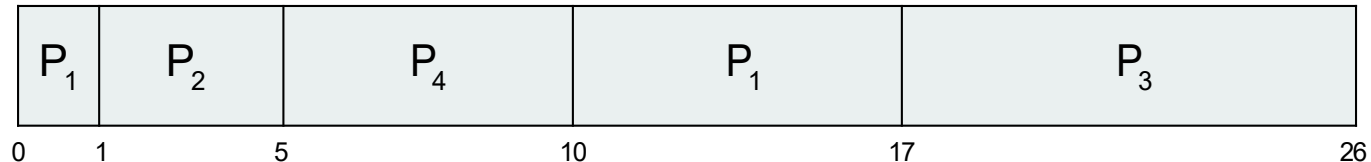


Example of Shortest-remaining-time-first

- Now we add the concepts of varying arrival times and preemption to the analysis

| <u>Process</u> | <u>Arrival Time</u> | <u>Burst Time</u> |
|----------------|---------------------|-------------------|
| P_1 | 0 | 8 |
| P_2 | 1 | 4 |
| P_3 | 2 | 9 |
| P_4 | 3 | 5 |

- Preemptive* SJF Gantt Chart



- Average waiting time = $[(10-1)+(1-1)+(17-2)+5-3]/4 = 26/4 = 6.5$ msec





Round Robin (RR)

- Each process gets a small unit of CPU time (**time quantum** q), usually 10-100 milliseconds. After this time has elapsed, the process is preempted and added to the end of the ready queue.
- If there are n processes in the ready queue and the time quantum is q , then each process gets $1/n$ of the CPU time in chunks of at most q time units at once. No process waits more than $(n-1)q$ time units.
- Timer interrupts every quantum to schedule next process
- Performance
 - q large \Rightarrow FIFO
 - q small $\Rightarrow q$ must be large with respect to context switch, otherwise overhead is too high

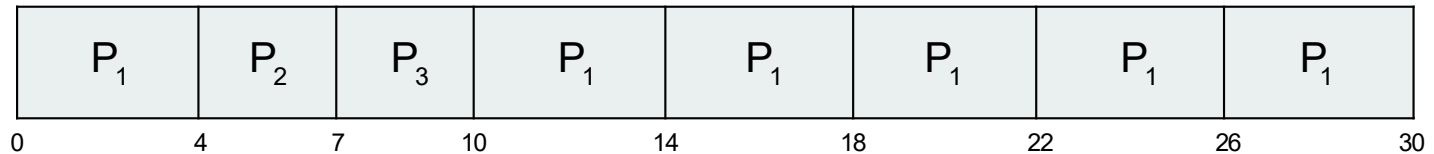




Example of RR with Time Quantum = 4

| <u>Process</u> | <u>Burst Time</u> |
|----------------|-------------------|
| P_1 | 24 |
| P_2 | 3 |
| P_3 | 3 |

□ The Gantt chart is:

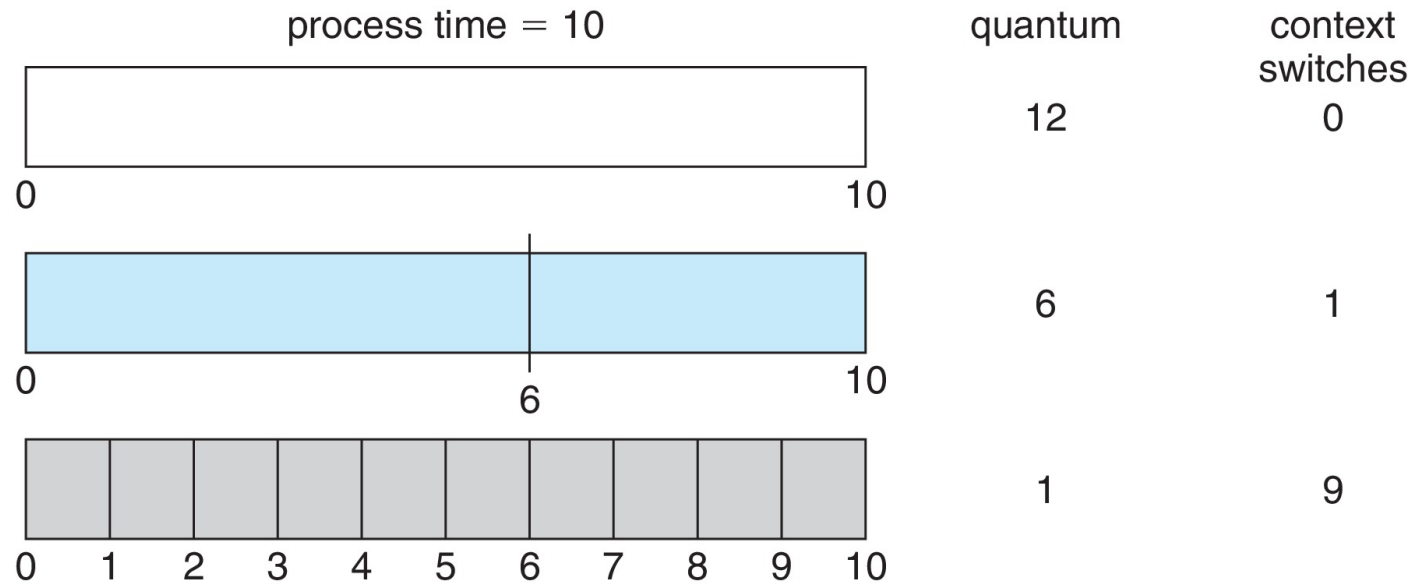


- Typically, higher average waiting time
- Typically, higher average turnaround than SJF
- q should be large compared to context switch time
- q usually 10ms to 100ms, context switch < 10 usec



