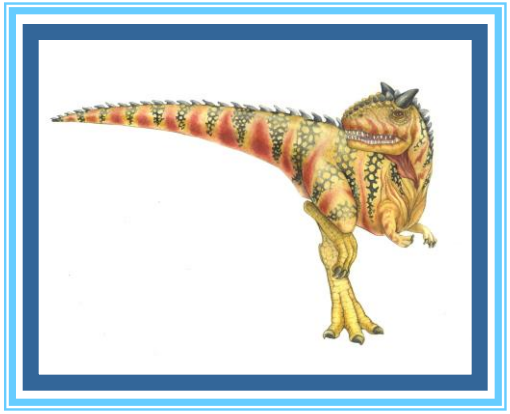
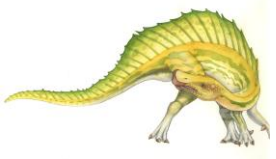
Chapter 6: CPU Scheduling



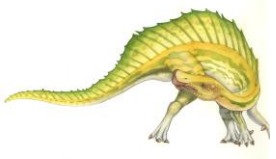
Chapter 6: CPU Scheduling

- Basic Concepts
- Scheduling Criteria
- Scheduling Algorithms
- Thread Scheduling
- Multiple-Processor Scheduling
- Operating Systems Examples
- Algorithm Evaluation



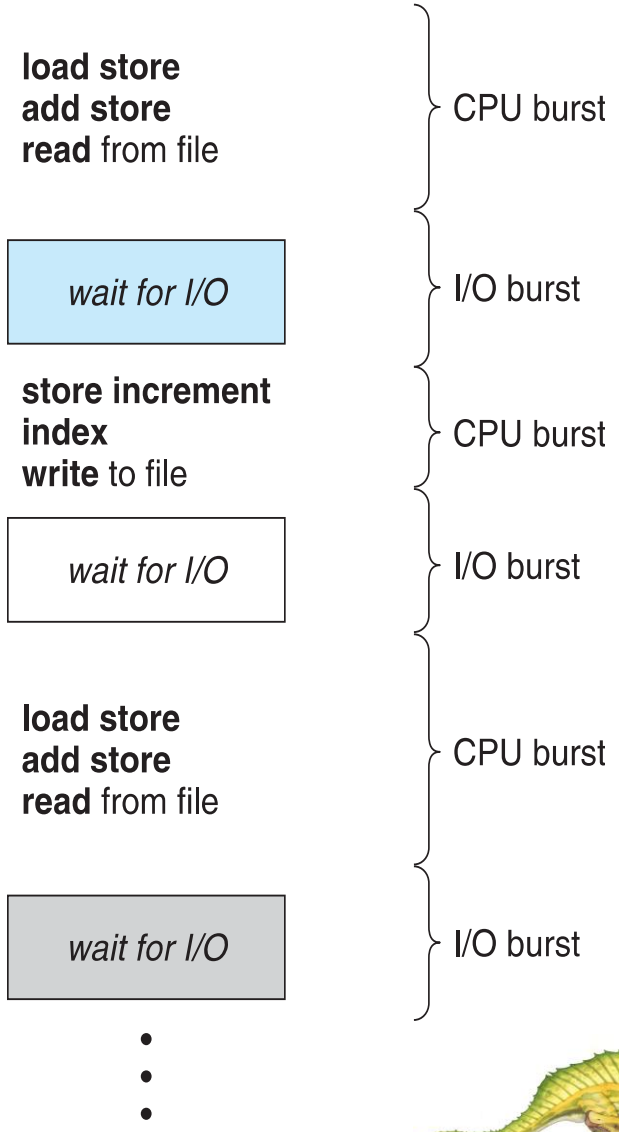
Objectives

- To describe various **CPU-scheduling algorithms**
- To discuss **evaluation criteria** for selecting a CPU-scheduling algorithm for a particular system
- To examine the scheduling algorithms of several operating systems



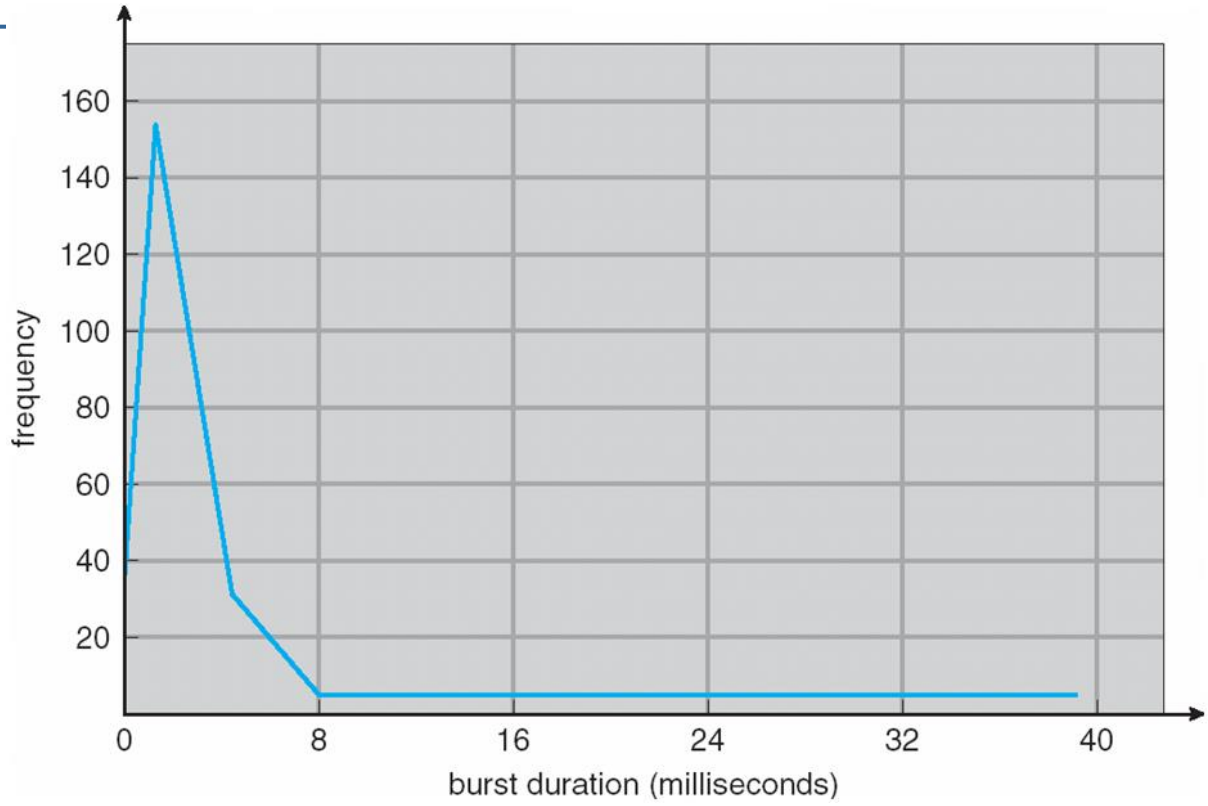
6.1 Basic Concepts

- Threads are scheduled not processes
- 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

