CPU Scheduling

Operating Systems

Yu-Ting Wu

(with slides borrowed from Prof. Jerry Chou)

Outline

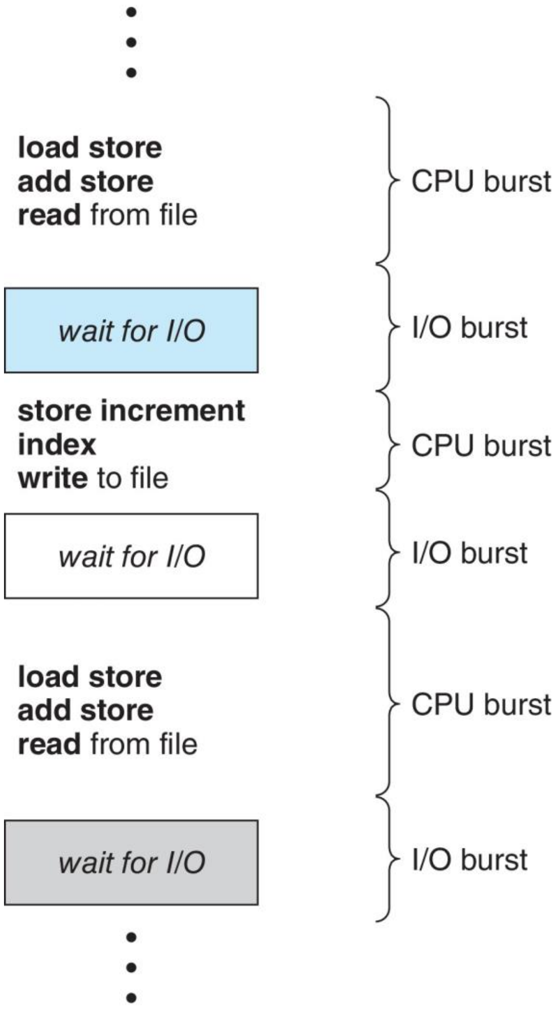
- Basic concepts
- Scheduling algorithms
- Special scheduling issues
- Scheduling case study

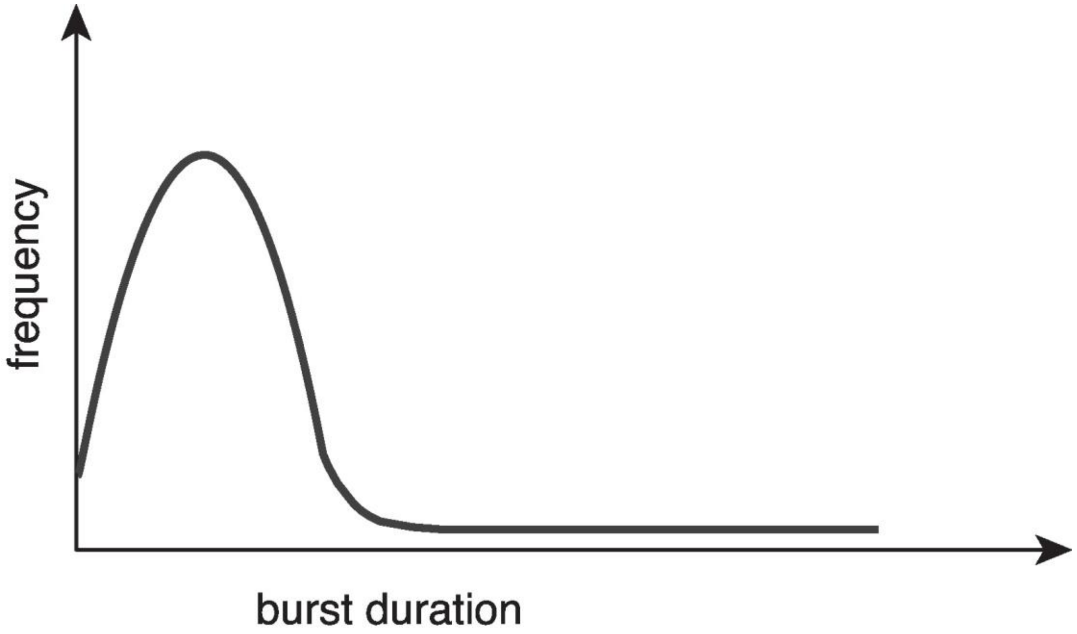
Basic Concepts

Basic Concepts

- The idea of multi-programming
  - Keep several processes in memory
  - Every time one process has to wait, another process takes over the use of the CPU
- **CPU-I/O burst cycle**
  - Process execution consists of a cycle of **CPU execution (CPU burst)** and **I/O wait (I/O burst)**
    - Generally, there is a large number of short CPU bursts, and a small number of long CPU bursts
    - An I/O-bound program would typically has many very short CPU bursts
    - A CPU-bound program might have a few long CPU bursts

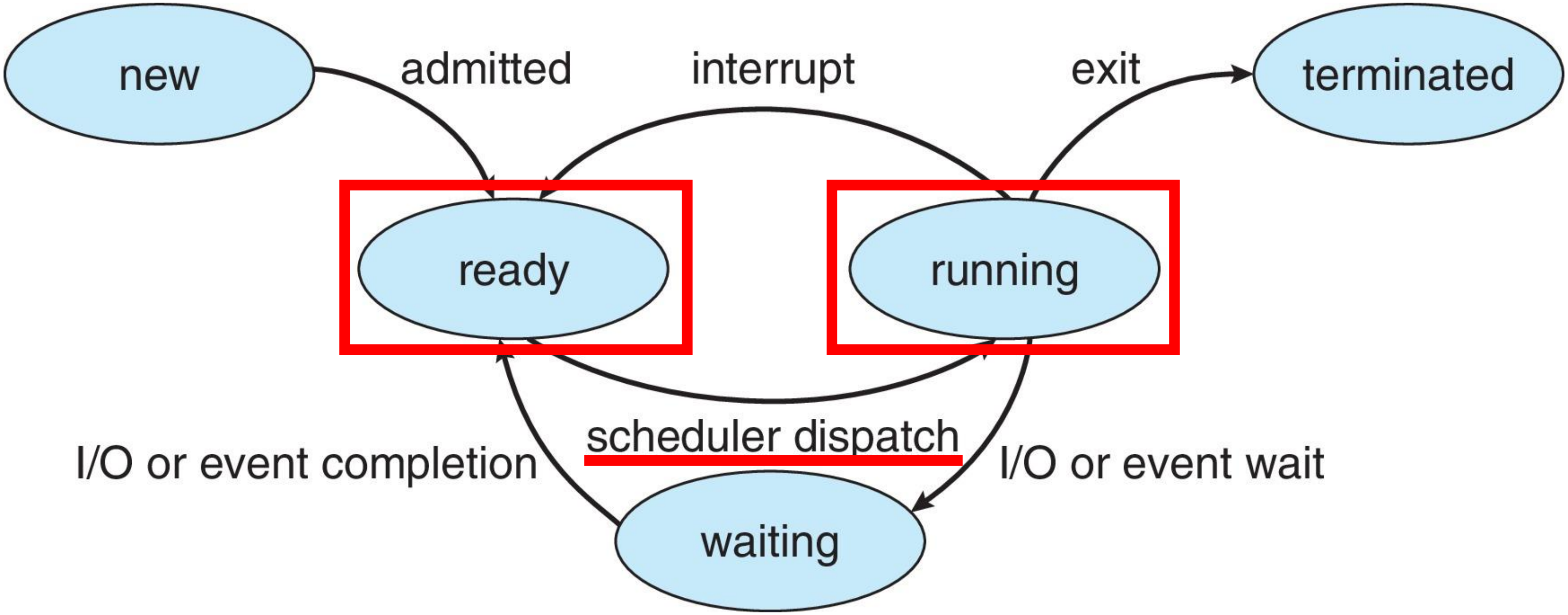
Basic Concepts (cont.)





CPU Scheduler

- Selects process from ready queue to execute
  - Allocate a CPU for the selected process



Preemptive vs Non-preemptive

- CPU scheduling decisions may take place when a process
  - Switches from running to waiting state
  - Switches from running to ready state
  - Switches from waiting to ready
  - Terminates
- **Non-preemptive scheduling**
  - Scheduling under 1 and 4 (no choice in terms of scheduling)
  - The process keeps the CPU until it is **terminated** or **switched to the waiting state**
- **Preemptive scheduling**
  - Scheduling under all cases
  - E.g., Windows 95 and subsequent versions, Mac OS X

Preemptive Issues

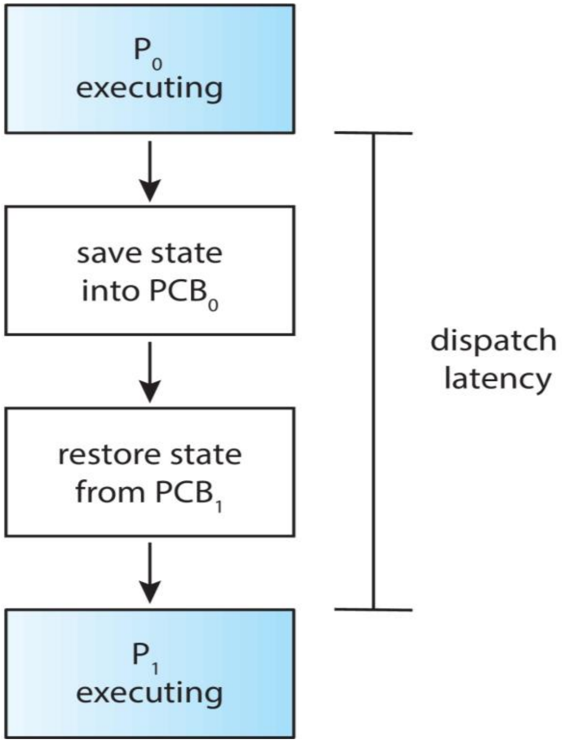
- **Inconsistent state of shared data**
  - Require process synchronization
  - Incur a cost associated with access to the shared data

- Example
  - Two processes share data
  - While one process is updating the data, it is preempted so the second process can run
  - The second process then tries to read the data
  - Inconsistent state happens!