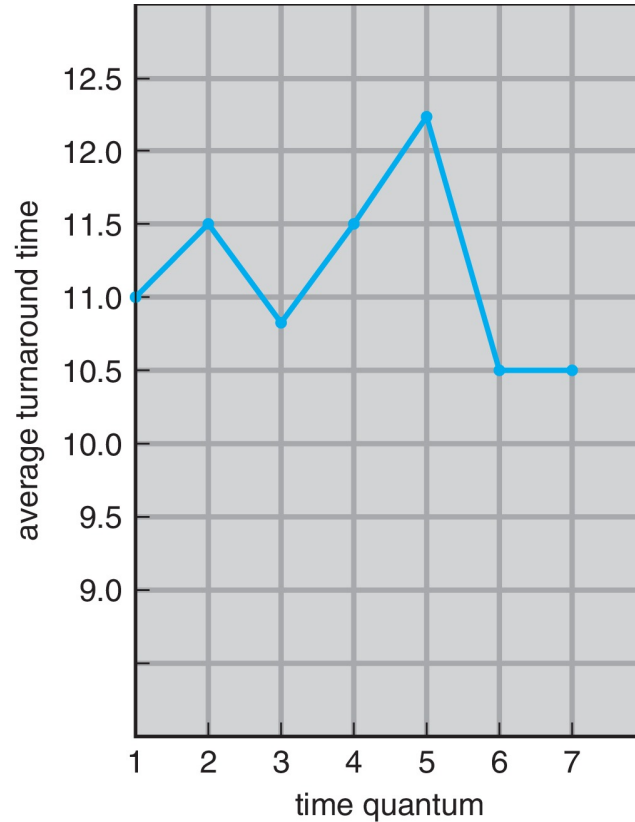


Time Quantum and Context Switch Time





Turnaround Time Varies With The Time Quantum



| process | time |
|---------|------|
| P_1 | 6 |
| P_2 | 3 |
| P_3 | 1 |
| P_4 | 7 |

80% of CPU bursts
should be shorter than q





Priority Scheduling

- A priority number (integer) is associated with each process
- The CPU is allocated to the process with the highest priority (smallest integer \equiv highest priority)
 - Preemptive
 - Nonpreemptive
- SJF is priority scheduling where priority is the inverse of predicted next CPU burst time
- Problem \equiv **Starvation** – low priority processes may never execute
- Solution \equiv **Aging** – as time progresses increase the priority of the process

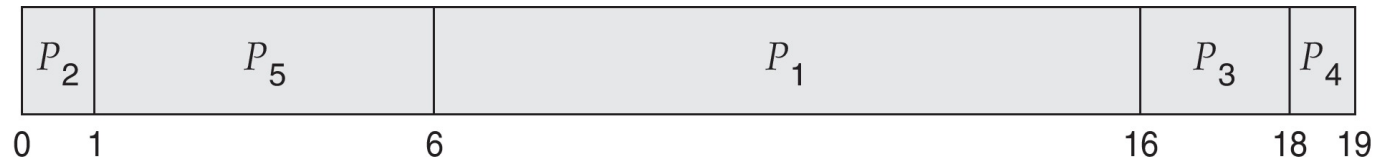




Example of Priority Scheduling

| <u>Process</u> | <u>Burst Time</u> | <u>Priority</u> |
|----------------|-------------------|-----------------|
| P_1 | 10 | 3 |
| P_2 | 1 | 1 |
| P_3 | 2 | 4 |
| P_4 | 1 | 5 |
| P_5 | 5 | 2 |

□ Priority scheduling Gantt Chart



□ Average waiting time = 8.2 msec

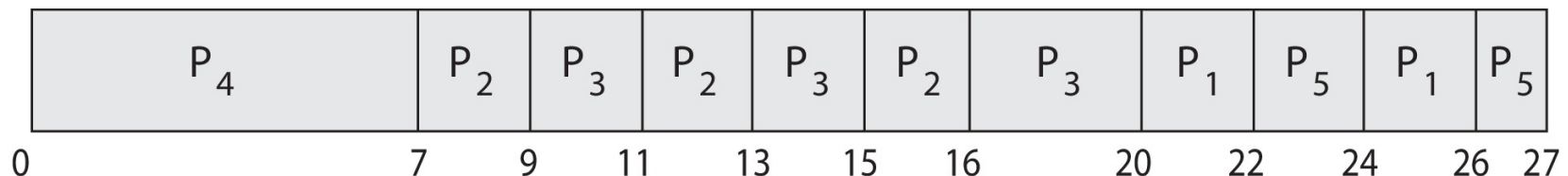




Priority Scheduling w/ Round-Robin

| <u>Process</u> | <u>Burst Time</u> | <u>Priority</u> |
|----------------|-------------------|-----------------|
| P_1 | 4 | 3 |
| P_2 | 5 | 2 |
| P_3 | 8 | 2 |
| P_4 | 7 | 1 |
| P_5 | 3 | 3 |

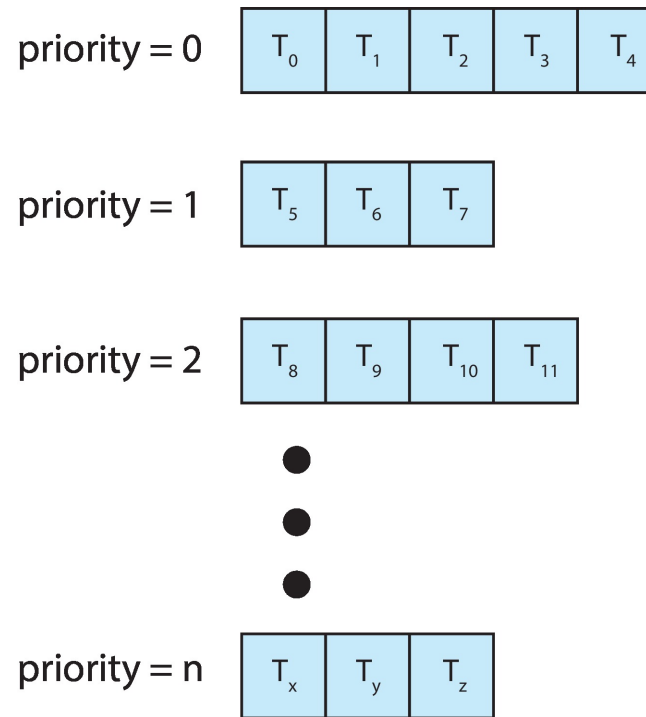
- ❑ Run the process with the highest priority. Processes with the same priority run round-robin
- ❑ Gantt Chart with 2 ms time quantum