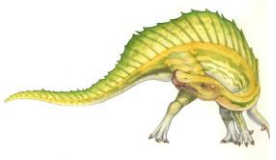
- The curve is generally characterized as exponential or hyperexponential, with a large number of short CPU bursts and a small number of long CPU bursts
- An **I/O-bound** program typically has many **short CPU bursts**
- A CPU-bound program might have a few long CPU bursts



This distribution can be important in the selection of an appropriate CPU-scheduling algorithm

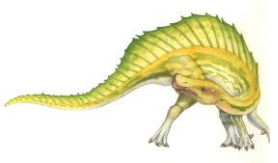
CPU Scheduler (Short Term)

- **Short-term scheduler** selects from among the processes in ready queue, and allocates the CPU to one of them. Queue may be ordered in various ways
- CPU scheduling decisions may take place when a process:
  1. Switches from running to waiting state
  2. Switches from running to ready state
  3. Switches from waiting to ready
  4. Terminates
- Scheduling under 1 **and** 4 is **nonpreemptive**
  - When scheduling takes place only under circumstances 1 and 4, we say that the scheduling scheme is **nonpreemptive** or **cooperative**.
  - Once the CPU has been allocated to a process, the process keeps the CPU until it releases the CPU **either** by **terminating** or by **switching** to the **waiting state**
- All other scheduling is **preemptive (Problem : race conditions)**
  - Consider access to shared data
  - Consider preemption while in kernel mode
  - Consider interrupts occurring during crucial OS activities



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
  - jumping to the proper location in the user program to restart that program
- It is **invoked** during **every process switch**
- **Dispatch latency** – time it takes for the dispatcher to stop one process and start another running

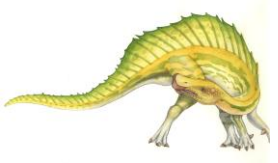


6.2 Scheduling Criteria

- **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 . Turnaround time is the sum of the periods spent waiting to get into memory, waiting in the ready queue, executing on the CPU, and doing I/O.
- **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 the time it takes output to response (for time-sharing environment)

Scheduling Algorithm Optimization Criteria

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



6.3 Scheduling Algorithms

- First-Come, First-Served Scheduling
- Shortest-Job-First Scheduling
- Priority Scheduling
- Round-Robin Scheduling
- Multilevel Queue Scheduling
- Multilevel Feedback Queue Scheduling

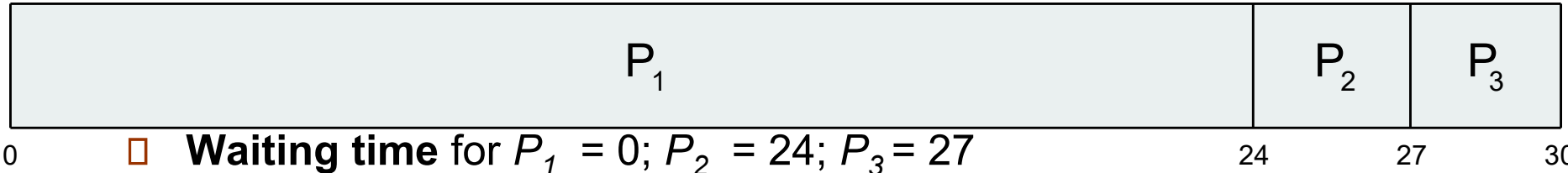
First- Come, First-Served (FCFS) Scheduling

- When a process enters the ready queue, its PCB is linked onto the tail of the queue
- When the CPU is free, it is allocated to the process at the head of the queue

Process	Burst Time
P_1	24
P_2	3
P_3	3

FCFS scheduling algorithm is **nonpreemptive**

- 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

The average waiting time under the FCFS policy is often quite long

- **Convoy effect** - short process behind long process
  - Consider one CPU-bound and many I/O-bound processes

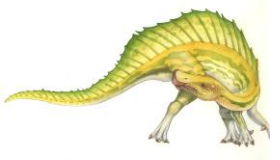
FCFS Scheduling (Cont.)

Suppose that the processes arrive in the order:

$$P_2 , P_3 , P_1$$

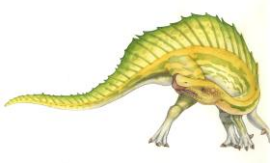
- The Gantt chart for the schedule is:

- 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



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



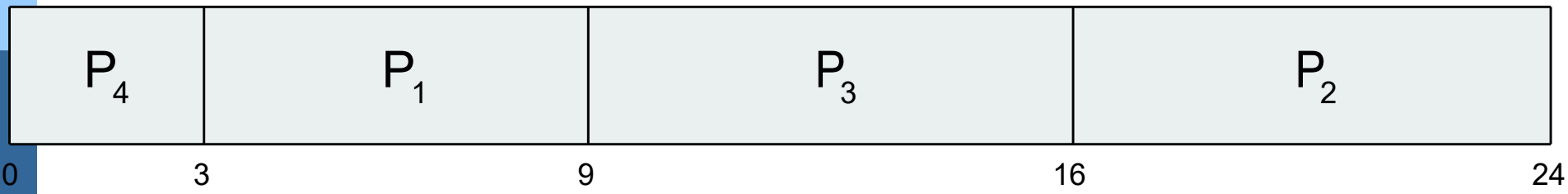
Example of SJF

Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3

The SJF algorithm can be either preemptive or nonpreemptive.