Dispatcher

- Dispatcher module gives control of the CPU to the process selected by scheduler
  - Switch context
  - Jump to the proper location in the selected program

- **Dispatch latency**
  - Time for the dispatcher to stop one process and start another process
    - Context switch time



Scheduling Algorithms

Scheduling Criteria

- **CPU utilization**
  - Theoretically 0% ~ 100%
  - Real systems: 40% (light) ~ 90% (heavy)
- **Throughput**
  - Number of completed processes per time unit

system view

- **Turnaround time**
  - Submission ~ completion
- **Waiting time**
  - Total waiting time in the ready queue
- **Response time**
  - Submission ~ the first response is produced

single job view

Scheduling Criteria

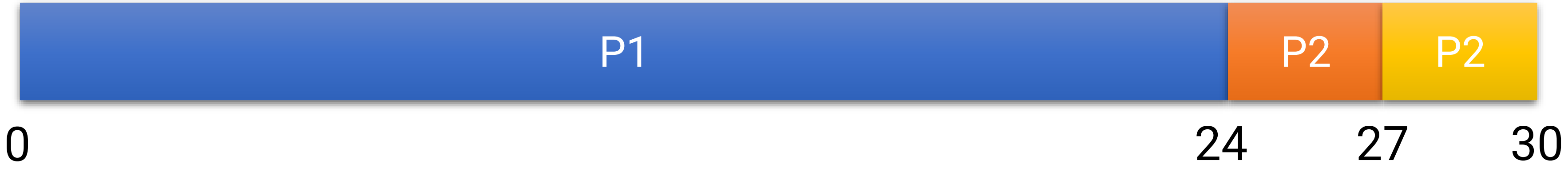
- **Max** CPU utilization
- **Max** Throughput
- **Min** Turnaround time
- **Min** Waiting time
- **Min** Response time

Algorithms

- First-Come, First-Served (FCFS) scheduling
- Shortest-Job-First (SJF) scheduling
- Priority scheduling
- Round-Robin scheduling
- Multi-level queue scheduling
- Multi-level feedback queue scheduling

FCFS Scheduling

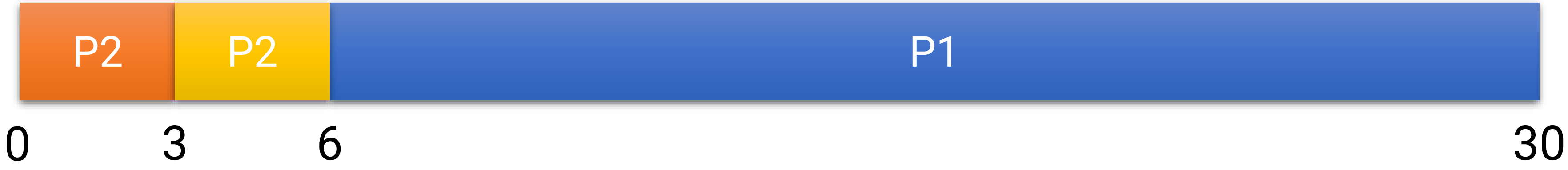
- Process (burst time) in arriving order
  - P1 (24), P2 (3), P3 (3)
- The Gantt Chart of the schedule



- Waiting time: P1 = 0, P2 = 24, P3 = 27
- Average Waiting Time (AWT): (0 + 24 + 27) / 3 = 17
- **Convoy effect**
  - Short processes behind a long process

FCFS Scheduling (cont.)

- Process (burst time) in arriving order
  - P2 (3), P3 (3), P1 (24)
- The Gantt Chart of the schedule



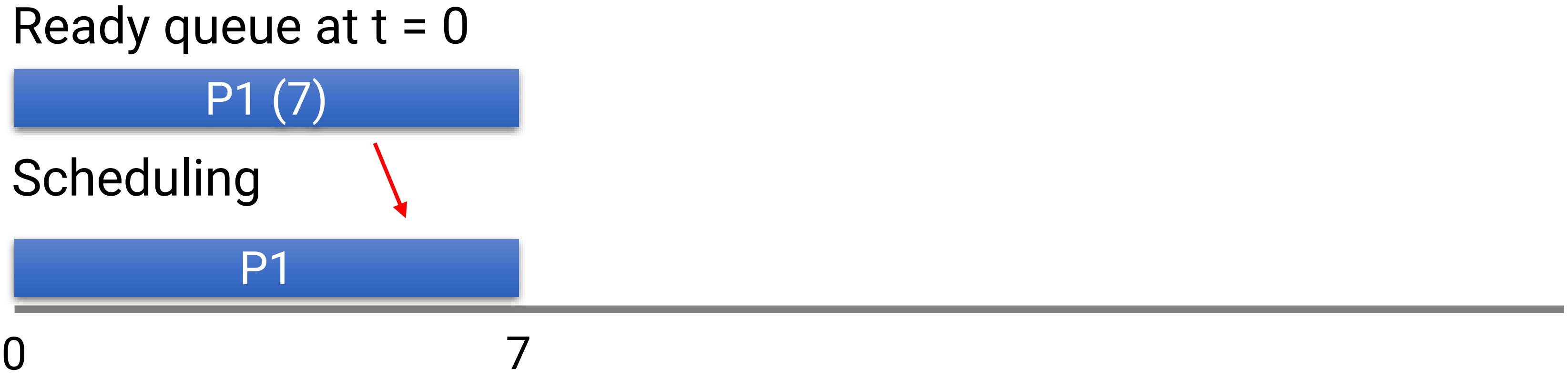
  - Waiting time: P1 = 6, P2 = 0, P3 = 3
  - Average Waiting Time (AWT): (6 + 0 + 3) / 3 = 3

Shortest-Job-First (SJF) Scheduling

- Associate with each process the length of its next CPU burst
- A process with shortest burst length gets the CPU first
- SJF provides the **minimum** (**optimal**) **average waiting time**
- Two schemes
  - **Non-preemptive**
    - Once the CPU is given to a process, it cannot be preempted until its completion
  - **Preemptive**
    - If a new process arrives with shorter burst length, preemption happens

Non-Preemptive SJF Example

Process	Arrival Time	Burst Time
P1	0	7
P2	2	4
P3	4	1
P4	5	4



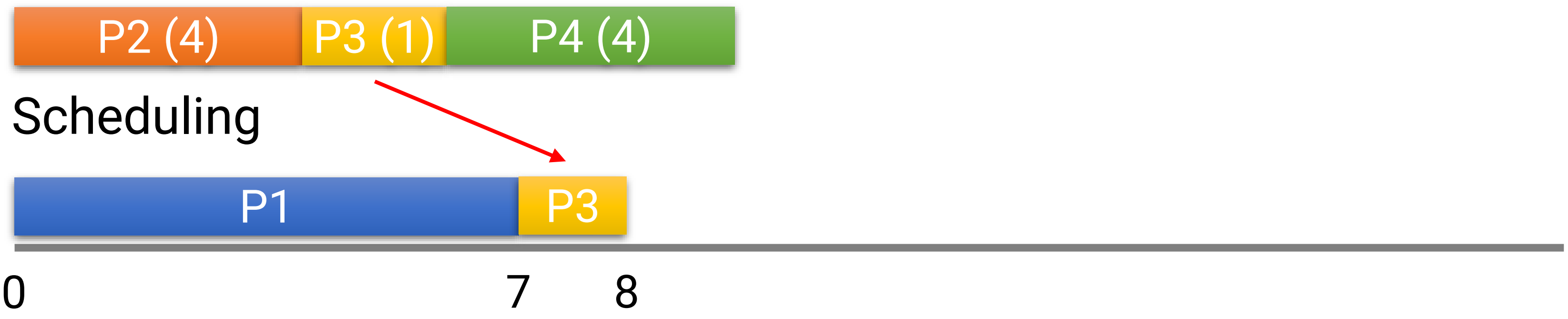
Ready queue at t = 0

Scheduling

Non-Preemptive SJF Example (cont.)

Process	Arrival Time	Burst Time
P1	0	7
P2	2	4
P3	4	1
P4	5	4

Ready queue at t = 7

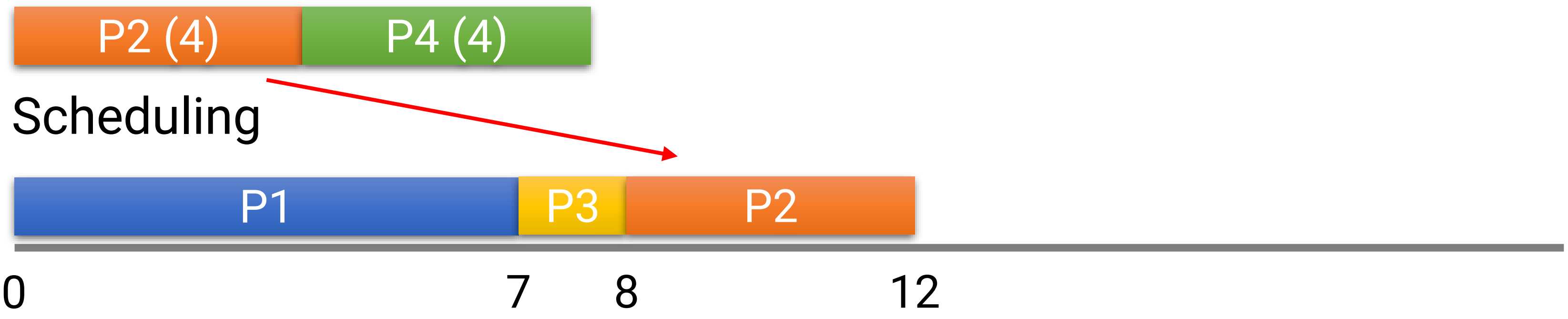


Scheduling

Non-Preemptive SJF Example (cont.)

Process	Arrival Time	Burst Time
P1	0	7
P2	2	4
P3	4	1
P4	5	4

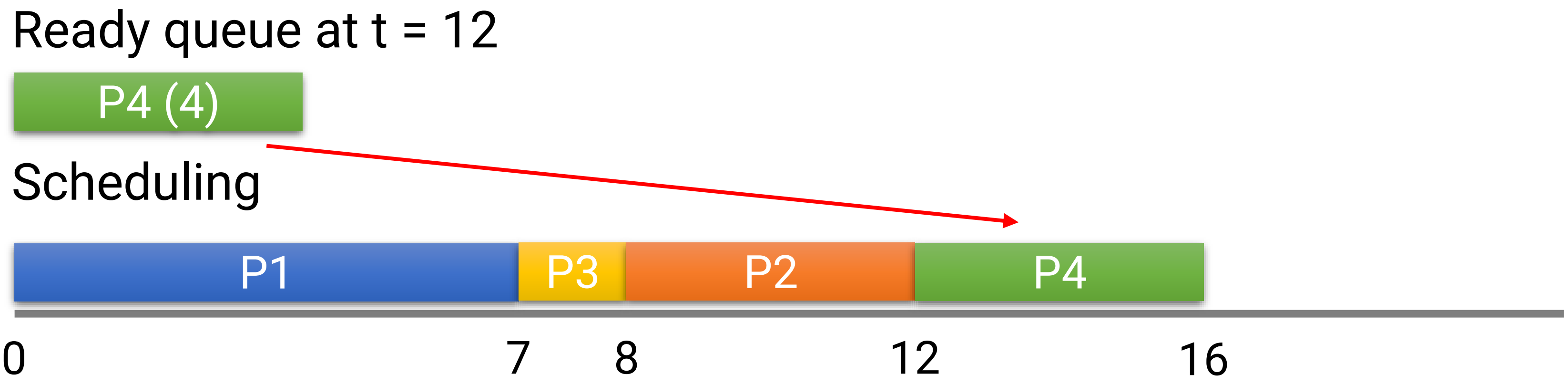
Ready queue at t = 8



Scheduling

Non-Preemptive SJF Example (cont.)