Multilevel Queue

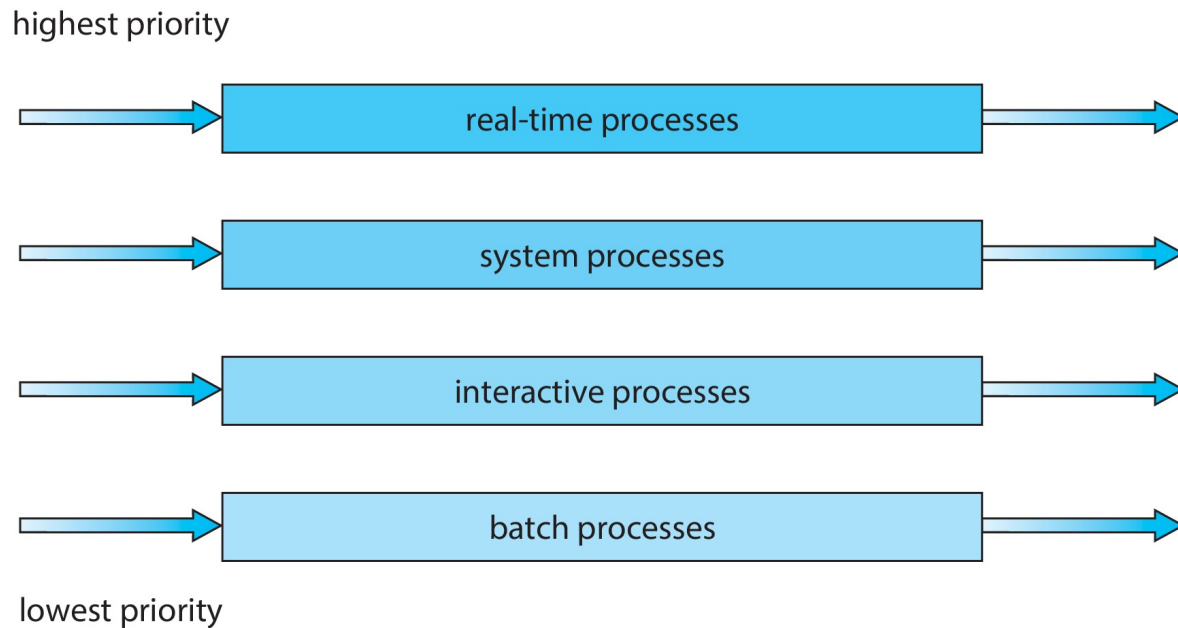
- With priority scheduling, have separate queues for each priority.
- Schedule the process in the highest-priority queue!





Multilevel Queue

- Prioritization based upon process type





Multilevel Feedback Queue

- A process can move between the various queues; aging can be implemented this way
- Multilevel-feedback-queue scheduler defined by the following parameters:
 - number of queues
 - scheduling algorithms for each queue
 - method used to determine when to upgrade a process
 - method used to determine when to demote a process
 - method used to determine which queue a process will enter when that process needs service





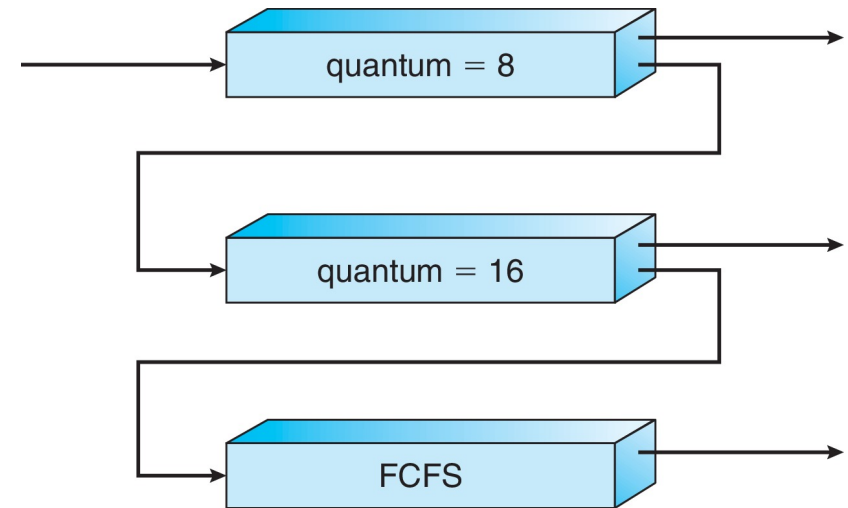
Example of Multilevel Feedback Queue

□ Three queues:

- Q_0 – RR with time quantum 8 milliseconds
- Q_1 – RR time quantum 16 milliseconds
- Q_2 – FCFS

□ Scheduling

- A new job enters queue Q_0 which is served FCFS
 - ▶ When it gains CPU, job receives 8 milliseconds
 - ▶ If it does not finish in 8 milliseconds, job is moved to queue Q_1
- At Q_1 job is again served FCFS and receives 16 additional milliseconds
 - ▶ If it still does not complete, it is preempted and moved to queue Q_2





