Process Scheduling

Gaurav Somani
LNM IIT, Jaipur
gaurav@lnmiit.ac.in

Ref: -[Galvin] [Tanenbaum][Stallings]

CPU Scheduling

- Basic Concepts
- Scheduling Criteria
- Scheduling Algorithms
- Multiple-Processor Scheduling
- Algorithm Evaluation

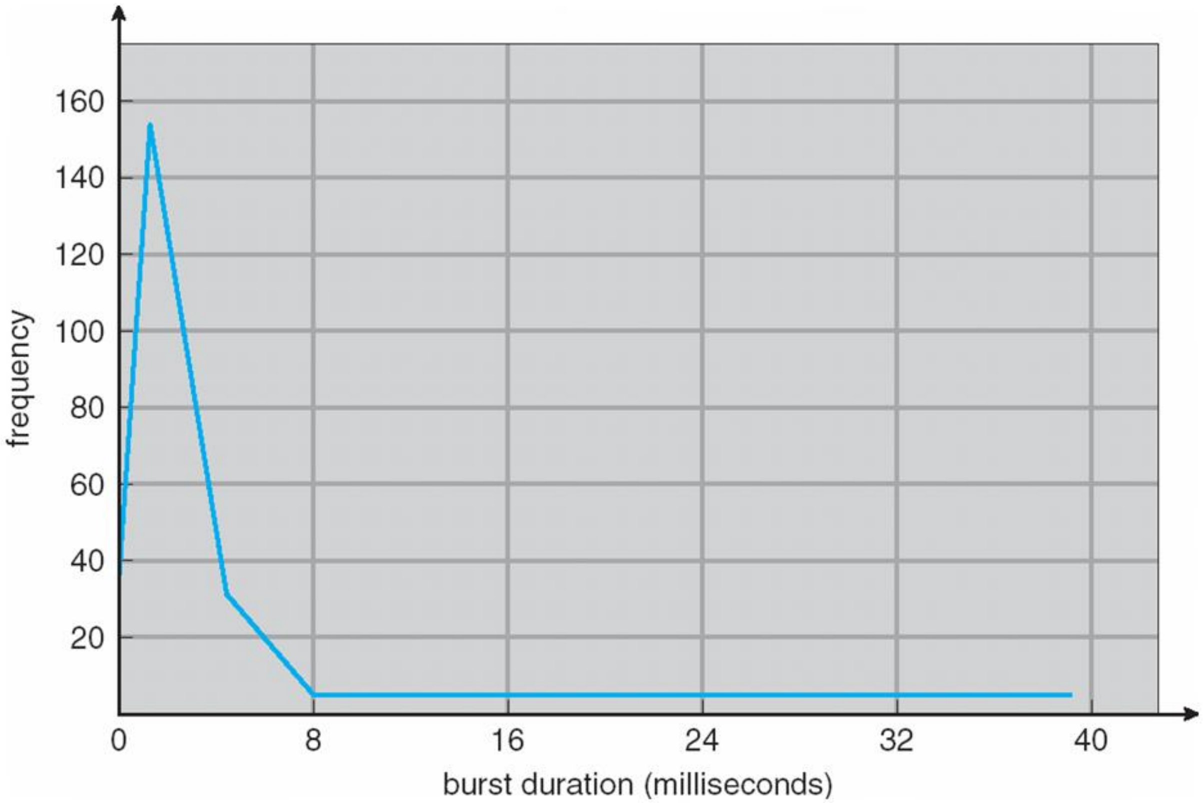
Objectives

- To introduce CPU scheduling, which is the basis for multiprogrammed operating systems
- To describe various CPU-scheduling algorithms
- To discuss evaluation criteria for selecting a CPU-scheduling algorithm for a particular system