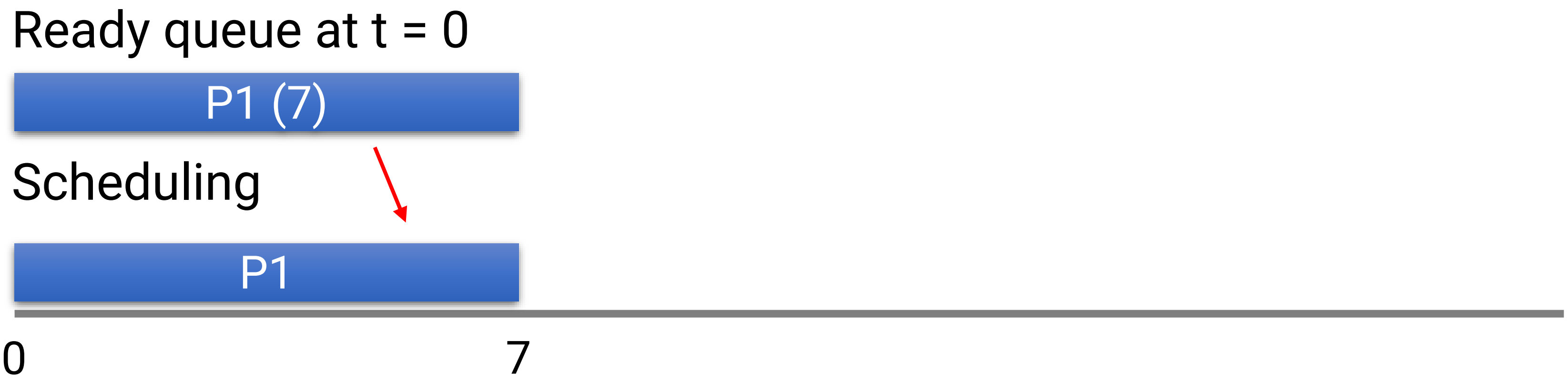
Process	Arrival Time	Burst Time
P1	0	7
P2	2	4
P3	4	1
P4	5	4



- **Wait time = completion time - arrival time - run time**
- **AWT** = [(7-0-7) + (12-2-4) + (8-4-1) + (16-5-4)] / 4 = 4
- **Response time**: P1 = 0, P2 = 6, P3 = 3, P4 = 7

Preemptive SJF Example

Process	Arrival Time	Burst Time
P1	0	7
P2	2	4
P3	4	1
P4	5	4

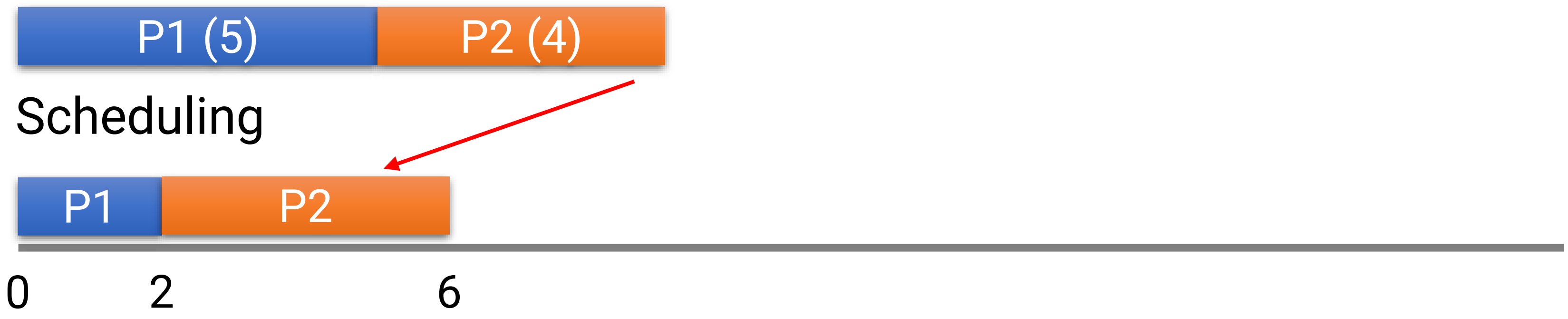

Ready queue at t = 0

Scheduling

Preemptive SJF Example

Process	Arrival Time	Burst Time
P1	0	7
P2	2	4
P3	4	1
P4	5	4

Ready queue at t = 2

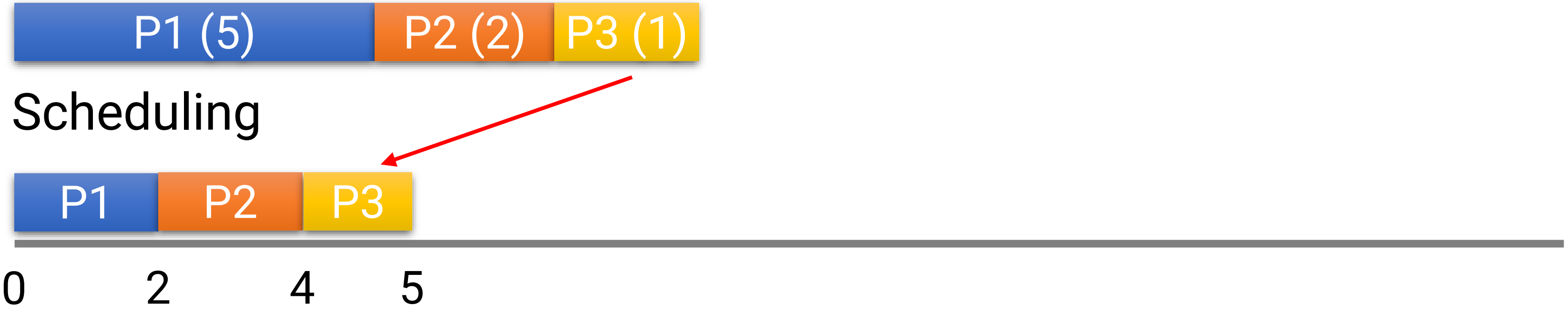


Preemptive SJF Example

Process	Arrival Time	Burst Time
P1	0	7
P2	2	4
P3	4	1
P4	5	4

Ready queue at t = 4

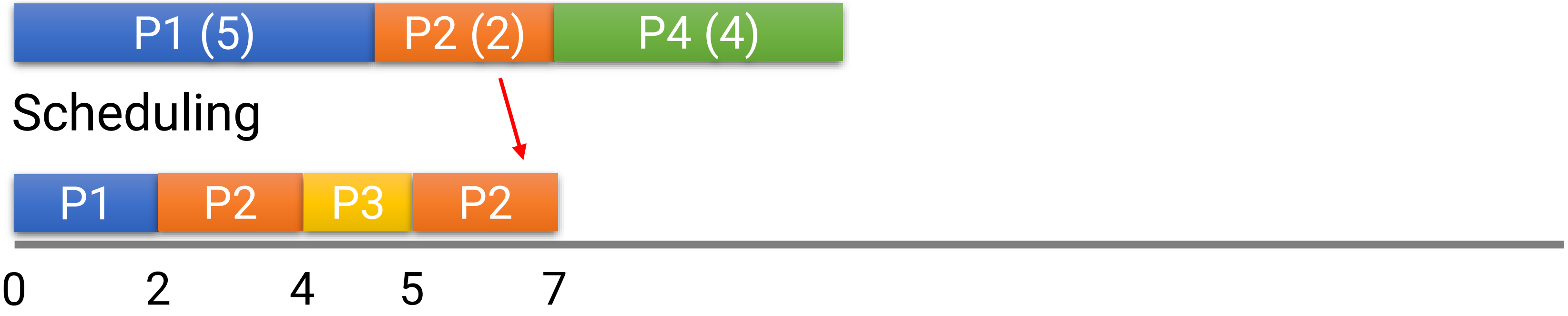
Scheduling



Preemptive SJF Example

Process	Arrival Time	Burst Time
P1	0	7
P2	2	4
P3	4	1
P4	5	4

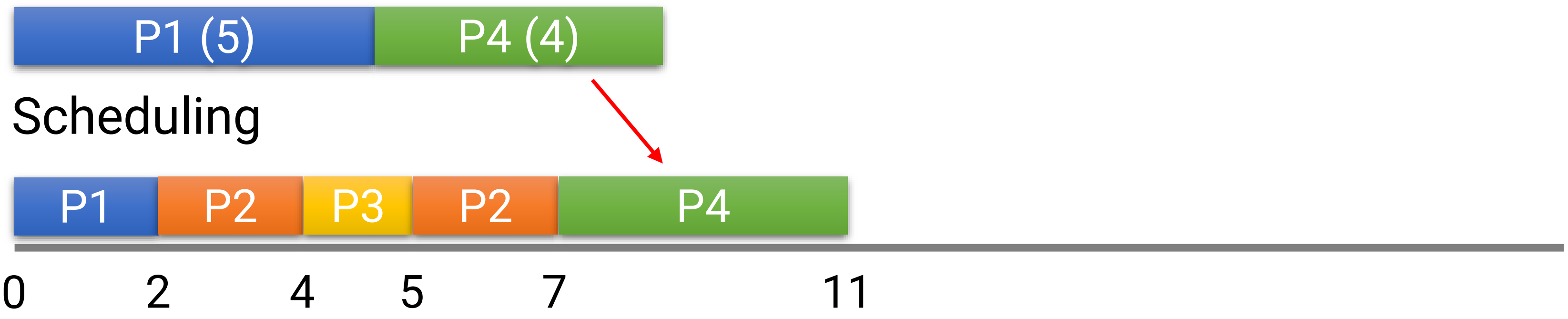
Ready queue at t = 5



Preemptive SJF Example

Process	Arrival Time	Burst Time
P1	0	7
P2	2	4
P3	4	1
P4	5	4

Ready queue at t = 7



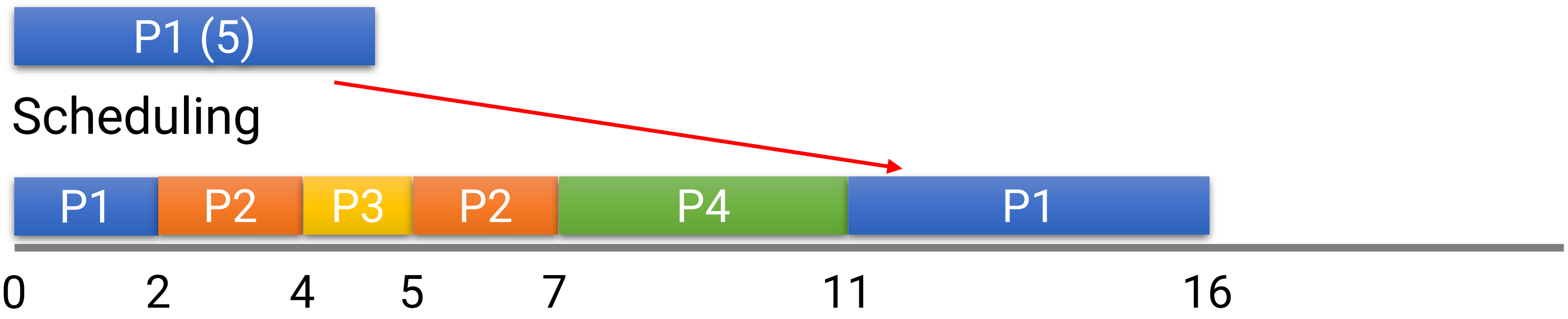
Scheduling

Preemptive SJF Example

Process	Arrival Time	Burst Time
P1	0	7
P2	2	4
P3	4	1
P4	5	4

Ready queue at t = 11

Scheduling



- **Wait time = completion time - arrival time - run time**
- **AWT** = [(16-0-7) + (7-2-4) + (5-4-1) + (11-5-4)] / 4 = 3
- **Response time**: P1 = 0, P2 = 0, P3 = 0, P4 = 2