Multiple-Processor Scheduling

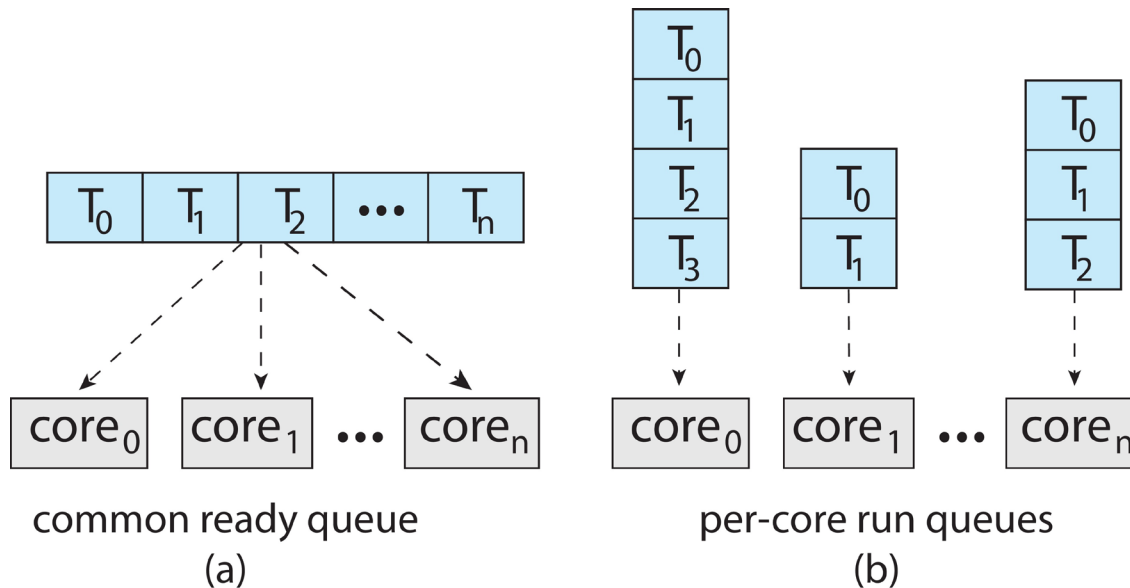
- CPU scheduling more complex when multiple CPUs are available
- Multiprocess may be any one of the following architectures:
 - Multicore CPUs
 - Multithreaded cores
 - NUMA systems
 - Heterogeneous multiprocessing





Multiple-Processor Scheduling

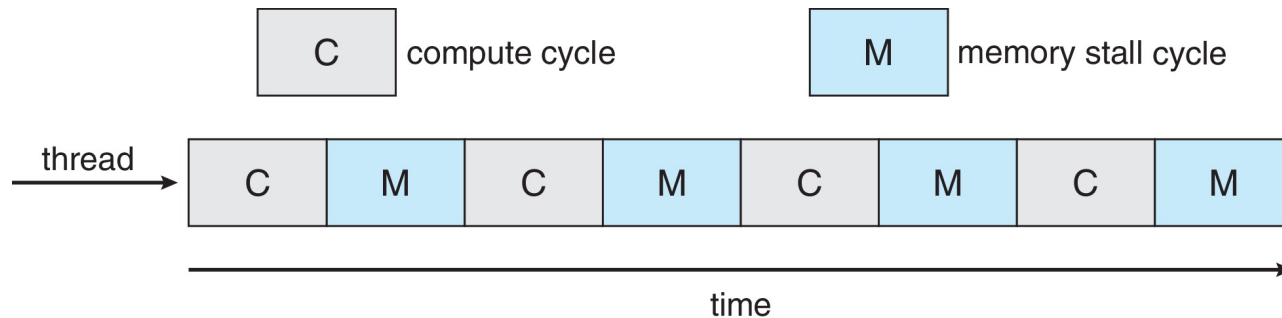
- Symmetric multiprocessing (SMP) is where each processor is self scheduling.
- All threads may be in a common ready queue (a)
- Each processor may have its own private queue of threads (b)





Multicore Processors

- Recent trend to place multiple processor cores on same physical chip
- Faster and consumes less power
- Multiple threads per core also growing
 - Takes advantage of memory stall to make progress on another thread while memory retrieve happens

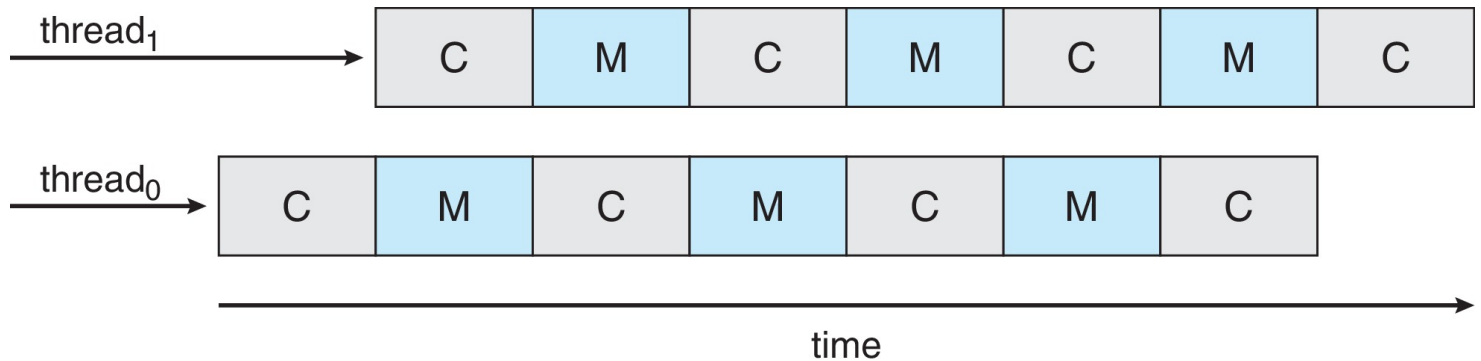




Multithreaded Multicore System

Each core has > 1 hardware threads.