- SJF scheduling chart

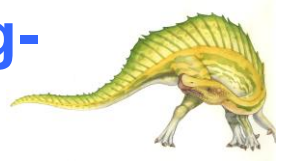
Nonpreemptive SJF



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

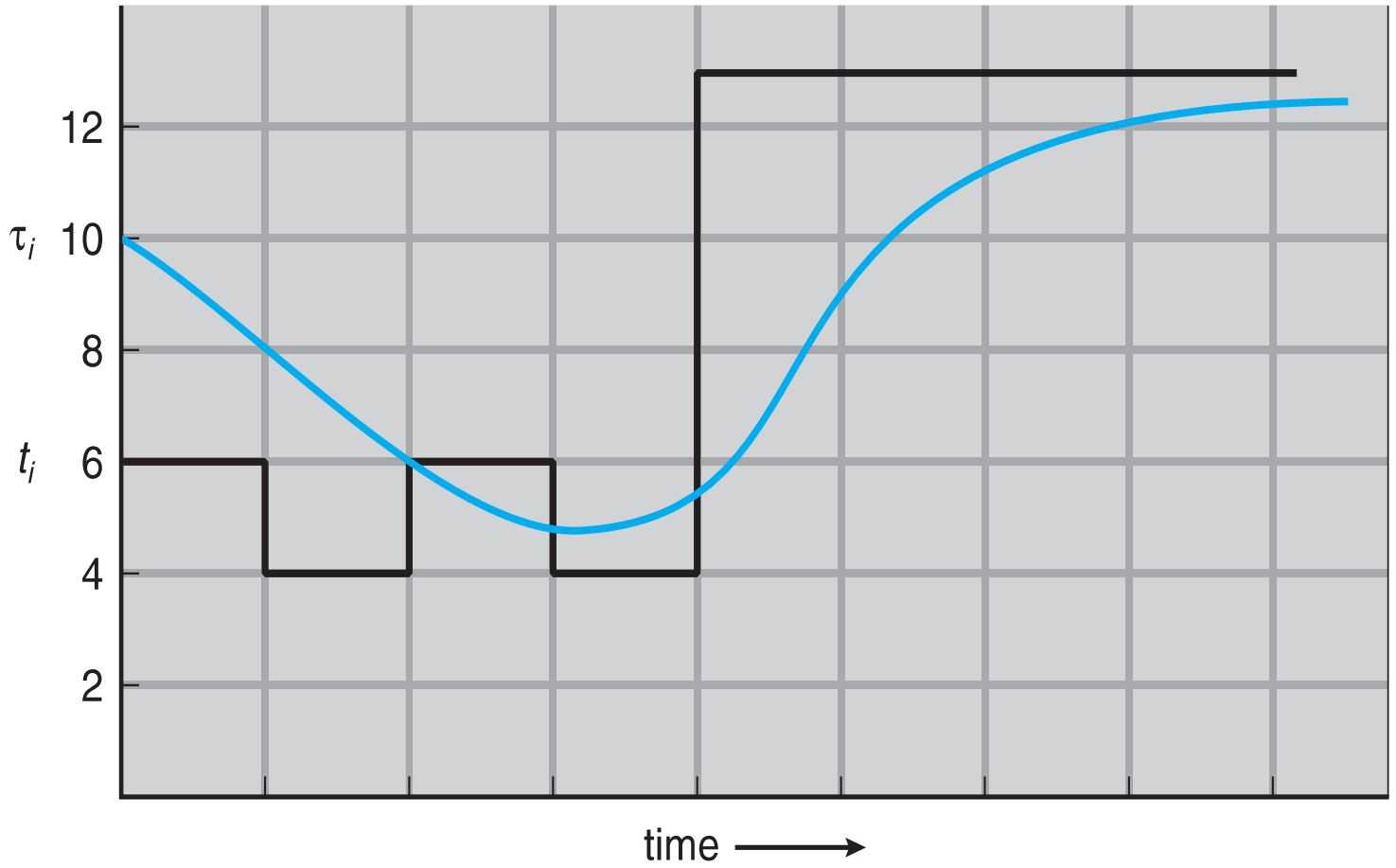
Determining Length of Next CPU Burst

- 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. $\alpha, 0 \le \alpha \le 1$
  4. Define : $\tau_{n+1} = \alpha\, t_n + (1-\alpha)\tau_n$
- Commonly, α set to ½
- Preemptive version called **shortest-remaining-time-first**

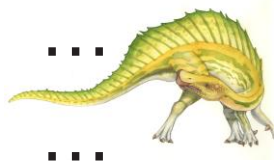


Prediction of the Length of the Next CPU Burst

$$\tau_{n+1} = \alpha\, t_n + (1-\alpha)\tau_n$$



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



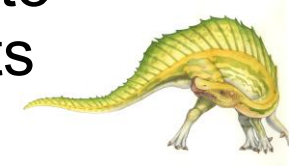
Examples of Exponential Averaging

$$\tau_{n+1} = \alpha\, t_n + (1-\alpha)\tau_n$$

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

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

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

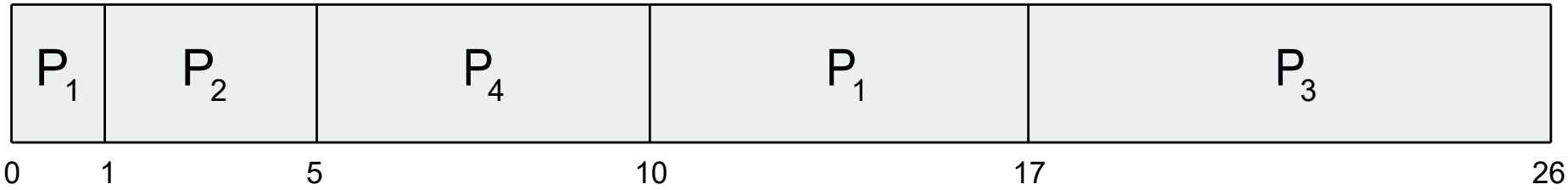


Example of Shortest-remaining-time-first

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

Process	*Arrival* Time	Burst Time
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

What about nonpreemtive SJF?

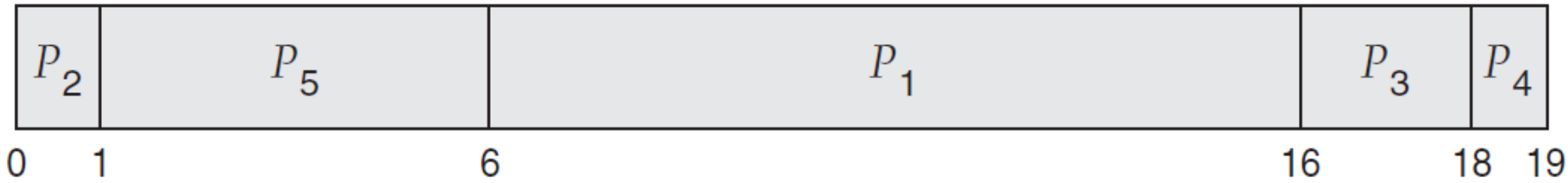
Priority Scheduling

- A priority number (integer) is associated with each process
- The CPU is allocated to the process with the highest priority (smallest integer ≡ highest priority)
  - Preemptive
  - Nonpreemptive
- SJF is priority scheduling where priority is the inverse of predicted next CPU burst time
- Problem ≡ **Starvation** – low priority processes may never execute (MIT 1973 case)
  - *When they shut down the IBM 7094 at MIT in 1973, they found a low-priority process that had been submitted in 1967 and had not yet been run.*
- Solution ≡ **Aging** – as time progresses increase the priority of the process

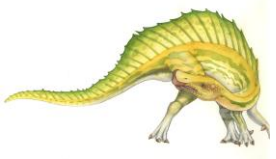
Example of Priority Scheduling

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

- Priority scheduling Gantt Chart

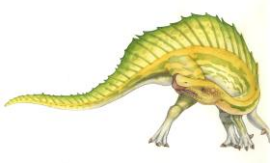


- Average waiting time = 8.2 msec



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



Round Robin (RR)