Approximate Shortest-Job-First (SJF)

- **SJF difficulty**
  - No way to know length of the next CPU burst
- **Approximate SJF**
  - The next burst can be predicted as an exponential average of the measured length of previous CPU bursts

$$\tau_{n+1} = \alpha \underset{\text{new one}}{\underline{t_n}} + (1-\alpha)\underset{\text{history}}{\underline{\tau_n}}$$

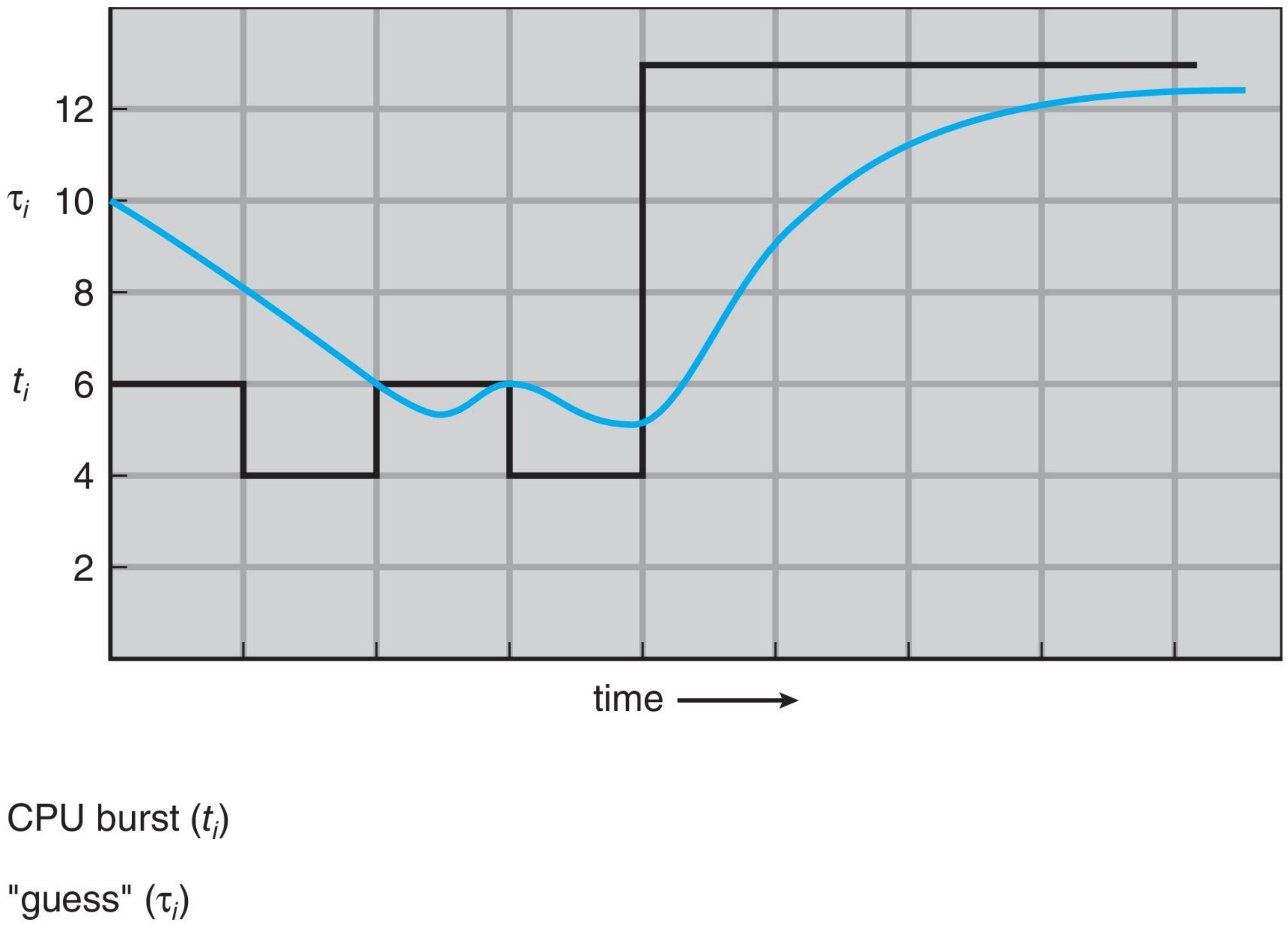
$$= \alpha t_n + (1-\alpha)\alpha t_{n-1} + (1-\alpha)^2 \alpha t_{n-2} + \ldots$$

Example:

$\alpha = 1/2$

$$= (\frac{1}{2})t_n + (\frac{1}{2})^2 t_{n-1} + (\frac{1}{2})^3 t_{n-2}$$

Exponential Prediction of Next CPU Burst



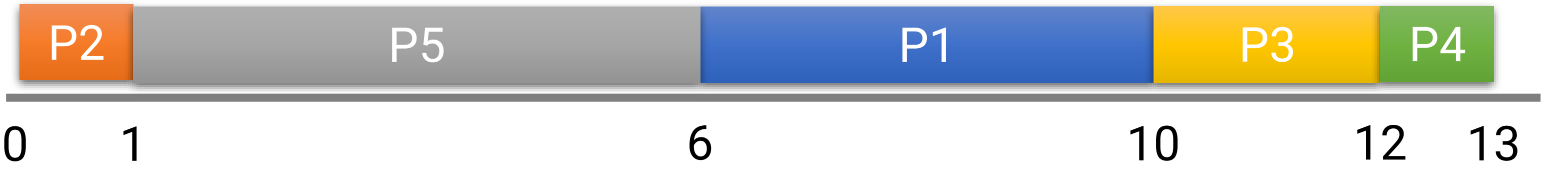
Priority Scheduling

- A **priority number** is associated with each process
- The CPU is allocated to the highest priority process
  - Preemptive
  - Non-preemptive
- **SJF is a priority scheduling** where priority is the predicted next CPU burst time
- Problem: **starvation**
  - Low priority processes never execute
  - Example: IBM 7094 shutdown at 1973, a 1967-process never run
  - Solution: **aging**
    - As time progresses, increase the priority of processes

Priority Scheduling (cont.)

Process	Burst Time	Priority
P1	4	3
P2	1	1
P3	2	4
P4	1	5
P5	5	2

Gantt Chart



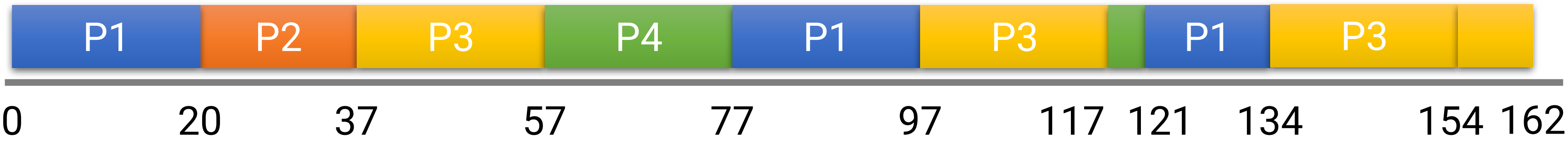
Round-Robin (RR) Scheduling

- Each process gets a small unit of CPU time (**time quantum**, TQ), usually 10 ~ 100 ms
  - Context switch time usually < 10 microseconds
- After TQ elapsed, 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*, each process gets *1/n* of the CPU time (q time units)
  - No process waits more than *(n-1)q* time units
- Performance
  - TQ large ➔ FIFO
  - TQ small ➔ (context switch) overhead increases

Round-Robin (RR) Scheduling (cont.)

Process	Burst Time
P1	53
P2	17
P3	68
P4	24

- If TQ = 20, the Gantt Chart is



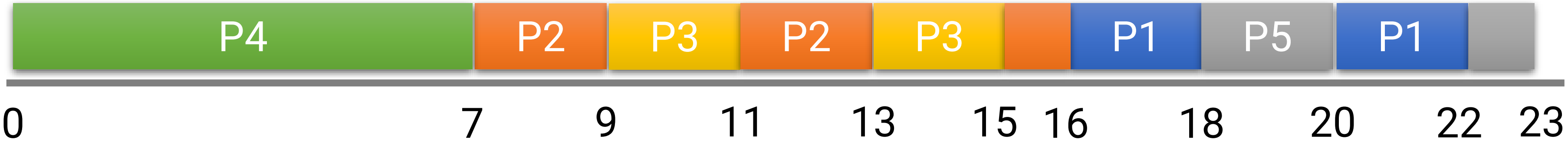
- Typically, higher average turnaround than SJF, but better response