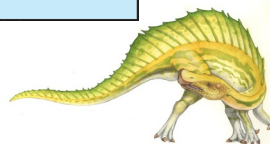
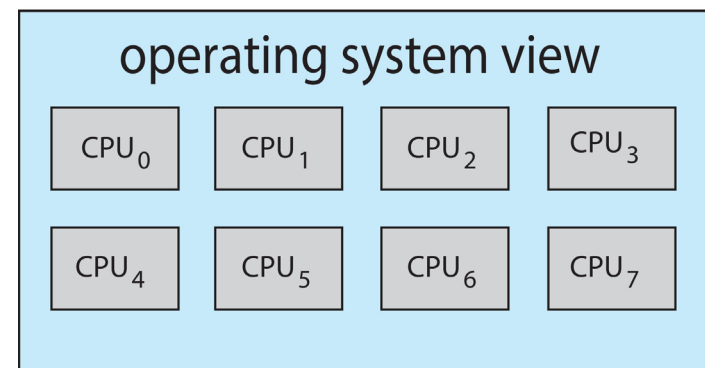
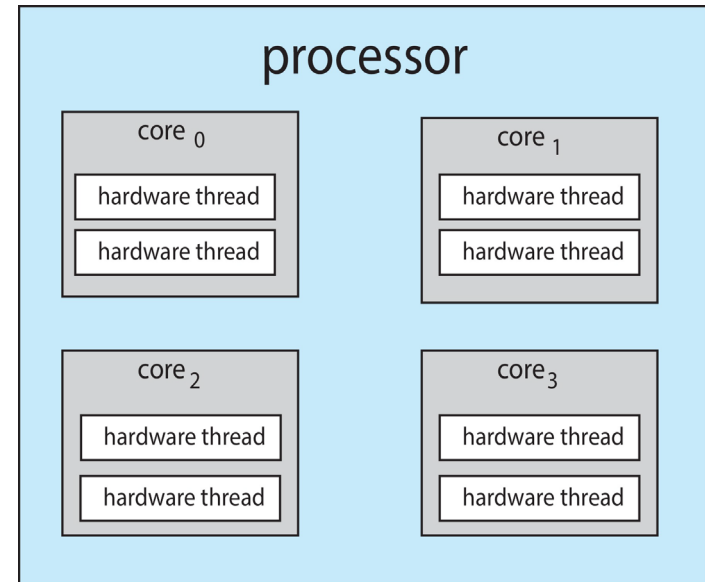
If one thread has a memory stall, switch to another thread!





Multithreaded Multicore System

- ❑ **Chip-multithreading (CMT)** assigns each core multiple hardware threads. (Intel refers to this as **hyperthreading**.)
- ❑ On a quad-core system with 2 hardware threads per core, the operating system sees 8 logical processors.





Multiple-Processor Scheduling – Load Balancing

- If SMP, need to keep all CPUs loaded for efficiency
- **Load balancing** attempts to keep workload evenly distributed
- **Push migration** – periodic task checks load on each processor, and if found pushes task from overloaded CPU to other CPUs
- **Pull migration** – idle processors pulls waiting task from busy processor





Multiple-Processor Scheduling – Processor Affinity

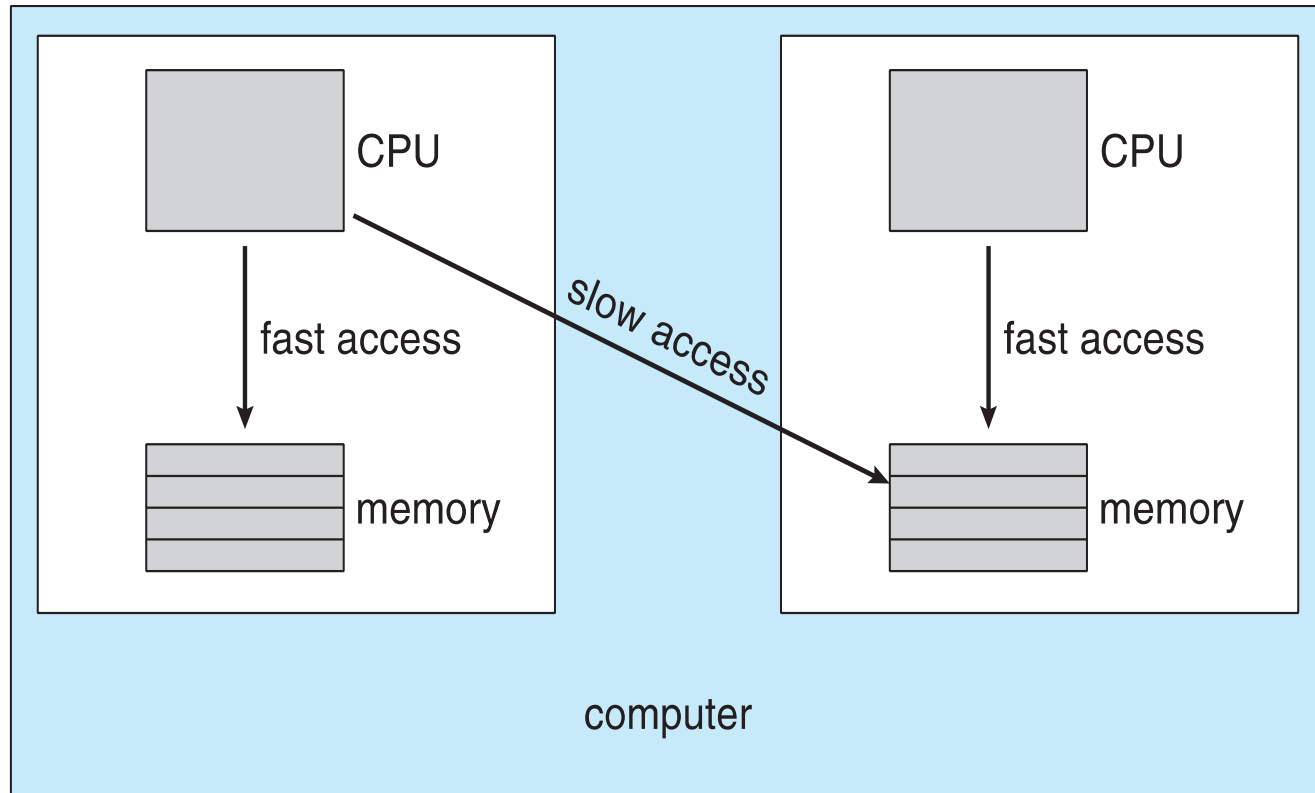
- When a thread has been running on one processor, the cache contents of that processor stores the memory accesses by that thread.
- We refer to this as a thread having affinity for a processor (i.e. “processor affinity”)
- Load balancing may affect processor affinity as a thread may be moved from one processor to another to balance loads, yet that thread loses the contents of what it had in the cache of the processor it was moved off of.
- **Soft affinity** – the operating system attempts to keep a thread running on the same processor, but no guarantees.
- **Hard affinity** – allows a process to specify a set of processors it may run on.





NUMA and CPU Scheduling

If the operating system is **NUMA-aware**, it will assign memory close to the CPU the thread is running on.