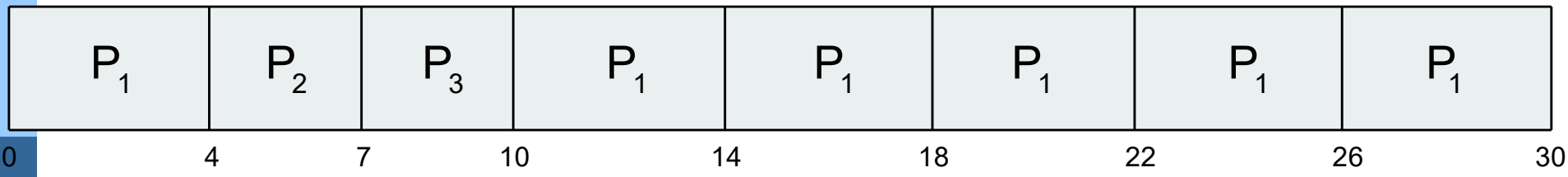
- Designed especially for time sharing systems
- It is similar to FCFS scheduling, but **preemption** is added to enable the system to switch between processes
- The ready queue is treated as a circular queue
  - New processes are added to the tail of the ready queue.
  - The CPU scheduler picks the first process from the ready queue, sets a timer to interrupt after 1 time quantum, and dispatches the process.
  - ....
  - After the context switch, the process will be put at the tail of the ready queue.

The RR scheduling algorithm is preemptive

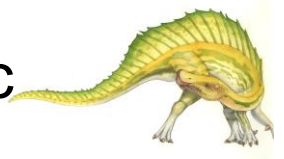
Example of RR with Time Quantum = 4

Process	Burst Time
P_1	24
P_2	3
P_3	3

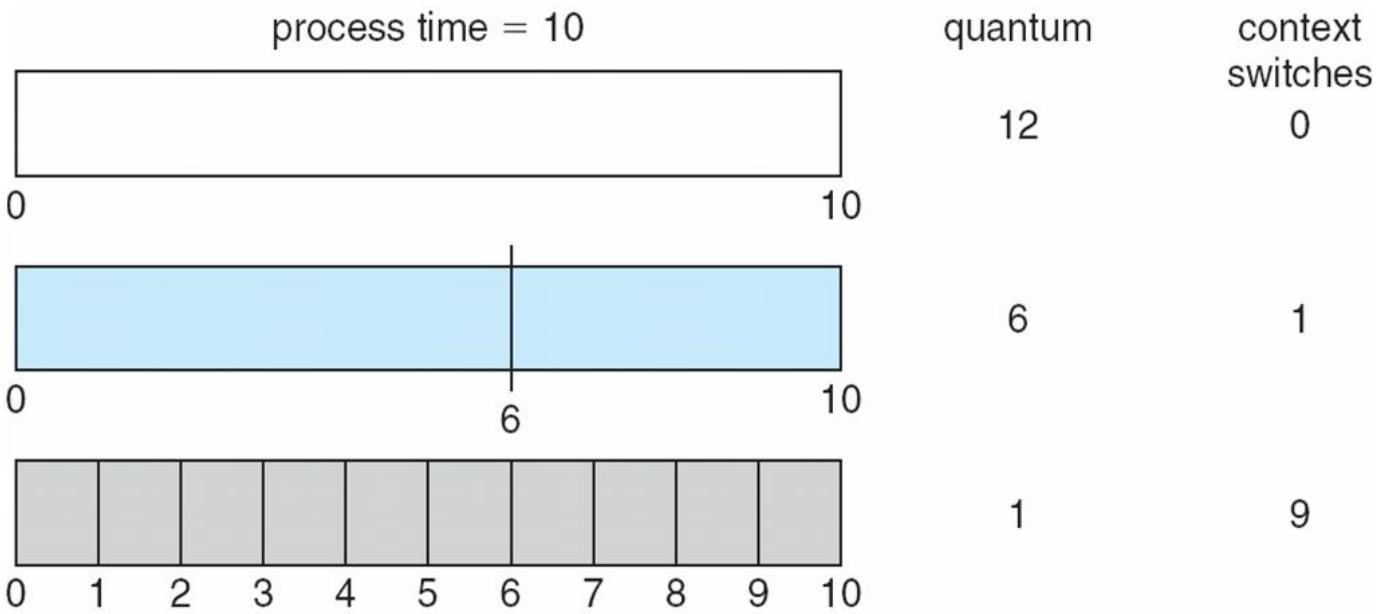
- The Gantt chart is:



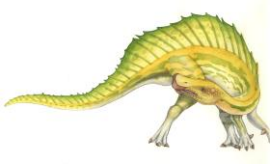
- Typically, higher average turnaround than SJF, but better ***response***
- q should be large compared to context switch time
- q usually 10ms to 100ms, context switch < 10 usec