Priority Scheduling with Round-Robin

Process	Burst Time	Priority
P1	4	3
P2	5	2
P3	4	2
P4	7	1
P5	3	3

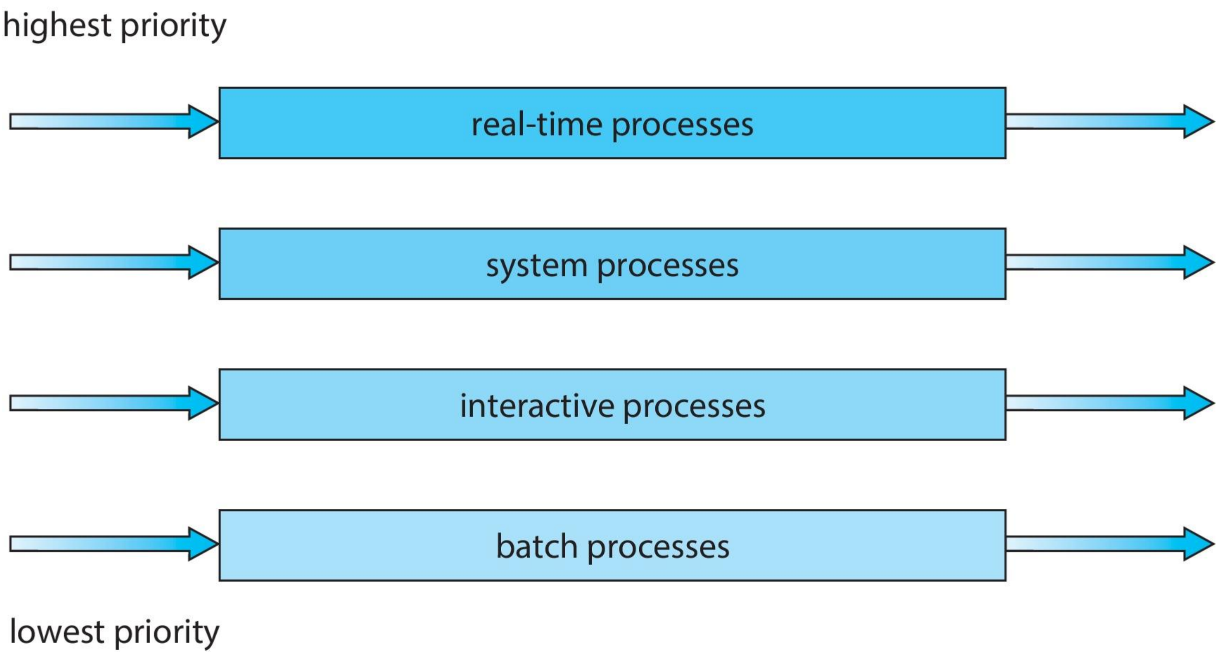
- Run the process with the highest priority
- Processes with the same priority run round-robin

Gantt Chart (TQ = 2)



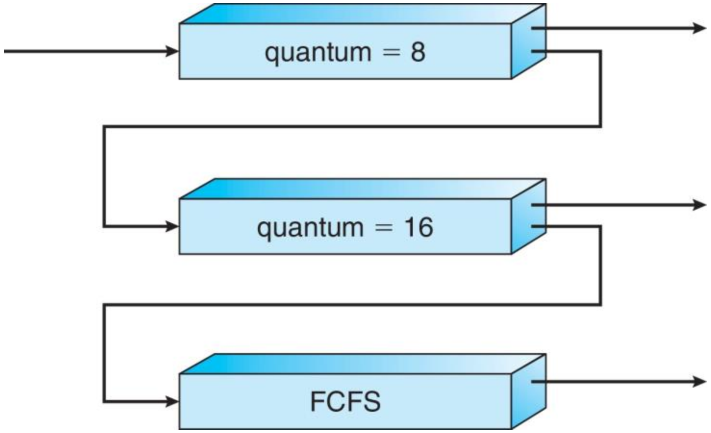
Multi-level Queue Scheduling

- Ready queue is partitioned into separate queues
- Each queue has its own scheduling algorithm
- Scheduling must be done **between** queues
  - Time slice: each queue gets a certain amount of CPU



Multi-level Feedback Queue Scheduling

- A process can move between the various queues
  - Aging must be implemented
- Idea: separate processes according to the characteristic of their CPU burst
  - **I/O-bound** and **interactive processes** in **higher priority** queue ➔ short CPU burst
  - **CPU-bound** processes in **lower priority** queue ➔ long CPU burst

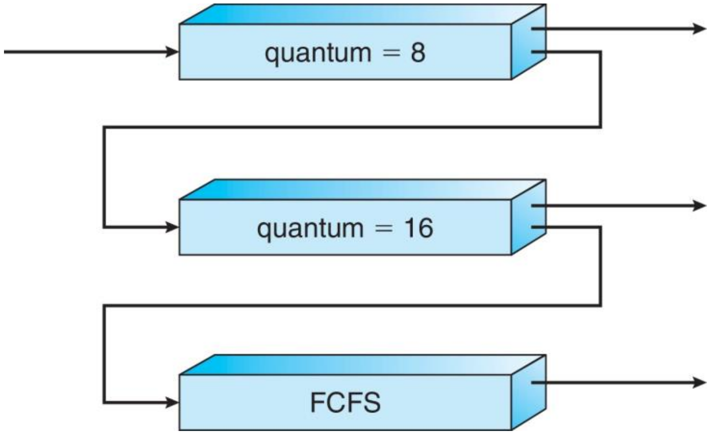


Multi-level Feedback Queue Scheduling

- In general, multi-level feedback queue scheduler is defined by the following parameters
  - Number of queues
  - Scheduling algorithm for each queue
  - Method used to determine when to upgrade a process
  - Method used to determine when to demote a process

Multi-level Feedback Queue (cont.)

- Three queues
  - Q0: RR with TQ 8 ms.
  - Q1: RR with TQ 16 ms.
  - Q2: FCFS



- Scheduling
  - A new process enters queue Q0 which is served in RR
    - When it gains CPU, the process receives 8 ms
    - If it does not finish in 8 ms., the process is moved to queue Q1
  - At Q1, job is again served in RR and receives 16 additional ms.
    - If it still does not complete, it is preempted and moved to queue Q2

Evaluation Methods

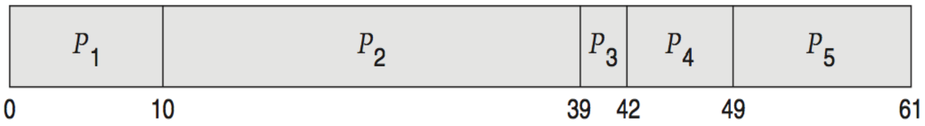
- **Deterministic modeling**
  - Take a particular predetermined workload and define the performance of each algorithm for that workload
- **Queueing model**
  - Mathematical analysis
- **Simulation**
  - Random number generator or trace tapes for workload generation
- **Implementation**
  - The only completely accurate way for algorithm evaluation

Deterministic Modeling

- Take a particular predetermined workload and define the performance of each algorithm for that workload
- Example: 5 processes arriving at time 0

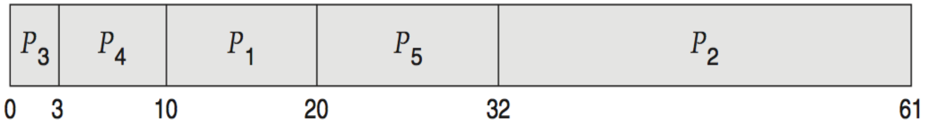
	P1	P2	P3	P4	P5
Burst Time	10	29	3	7	12

FCFS



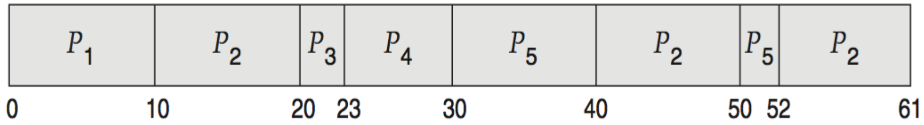
AWT = 28

Non-preemptive SFJ



AWT = 13

RR (TQ = 23)



AWT = 22

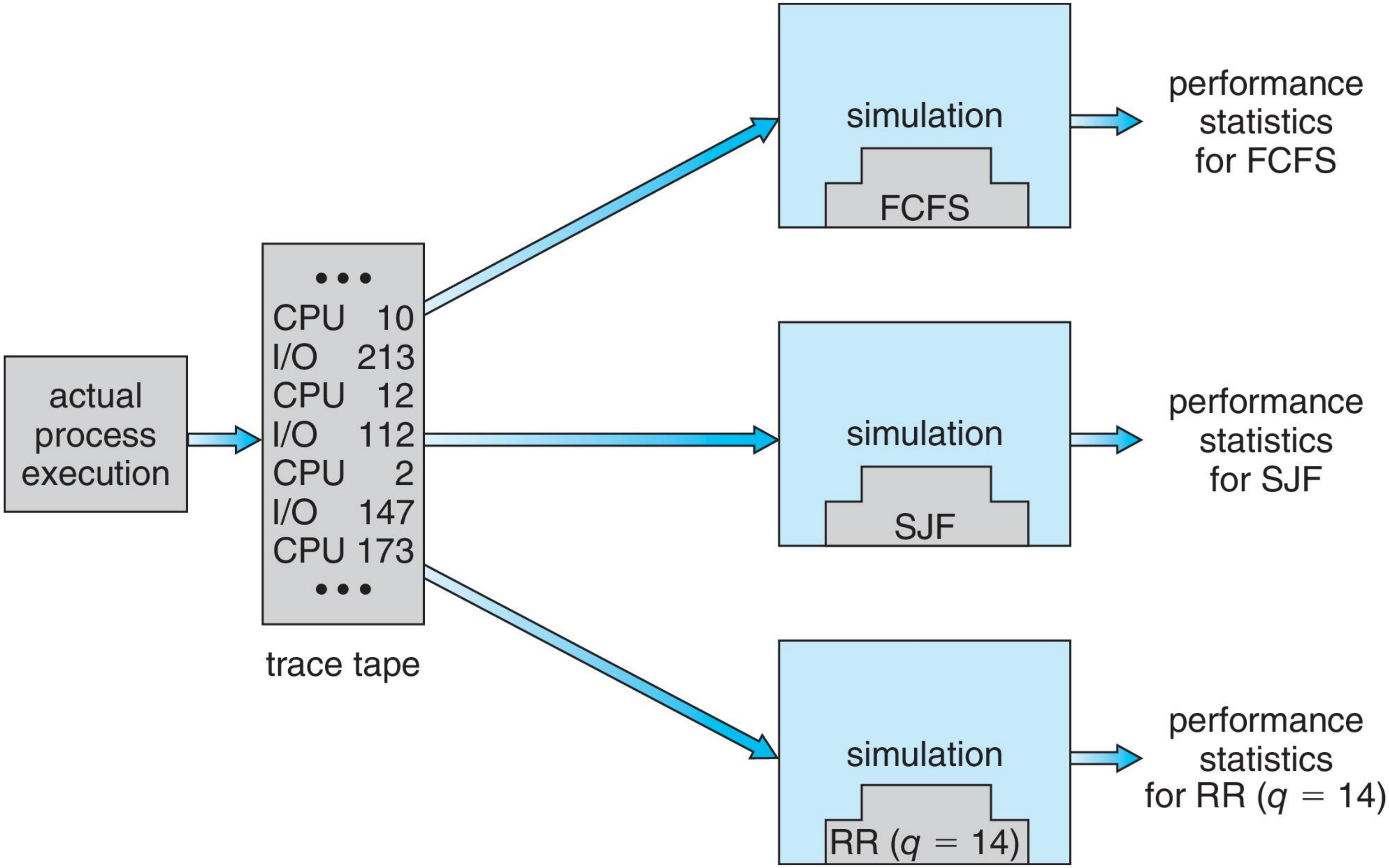
Queueing Models

- Describe the arrival of processes, and CPU and I/O bursts **probabilistically**
- Compute average throughput, utilization, waiting time, etc.