Real-Time CPU Scheduling

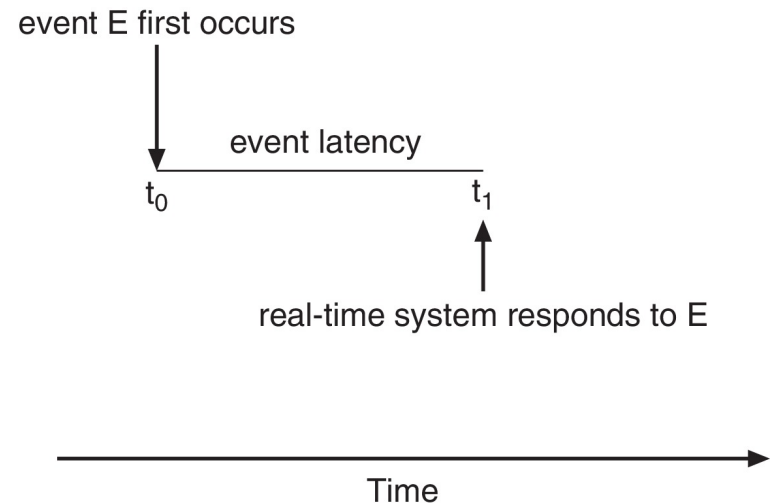
- Can present obvious challenges
- **Soft real-time systems** – Critical real-time tasks have the highest priority, but no guarantee as to when tasks will be scheduled
- **Hard real-time systems** – task must be serviced by its deadline





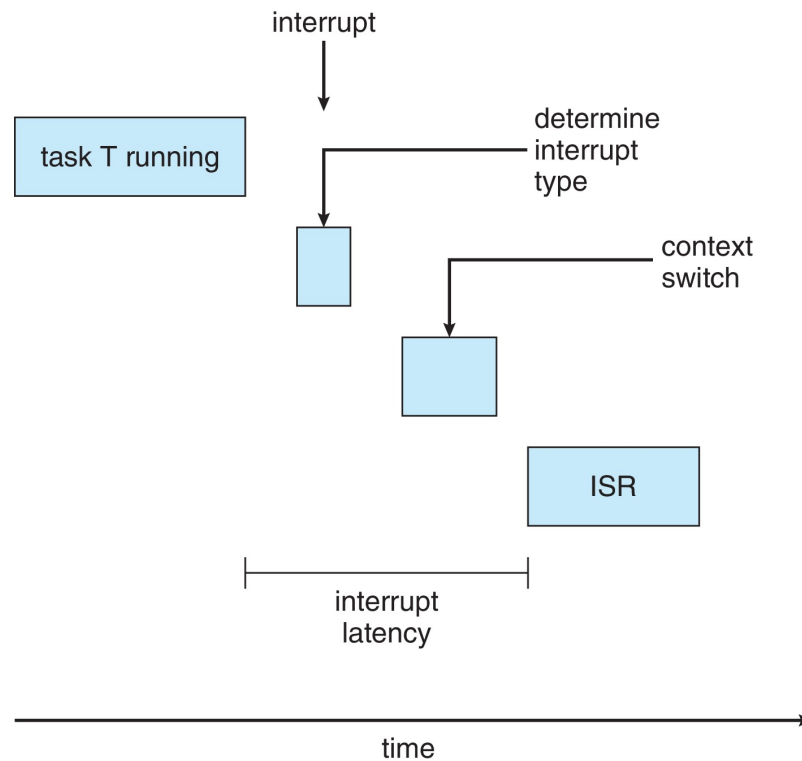
Real-Time CPU Scheduling

- Event latency – the amount of time that elapses from when an event occurs to when it is serviced.
- Two types of latencies affect performance
 1. **Interrupt latency** – time from arrival of interrupt to start of routine that services interrupt
 2. **Dispatch latency** – time for schedule to take current process off CPU and switch to another