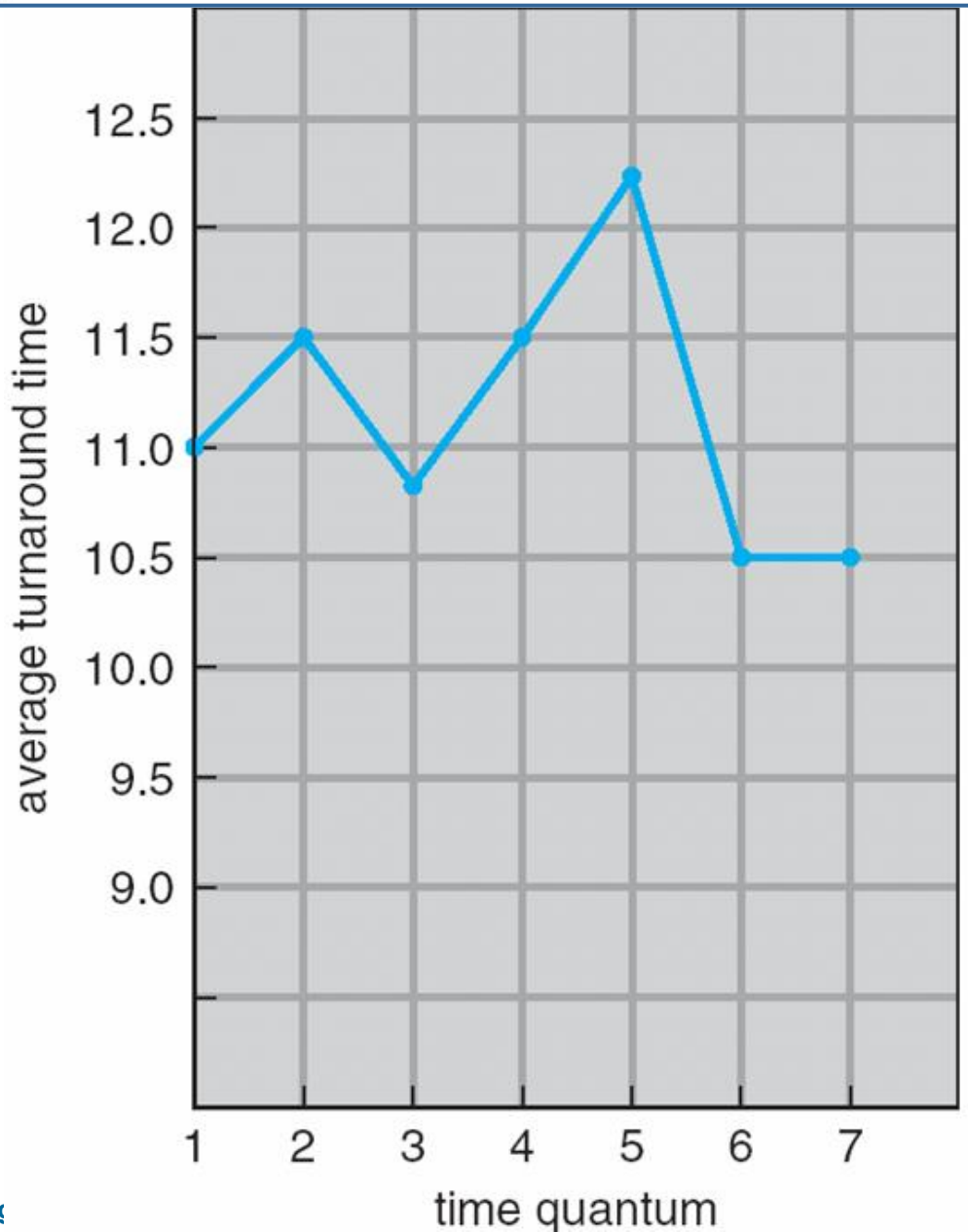
Time Quantum and Context Switch Time



Turnaround times for 3 process with 10 units quanta=1 vs quanta=10



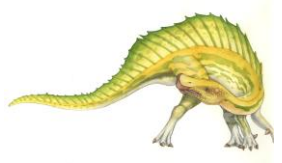
Turnaround Time Varies With The Time Quantum



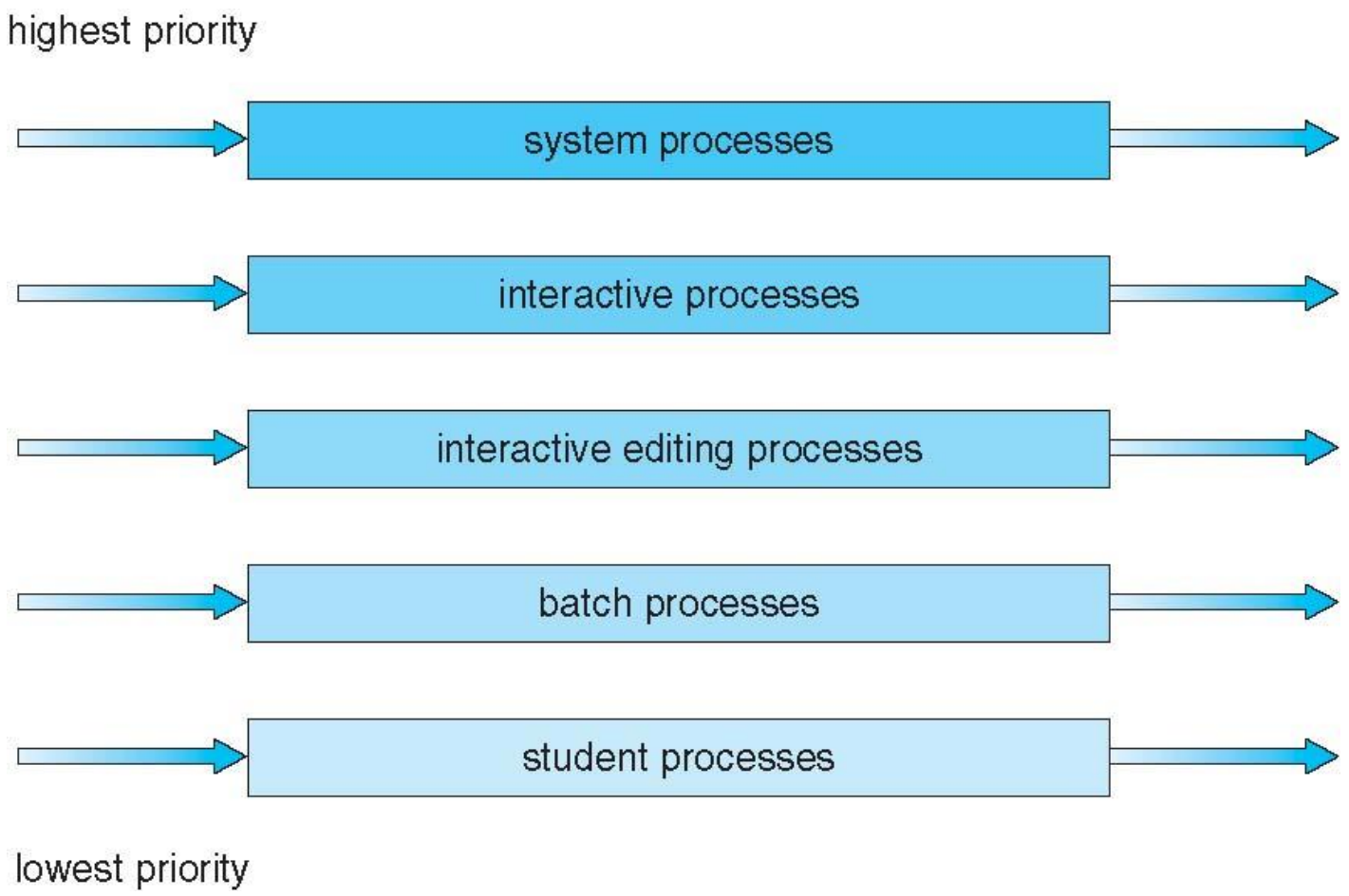
process	time
P_1	6
P_2	3
P_3	1
P_4	7

Multilevel Queue

- Ready queue is partitioned into separate queues, eg:
  - **foreground** (interactive)
  - **background** (batch)
- Process permanently in a given queue
- **Each queue has its own scheduling algorithm:**
  - foreground – RR
  - background – FCFS
- **Scheduling must be done between the queues:**
  - **Fixed priority scheduling**; (i.e., serve all from foreground then from background). **Possibility of starvation**.
  - **Time slice** – each queue gets a certain amount of CPU time which it can schedule amongst its processes; i.e.,
    - 80% to foreground in RR
    - 20% to background in FCFS