Simulations

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

Simulations (cont.)



actual process execution

• • •
CPU 10
I/O 213
CPU 12
I/O 112
CPU 2
I/O 147
CPU 173
• • •

trace tape

simulation
FCFS
performance statistics for FCFS
simulation
SJF
performance statistics for SJF
simulation
RR (q = 14)
performance statistics for RR (q = 14)

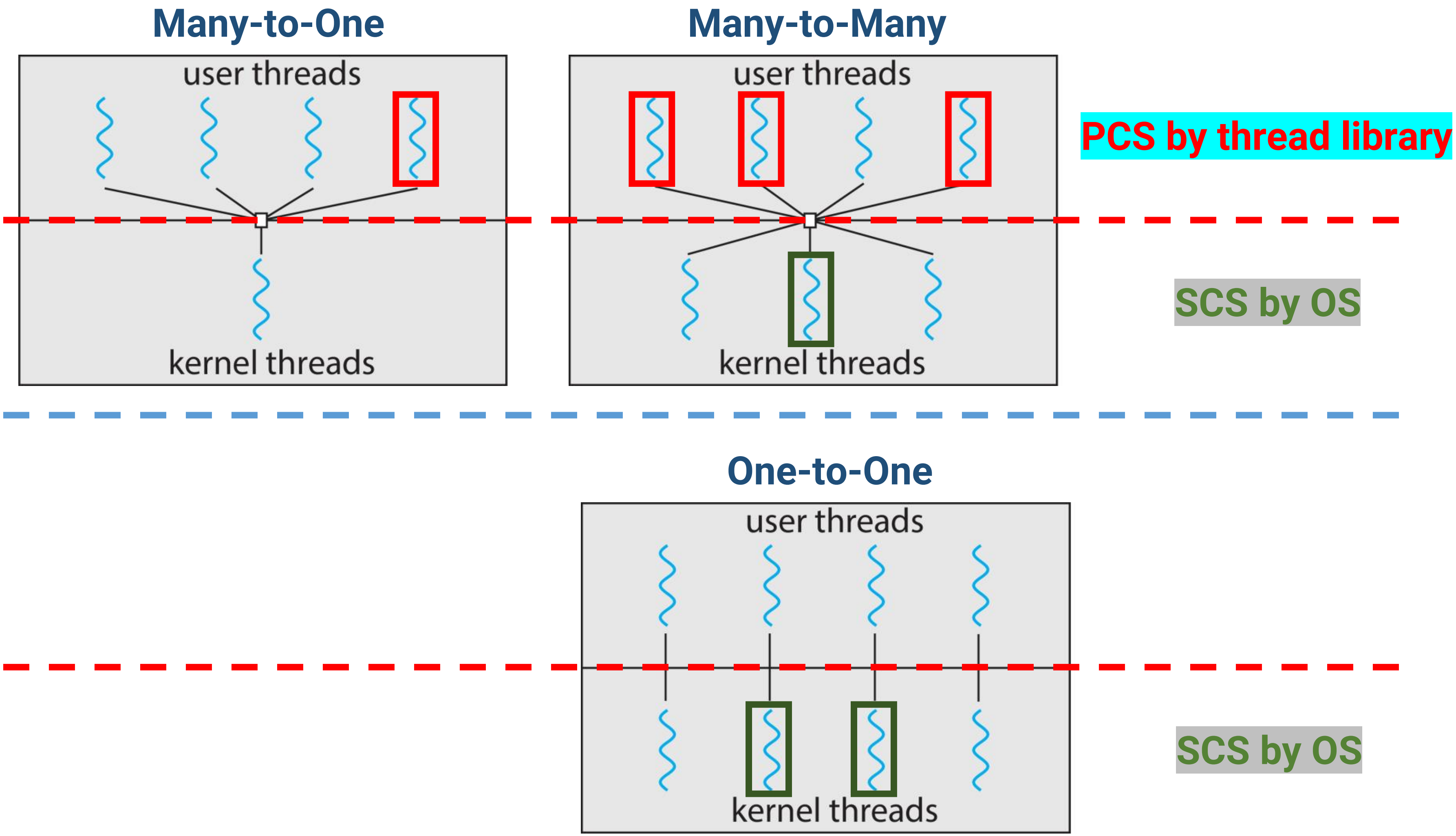
Implementation

- Even simulations have limited accuracy
- The only completely accurate way for algorithm evaluation is to implement new scheduler and test in real systems
  - High cost and risk
  - Also need to consider the varieties of environments

Thread Scheduling

- Distinction between user-level and kernel-level threads
- When threads supported, threads scheduled, not processes
- For **many-to-one** and **many-to-many** models, **thread library** schedules **user-level threads** to run on light-weight process
  - Known as **process-contention scope (PCS)** since scheduling competition is within the process
  - Typically done via **priority** set by programmer
- Kernel thread scheduled onto available CPU is **system-contention scope (SCS)** – competition among all threads in system

Thread Scheduling (cont.)



Many-to-One

Many-to-Many

PCS by thread library

SCS by OS

One-to-One

SCS by OS

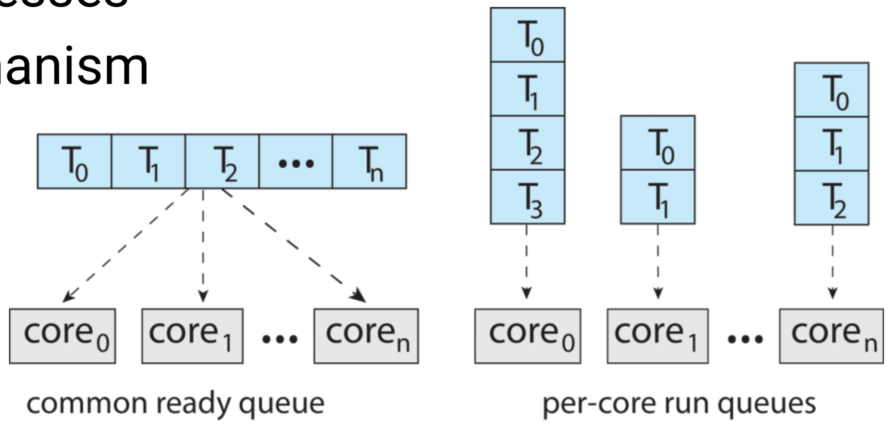
Special Scheduling Issues

Special Scheduling Issues

- Multi-processor scheduling
- Multi-core processor scheduling
- Real-time scheduling

Multi-Processor Scheduling

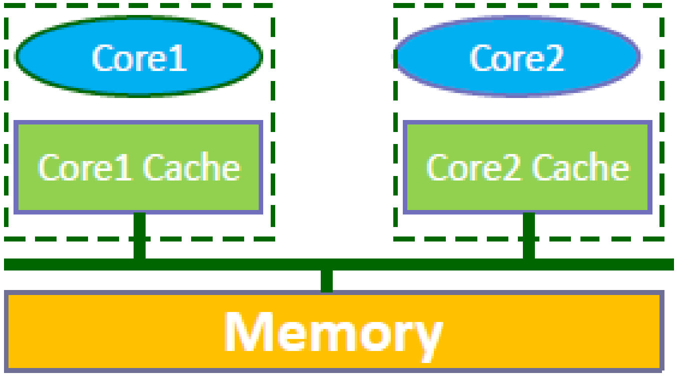
- **Asymmetric multi-processing**
  - All system activities are handled by a processor (alleviating the need for data sharing)
  - The others only execute user code (allocated by the master)
- **Symmetric multi-processing (SMP)**
  - Each processor is self-scheduling
  - All processes in common ready queue, or each has its own private queue of ready processes
  - Need synchronization mechanism



Processor Affinity

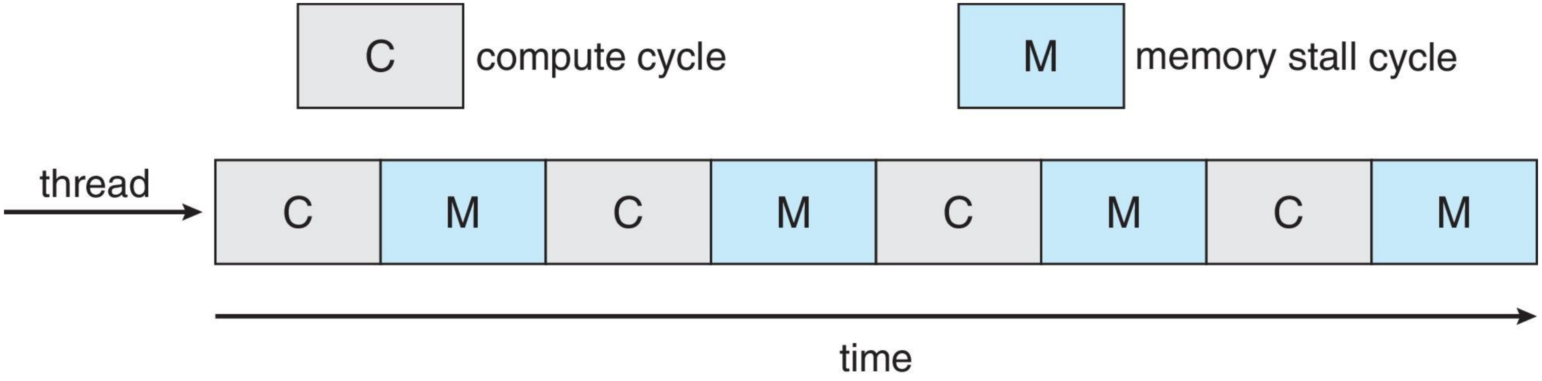
- **Processor affinity**
  - A process has an affinity for the processor on which it is currently running
    - A process populates its recent used data in cache
    - Cache invalidation and repopulation has high cost

- Solution
  - **Soft affinity**
    - Possible to migrate to other processors
  - **Hard affinity**
    - Not to migrate to other processor