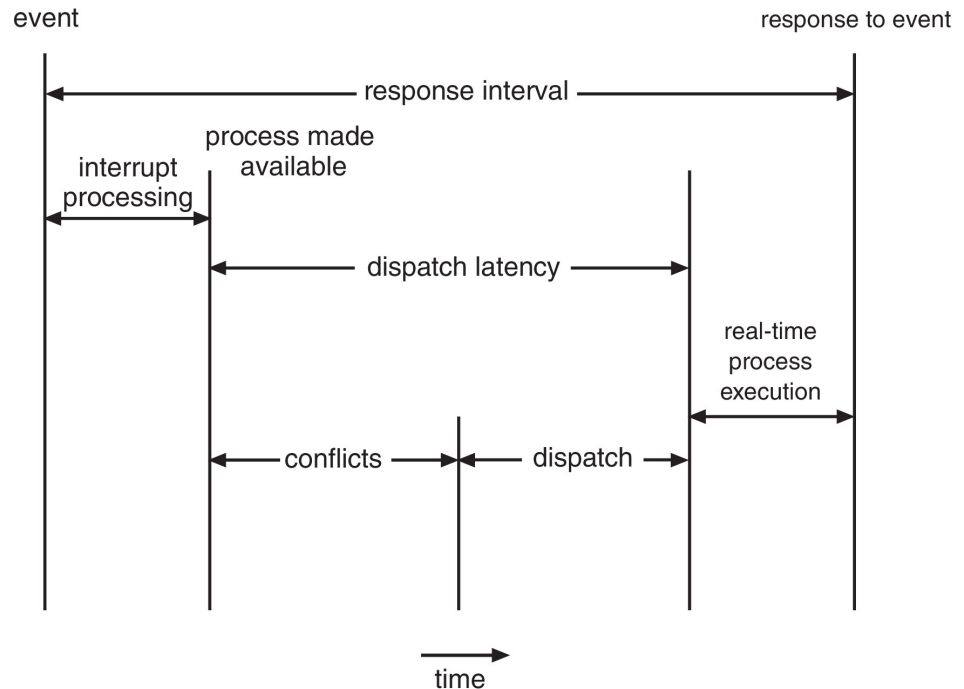
Interrupt Latency





Dispatch Latency

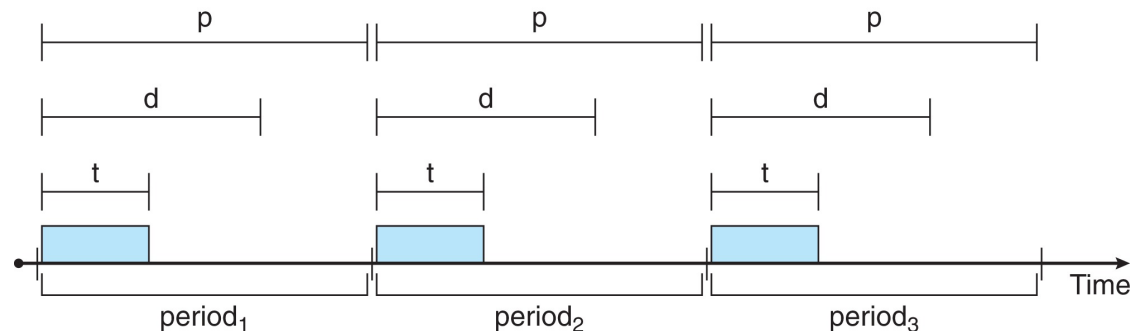
- Conflict phase of dispatch latency:
 1. Preemption of any process running in kernel mode
 2. Release by low-priority process of resources needed by high-priority processes





Priority-based Scheduling

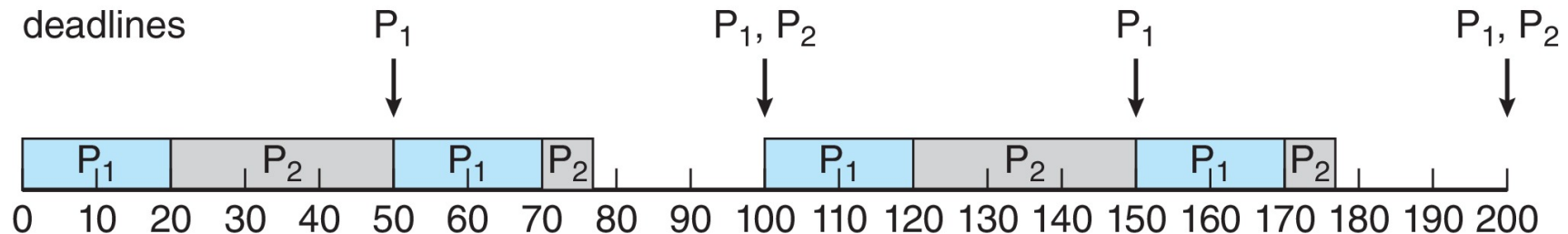
- For real-time scheduling, scheduler must support preemptive, priority-based scheduling
 - But only guarantees soft real-time
- For hard real-time must also provide ability to meet deadlines
- Processes have new characteristics: **periodic** ones require CPU at constant intervals
 - Has processing time t , deadline d , period p
 - $0 \leq t \leq d \leq p$
 - **Rate** of periodic task is $1/p$





Rate Monotonic Scheduling

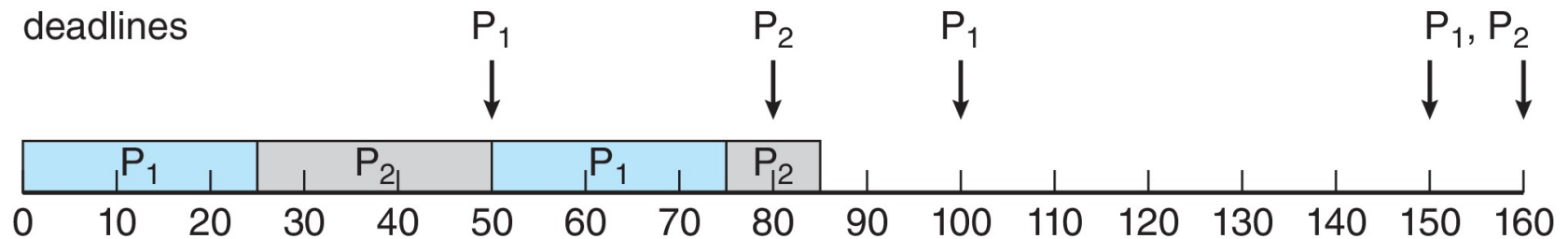
- A priority is assigned based on the inverse of its period
- Shorter periods = higher priority;
- Longer periods = lower priority
- P_1 is assigned a higher priority than P_2 .





Missed Deadlines with Rate Monotonic Scheduling

Process P2 misses finishing its deadline at time 80



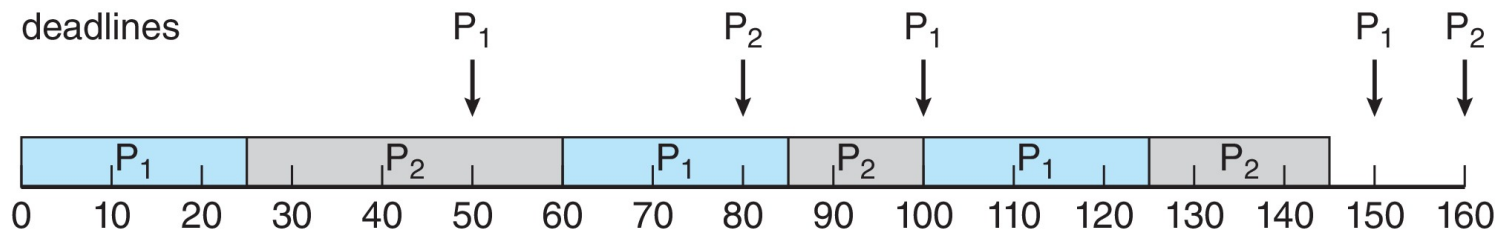


Earliest Deadline First Scheduling (EDF)