Multilevel Queue Scheduling

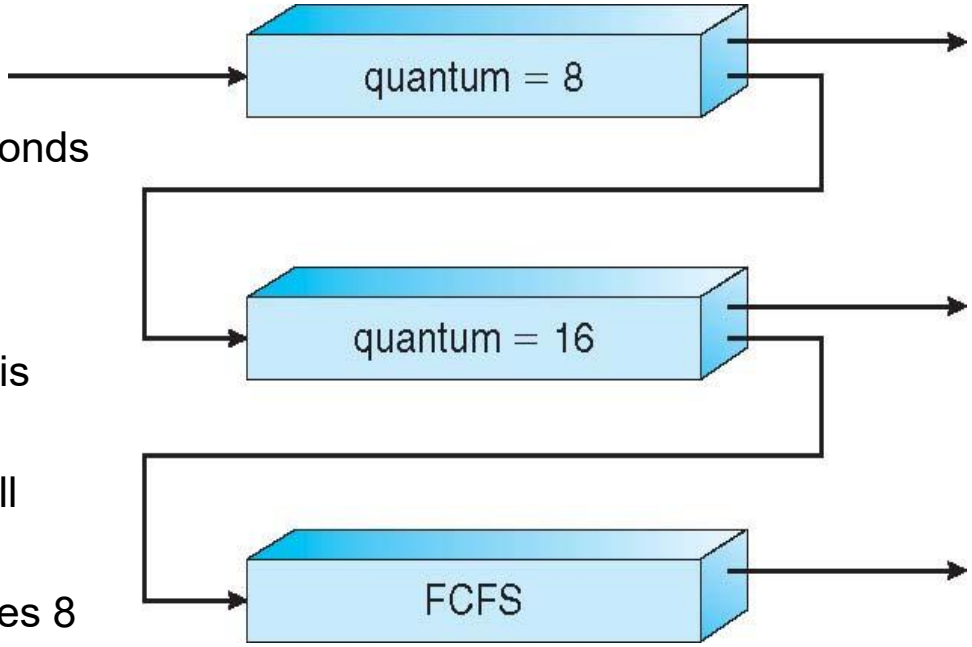


Multilevel Feedback Queue

- 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

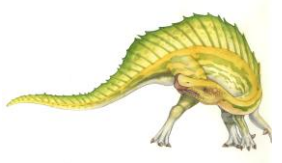
- Three queues:
  - Q_0 – RR with time quantum 8 milliseconds
  - Q_1 – RR time quantum 16 milliseconds
  - Q_2 – FCFS
- Scheduling
  - A new job enters queue Q_0 which is served RR
    - The scheduler first executes all processes in queue 0
    - When it gains CPU, job receives 8 milliseconds
    - If it does not finish in 8 milliseconds, job is moved to queue Q_1
  - At Q_1 job is again served RR and receives 16 additional milliseconds
    - If it still does not complete, it is preempted and moved to queue Q_2



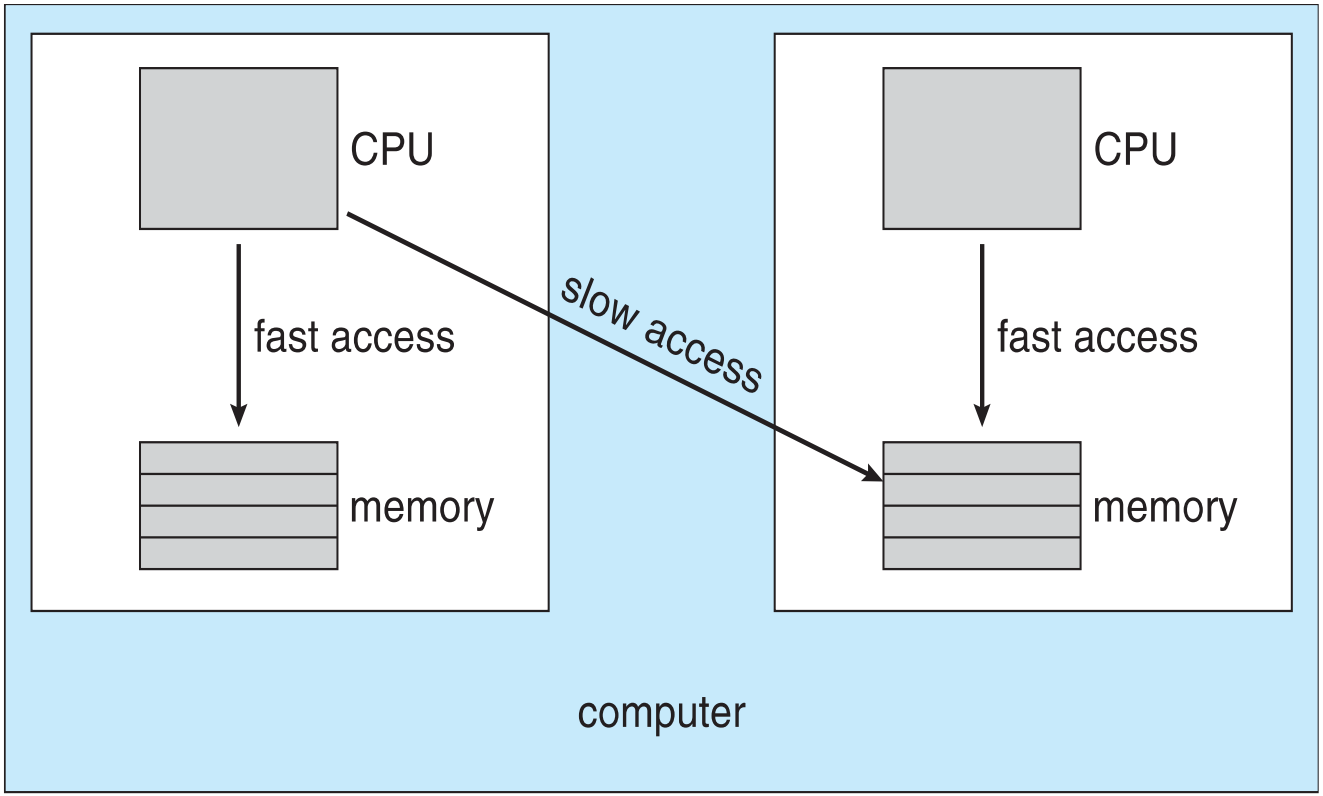
Process moves between the queues

Multiple-Processor Scheduling

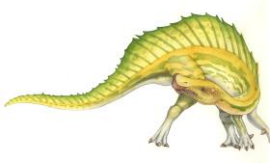
- CPU scheduling more complex when multiple CPUs are available
- **Homogeneous processors** within a multiprocessor
- **Asymmetric multiprocessing** – only one processor accesses the system data structures, alleviating the need for data sharing
- **Symmetric multiprocessing (SMP)** – each processor is **self-scheduling**, all processes in common ready queue, or each has its own private queue of ready processes
  - **Currently, most common**
- **Processor affinity** – process has affinity for processor on which it is currently running (invalidate and repopulate)
  - **soft affinity:** attempting to keep a process running on the same processor **without any guarantee**
  - **hard affinity :** allowing a process to specify a subset of processors on which it may run