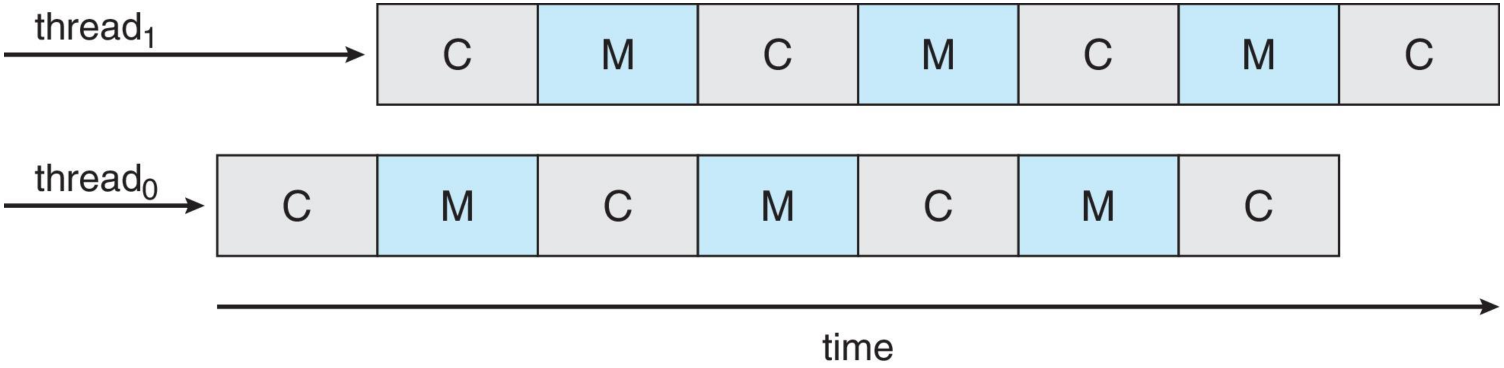
Multi-core Processors

- Recent trend to place multiple processor cores **on same physical chip**
- **Faster** and **consumes less power**
- Memory stall
  - When access memory, it spends a significant amount of time waiting for the data become available (e.g., cache miss)



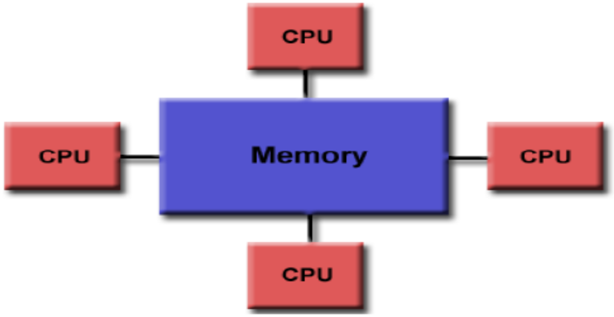
Multi-threaded Multi-core System

- Two (or more) hardware threads are assigned to each core
  - Each core has > 1 hardware threads
- Take advantage of memory stall to make progress on another thread while memory retrieve happens
  - If one thread has a memory stall, switch to another thread

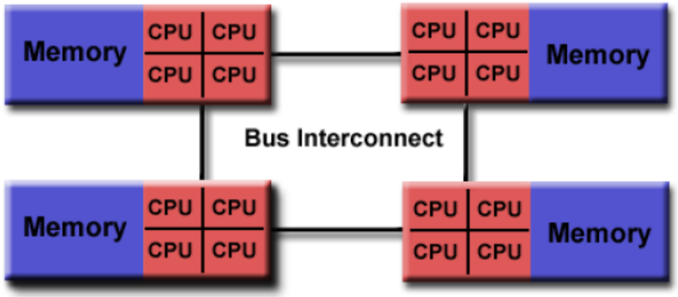


Memory Access Architecture

- **Uniform Memory Access (UMA)**
  - Most commonly represented today by Symmetric multi-processor (SMP) machines
  - Equal access times to memory
  - Example: most commodity computers

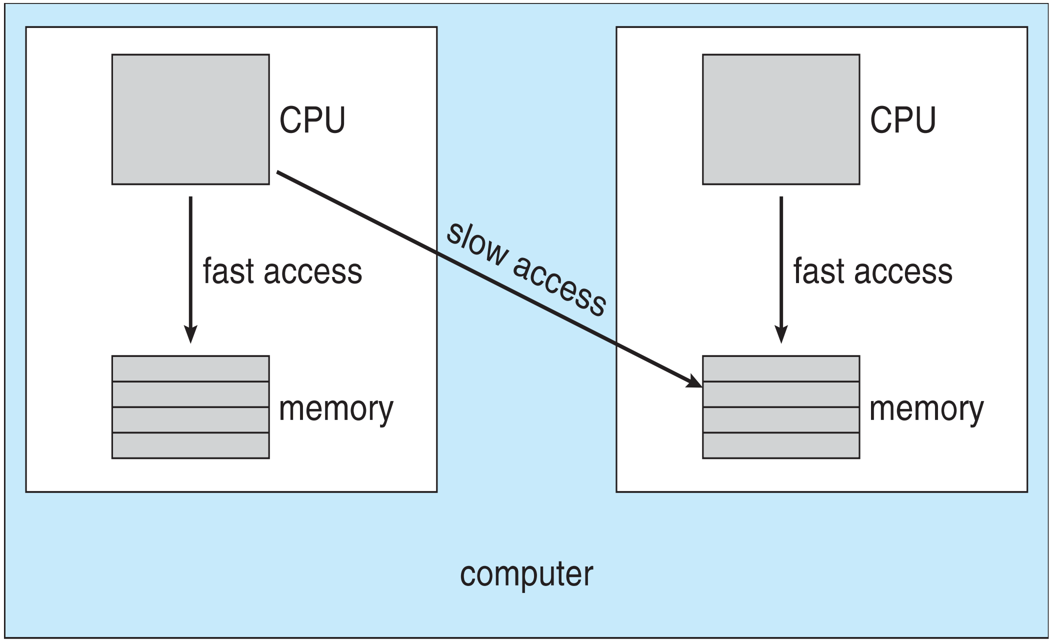


- **Non-Uniform Memory Access (NUMA)**
  - Often made by physically linking two or more SMPs
  - One SMP can directly access memory of another SMP
  - Memory access across link is slower
  - Example: IBM Blade server



NUMA and CPU Scheduling

- Occurs in systems containing combined CPU and memory boards
- CPU scheduler and memory-placement works together



Load Balancing

- Keep the workload evenly distributed across all processors
    - Only necessary on systems where each processor has its own private queue of eligible processes to execute
- **Two strategies**
    - **Push migration**
        - Move (push) processes from overloaded to idle or less-busy processor
    - **Pull migration**
        - Idle processor pulls a waiting task from a busy processor
    - Often implemented in parallel
- Load balancing often counteracts the benefits of processor affinity

Real-time Scheduling

- Real-time does not mean speed, but **keeping deadlines**

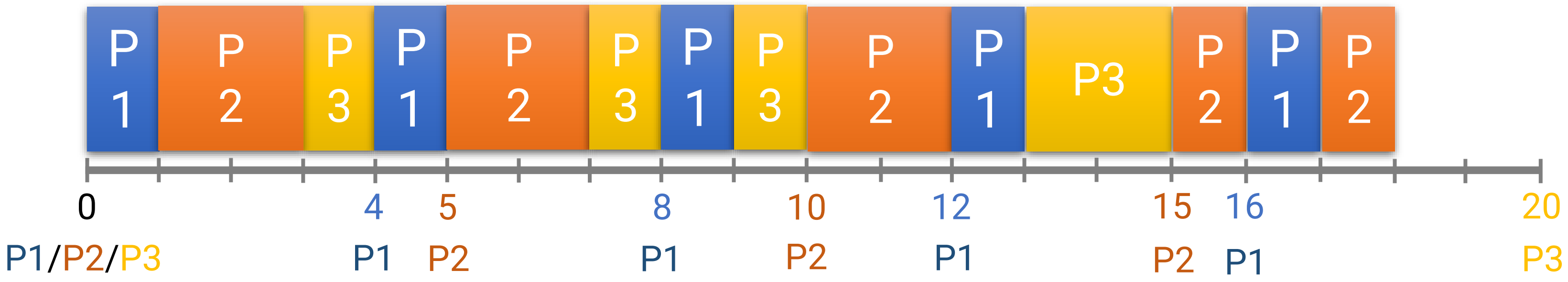
- **Soft real-time requirements**
  - Missing the deadline is unwanted, but is not immediately critical
  - Example: multimedia streaming
- **Hard real-time requirements**
  - Missing the deadline results in a fundamental failure
  - Example: nuclear power plant controller

Real-time Scheduling Algorithms

- Must support **preemptive**, **priority-based** scheduling
  - But only guarantees **soft real-time**
- **Description**
  - T1 = (0, 4, 10) == (Ready, Execution, Period)
  - T2 = (1, 2, 4)
- **Rate-Monotonic (RM) algorithm**
  - **Shorter period → high priority**
  - Fixed-priority real-time system scheduling algorithm
- **Earliest-deadline-first (EDF) algorithm**
  - **Earlier deadline → higher priority**
  - Dynamic priority algorithm

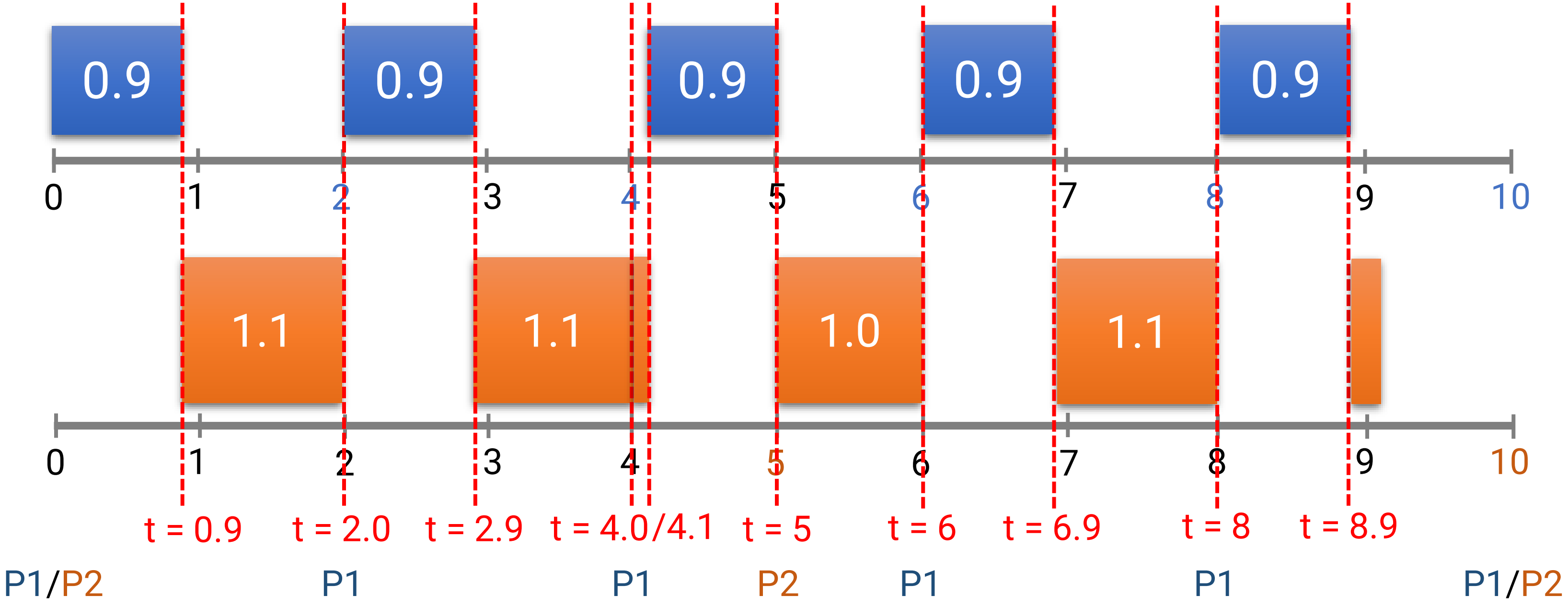
Rate-Monotonic (RM) Scheduler

- **Fixed-priority scheduling**
    - All jobs of the same task have same priority
    - The task’s priority is fixed
- **The shorter period, the higher priority**
- Example (Preempted)
    - T1 = (4, 1), T2 = (5, 2), T3 = (20, 5) for (period, execution)
        - Period: 4 < 5 < 20
        - Priority: T1 > T2 > T3