- Priorities are assigned according to deadlines:

the earlier the deadline, the higher the priority;

the later the deadline, the lower the priority





Algorithm Evaluation

- How to select CPU-scheduling algorithm for an OS?
- Determine criteria, then evaluate algorithms
- **Deterministic modeling**
 - Type of **analytic evaluation**
 - Takes a particular predetermined workload and defines the performance of each algorithm for that workload
- Consider 5 processes arriving at time 0:

| <u>Process</u> | <u>Burst Time</u> |
|----------------|-------------------|
| P_1 | 10 |
| P_2 | 29 |
| P_3 | 3 |
| P_4 | 7 |
| P_5 | 12 |



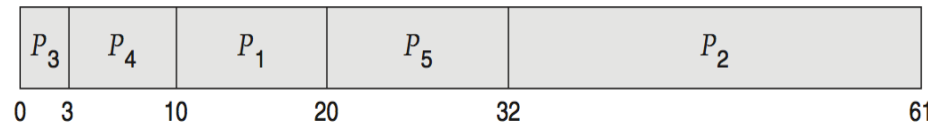


Deterministic Evaluation

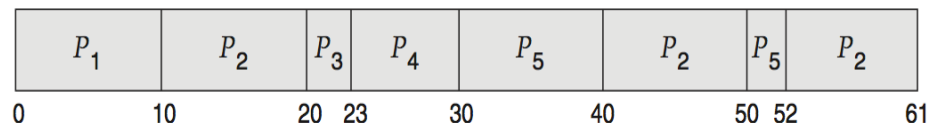
- For each algorithm, calculate minimum average waiting time
- Simple and fast, but requires exact numbers for input, applies only to those inputs
- FCS is 28ms:



- Non-preemptive SJF is 13ms:



- RR is 23ms:





Queueing Models

- Describes the arrival of processes, and CPU and I/O bursts **probabilistically**
 - Commonly exponential, and described by mean
 - Computes average throughput, utilization, waiting time, etc.
- Computer system described as network of servers, each with queue of waiting processes
 - Knowing arrival rates and service rates
 - Computes utilization, average queue length, average wait time, etc.





Little's Formula

- n = average queue length
- W = average waiting time in queue
- λ = average arrival rate into queue
- Little's law – in the steady state, processes leaving queue must equal processes arriving, thus:
$$n = \lambda \times W$$
 - Valid for any scheduling algorithm and arrival distribution
- For example, if on average 7 processes arrive per second, and normally 14 processes in queue, then average wait time per process = 2 seconds





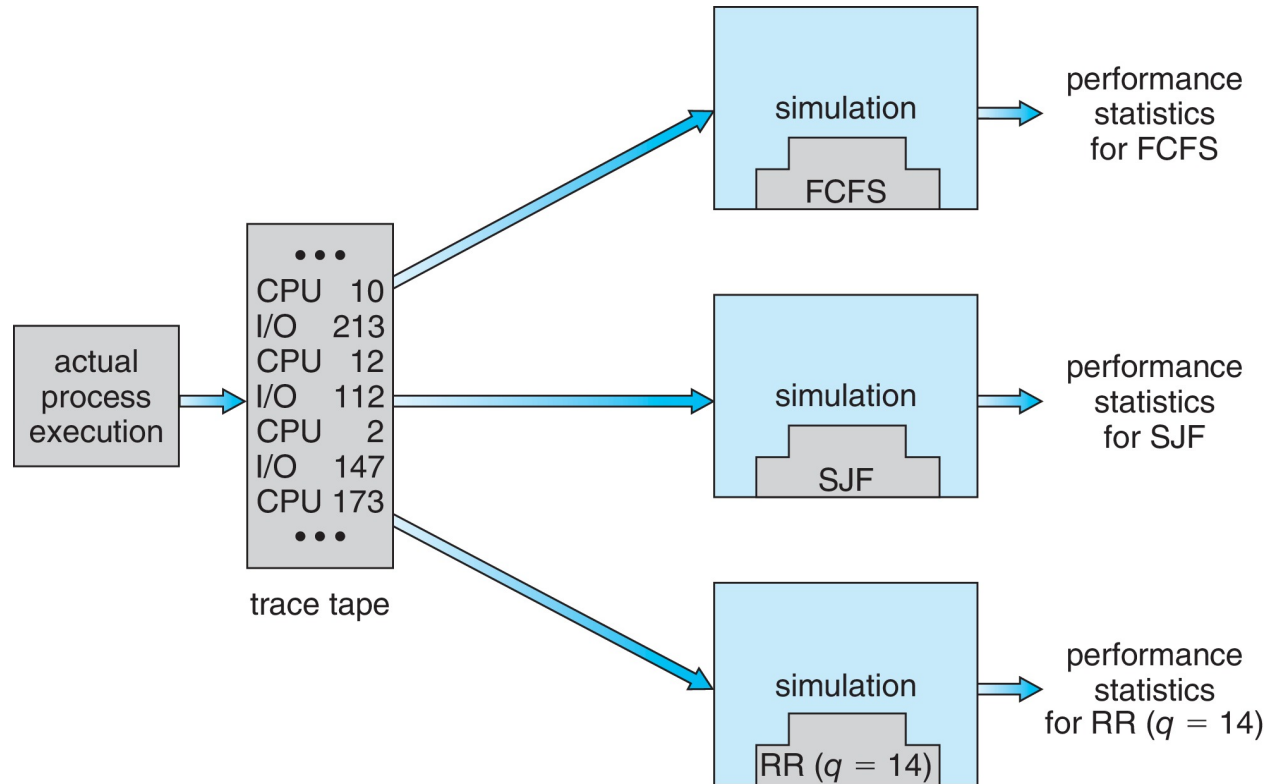
Simulations

- Queueing models limited
- **Simulations** more accurate
 - Programmed model of computer system
 - Clock is a variable
 - Gather statistics, indicating algorithm performance
 - Data to drive simulation gathered via
 - ▶ Random number generator according to probabilities
 - ▶ Distributions defined mathematically or empirically
 - ▶ Trace tapes record sequences of real events in real systems





Evaluation of CPU Schedulers by Simulation





Implementation

- Even simulations have limited accuracy
- **Just implement new scheduler and test in real systems**
 - High cost, high risk
 - Environments vary
- Most flexible schedulers can be modified per-site or per-system
- But again environments vary



End of Chapter 5