NUMA and CPU Scheduling

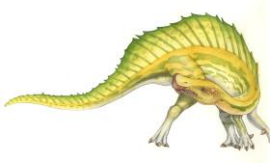


Note that memory-placement algorithms can also consider affinity



Multiple-Processor Scheduling – Load Balancing

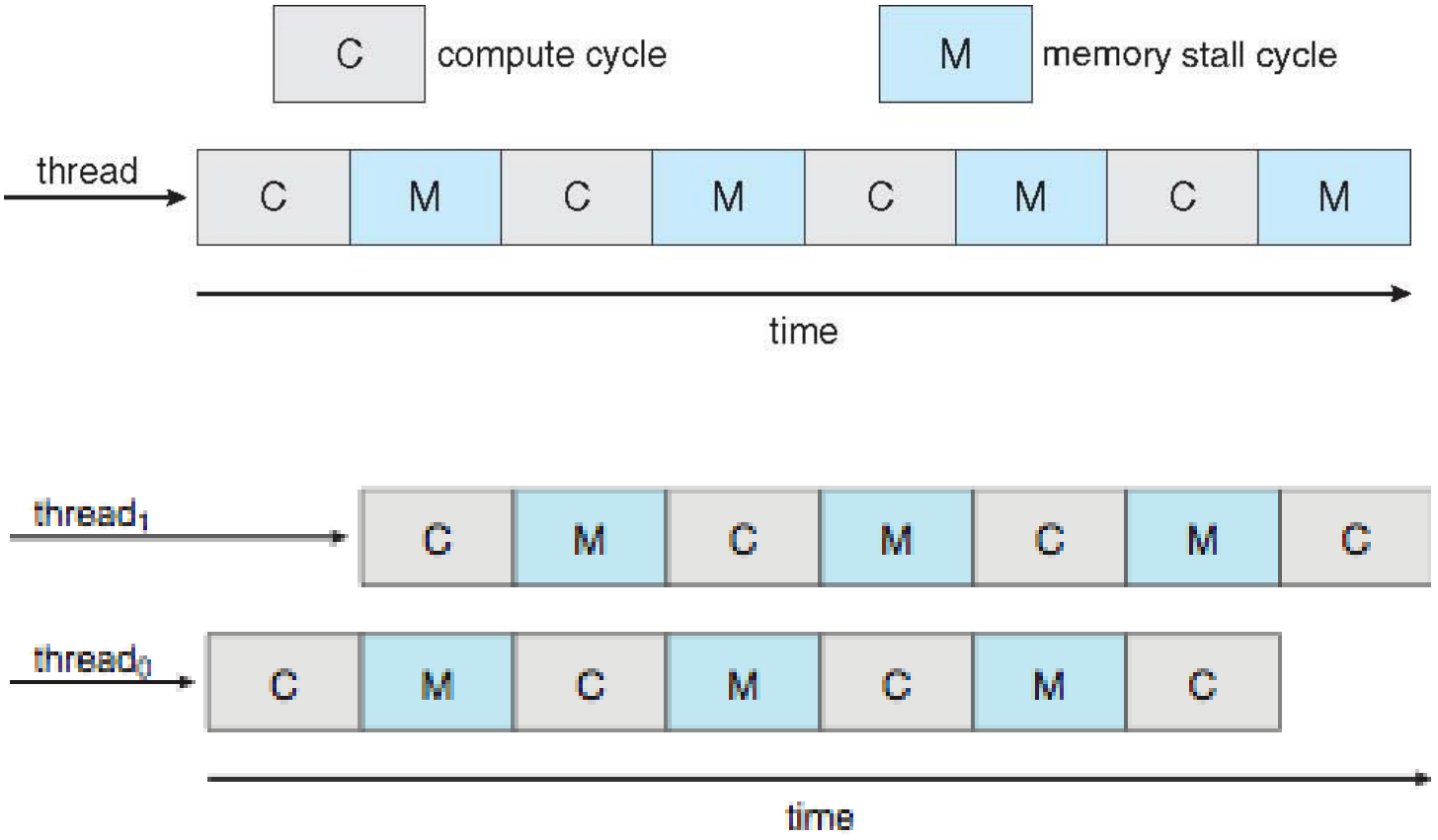
- In SMP, need to keep all CPUs loaded for efficiency
- **Load balancing** attempts to keep workload evenly distributed
  - **Push migration** – **periodic task** checks load on each processor, and if found **pushes task** **from overloaded CPU to other CPUs**
  - **Pull migration** – **idle processors** pulls waiting task from busy processor
- **no absolute rule concerning what policy is best**



Multicore Processors

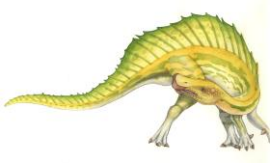
- Recent trend to place multiple processor cores on same physical chip
- Faster and consumes less power
- Multiple threads per core also growing
  - Takes advantage of **memory stall** to make progress on another thread while memory retrieve happens

Multithreaded Multicore System



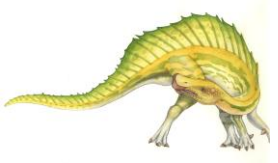
Operating System Examples

- Linux scheduling
- Windows scheduling



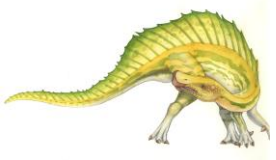
Linux Scheduling Through Version 2.5

- Prior to kernel version 2.5, ran variation of standard UNIX scheduling algorithm
- Version 2.5 moved to constant order *O*(1) scheduling time
  - **Preemptive**, **priority based**
  - Two priority ranges: time-sharing and real-time
  - **Real-time** range from 0 to 99 and **nice** value from 100 to 140
  - Higher priority gets larger q
  - Worked well, but poor response times for interactive processes