Early Deadline First (EDF) Scheduler

- **Dynamic-priority scheduler**
  - Task's priority is not fixed
  - Task's priority is determined by deadline
- Example (preempted)
  - T1 = (2, 0.9), T2 = (5, 2.3) for (period, execution)



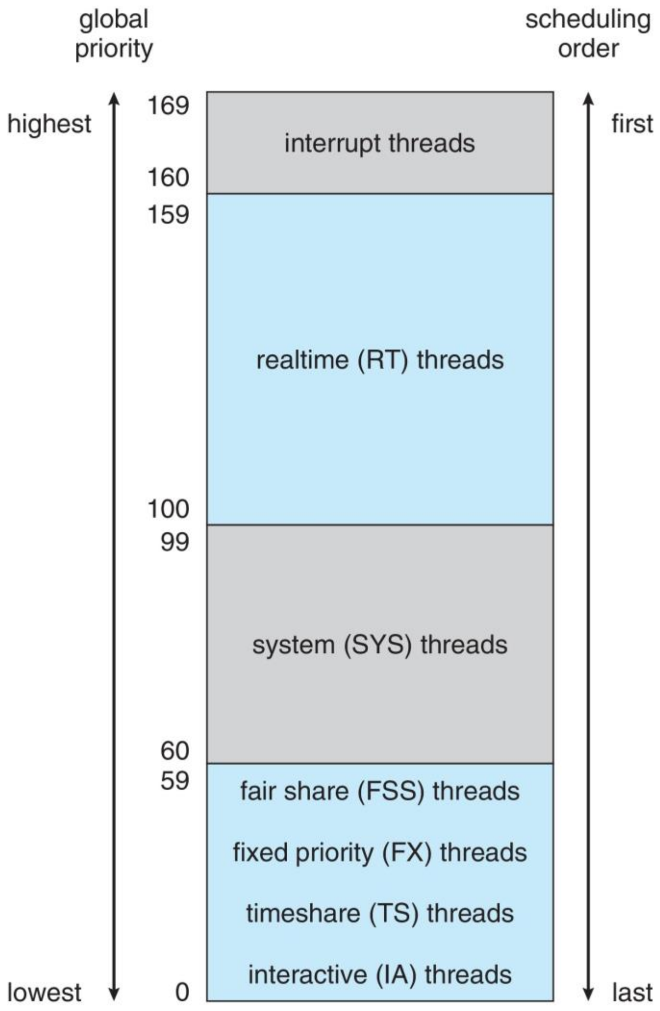
Scheduling Case Study

Operating System Examples

- Solaris
- Windows
- Linux

Solaris Scheduler

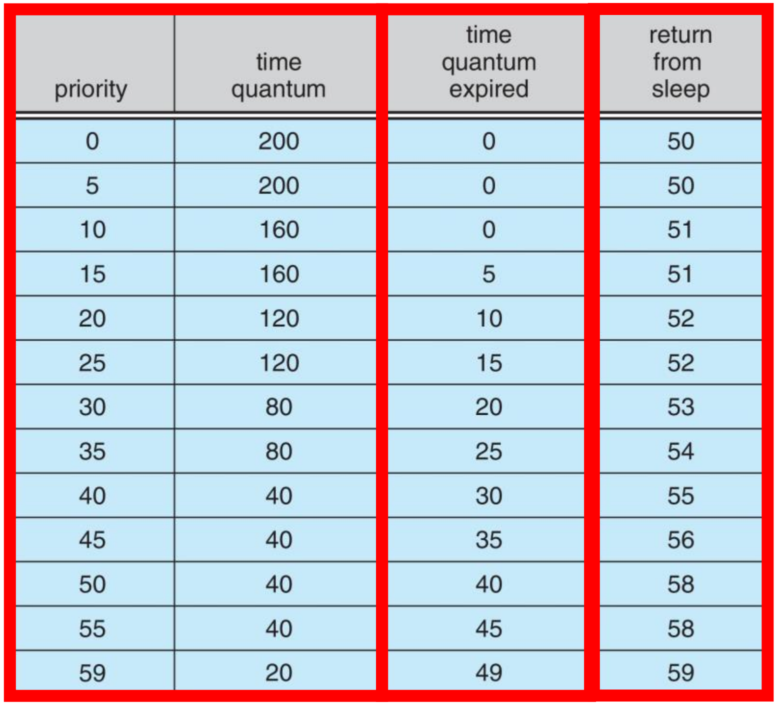
- **Priority-based** multi-level feedback queue scheduling
- Six classes of scheduling
  - Real-time
  - System
  - Time sharing
  - Interactive
  - Fair share
  - Fixed priority
- Each class has its **own priorities** and **scheduling algorithm**
- The scheduler converts the class specific priorities into global priorities



Solaris Scheduler (cont.)

- For time sharing and interactive processes
  - Inverse relationship between priorities and time slices: **the higher the priority, the smaller the time slice**

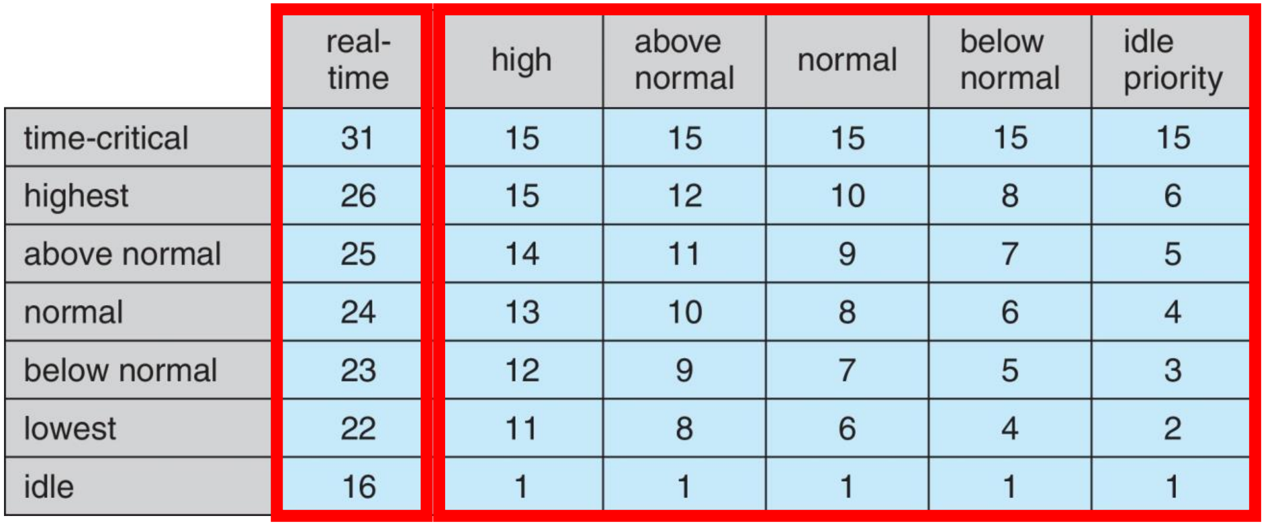
- **Time quantum expired**
  - The new priority of a thread that has used its entire time quantum without blocking
- **Return from sleep**
  - The new priority of a thread that is returning from sleeping (I/O wait)



priority	time quantum	time quantum expired	return from sleep
0	200	0	50
5	200	0	50
10	160	0	51
15	160	5	51
20	120	10	52
25	120	15	52
30	80	20	53
35	80	25	54
40	40	30	55
45	40	35	56
50	40	40	58
55	40	45	58
59	20	49	59

Windows XP Scheduler

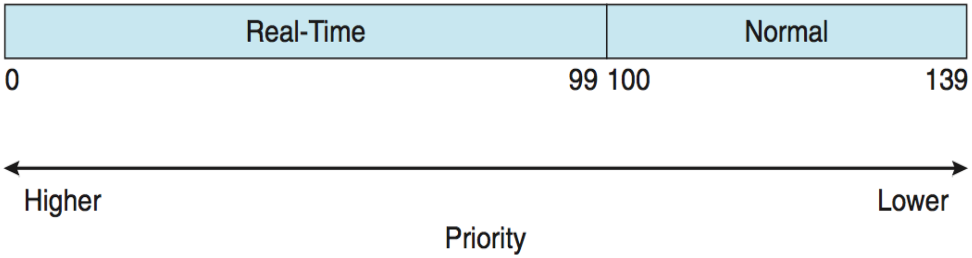
- Similar to Solaris: Multi-level feedback queue
- Scheduling
  - From the highest priority queue to lowest priority queue (0 ~ 31)
    - The highest-priority thread always run
    - Round-robin in each priority queue
  - Priority changes dynamically except for real-time class



	real-time	high	above normal	normal	below normal	idle priority
time-critical	31	15	15	15	15	15
highest	26	15	12	10	8	6
above normal	25	14	11	9	7	5
normal	24	13	10	8	6	4
below normal	23	12	9	7	5	3
lowest	22	11	8	6	4	2
idle	16	1	1	1	1	1

Linux Scheduler

- Preemptive priority based scheduling
  - But allows only user mode processes to be preempted
  - Two separate process priority ranges
  - **Lower values indicate higher priorities**
  - Higher priority with longer time quantum
- **Real-time tasks (0 ~ 99)**
  - Static priorities
- **Other tasks (100 ~ 140)**
  - Dynamic priorities based on task interactivity



Objectives Review

- Describe various CPU scheduling algorithms
- Explain the issues related to multiprocessor and multicore scheduling
- Describe various real-time scheduling algorithms
- Describe the scheduling algorithms used in the Windows, Linux, and Solaris operating systems