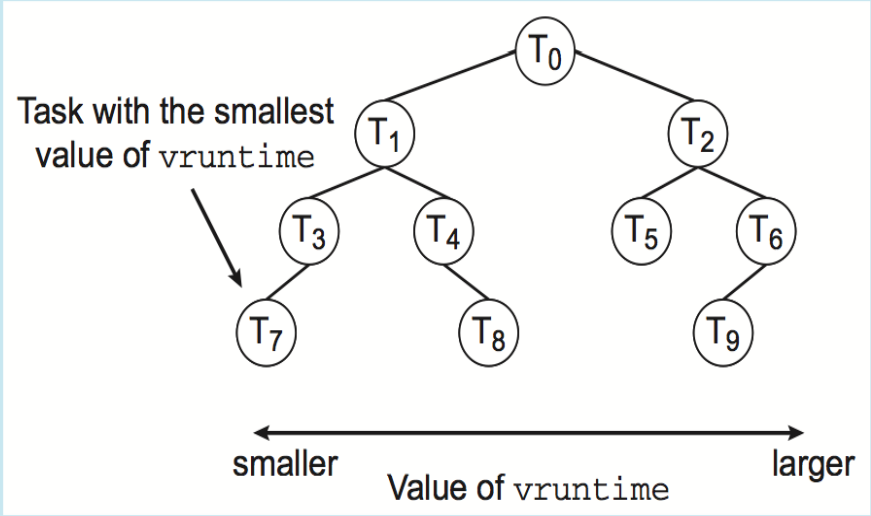
Linux Scheduling in Version 2.6.23 +

- ***Completely Fair Scheduler*** (CFS)
- **Scheduling classes**
  - Each has specific priority
  - Scheduler picks highest priority task in highest scheduling class
  - 2 scheduling classes included, others can be added
    1. default
    2. real-time
- Quantum calculated based on **nice value** from -20 to +19
  - Lower value is higher priority
- CFS scheduler maintains per task **virtual run time** in variable **`vruntime`**
- To decide next task to run, scheduler picks task with lowest virtual run time



CFS Performance

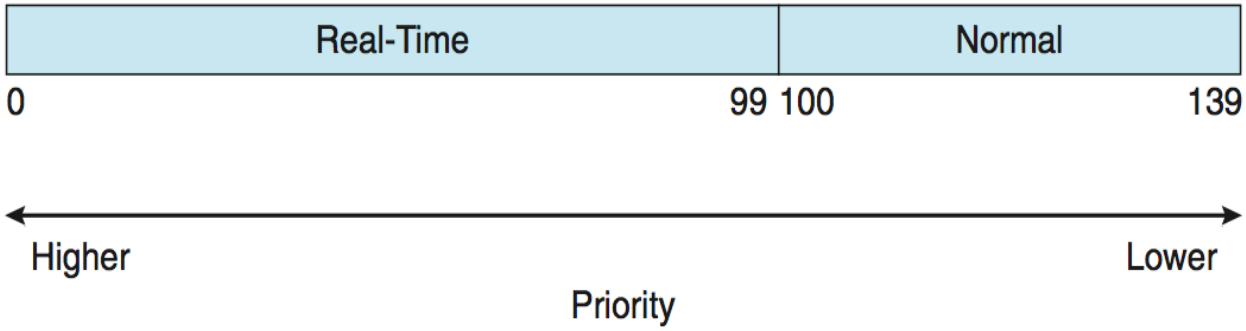
The Linux CFS scheduler provides an efficient algorithm for selecting which task to run next. Each runnable task is placed in a red-black tree—a balanced binary search tree whose key is based on the value of `vruntime`. This tree is shown below:

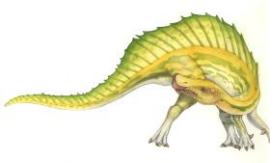


When a task becomes runnable, it is added to the tree. If a task on the tree is not runnable (for example, if it is blocked while waiting for I/O), it is removed. Generally speaking, tasks that have been given less processing time (smaller values of `vruntime`) are toward the left side of the tree, and tasks that have been given more processing time are on the right side. According to the properties of a binary search tree, the leftmost node has the smallest key value, which for the sake of the CFS scheduler means that it is the task with the highest priority. Because the red-black tree is balanced, navigating it to discover the leftmost node will require $O(\lg N)$ operations (where N is the number of nodes in the tree). However, for efficiency reasons, the Linux scheduler caches this value in the variable `rb_leftmost`, and thus determining which task to run next requires only retrieving the cached value.

Linux Scheduling (Cont.)

- Real-time scheduling according to POSIX.1b
  - Real-time tasks have static priorities
- Real-time plus normal map into global priority scheme
- Nice value of -20 maps to global priority 100
- Nice value of +19 maps to priority 139





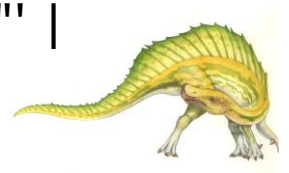
Windows Scheduling

- **Windows** uses **priority-based preemptive scheduling**
- Highest-priority thread runs next
- **Dispatcher** is scheduler
- Thread runs until (1) blocks, (2) uses time slice, (3) preempted by higher-priority thread
- Real-time threads can preempt non-real-time
- 32-level priority scheme
- **Variable class** is 1-15, **real-time class** is 16-31
- Priority 0 is memory-management thread
- Queue for each priority
- If no run-able thread, runs **idle thread**

Priority:
idle: 64
below normal: 16384
normal: 32
above normal: 32768
high priority: 128
real time: 24

```
wmic process where name="calc.exe" CALL setpriority 32768
```

```
Get-WmiObject Win32_process  -filter 'name="Calculator.exe"' |
foreach { $_.SetPriority(64)}
```



Algorithm Evaluation

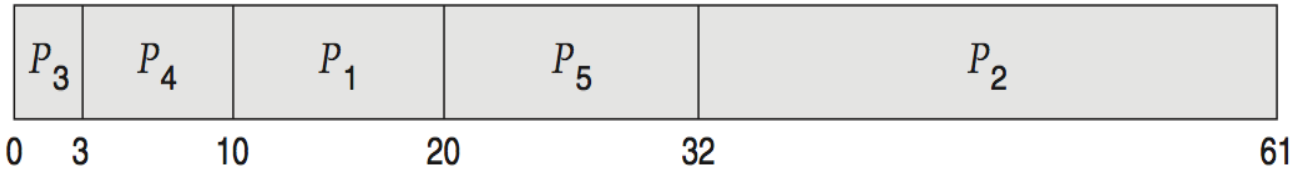
- How to select CPU-scheduling algorithm for an OS?
- Determine criteria, then evaluate algorithms
- **Deterministic modeling**
  - Type of **analytic evaluation**
  - Takes a particular predetermined workload and defines the performance of each algorithm for that workload
- Consider 5 processes arriving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

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

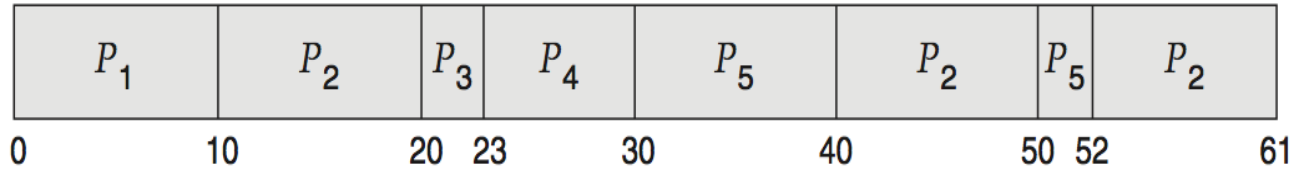
Deterministic Evaluation

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

  - Non-preemptive SJF is 13ms:



  - RR is 23ms:



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

End of Chapter 6