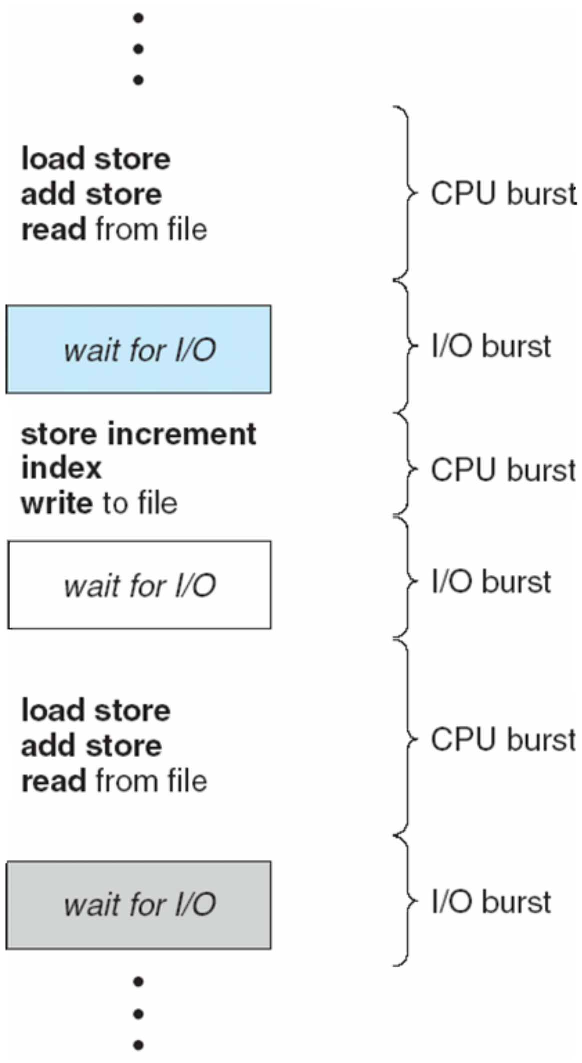
Basic Concepts

- 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** distribution

Histogram of CPU-burst Times



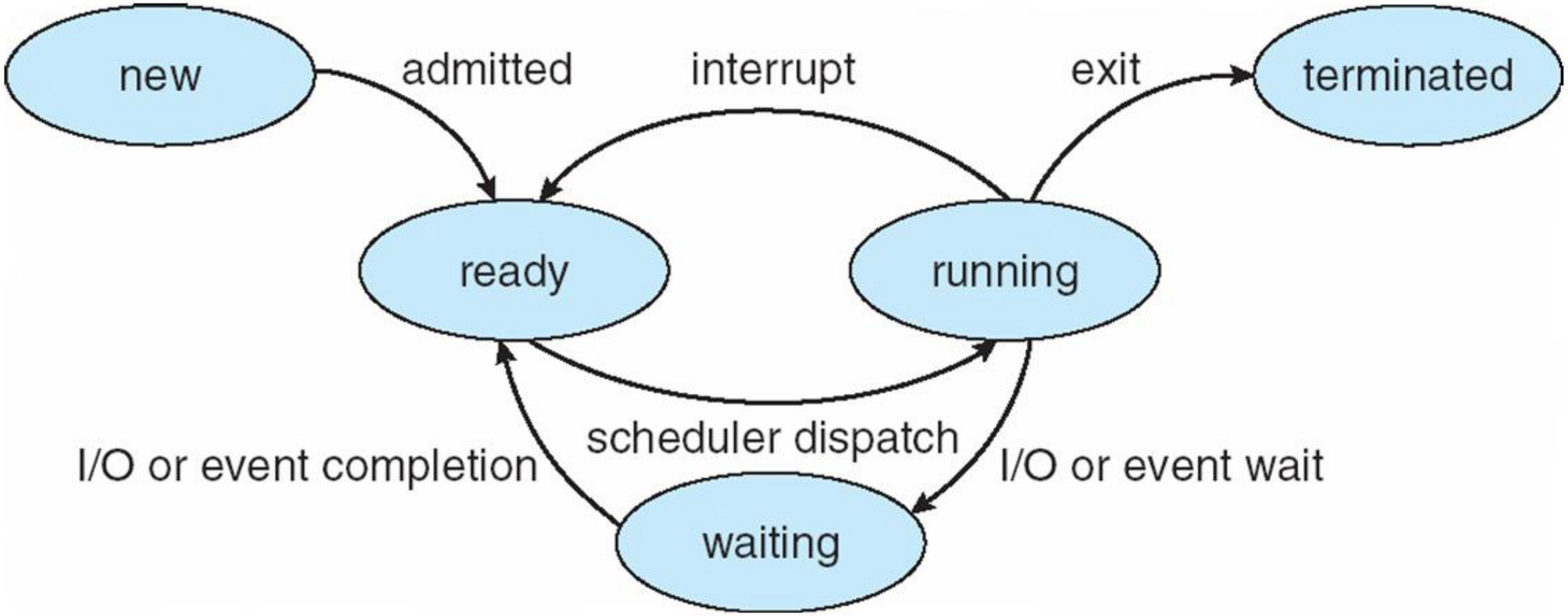
Alternating Sequence of CPU And I/O Bursts



Process State

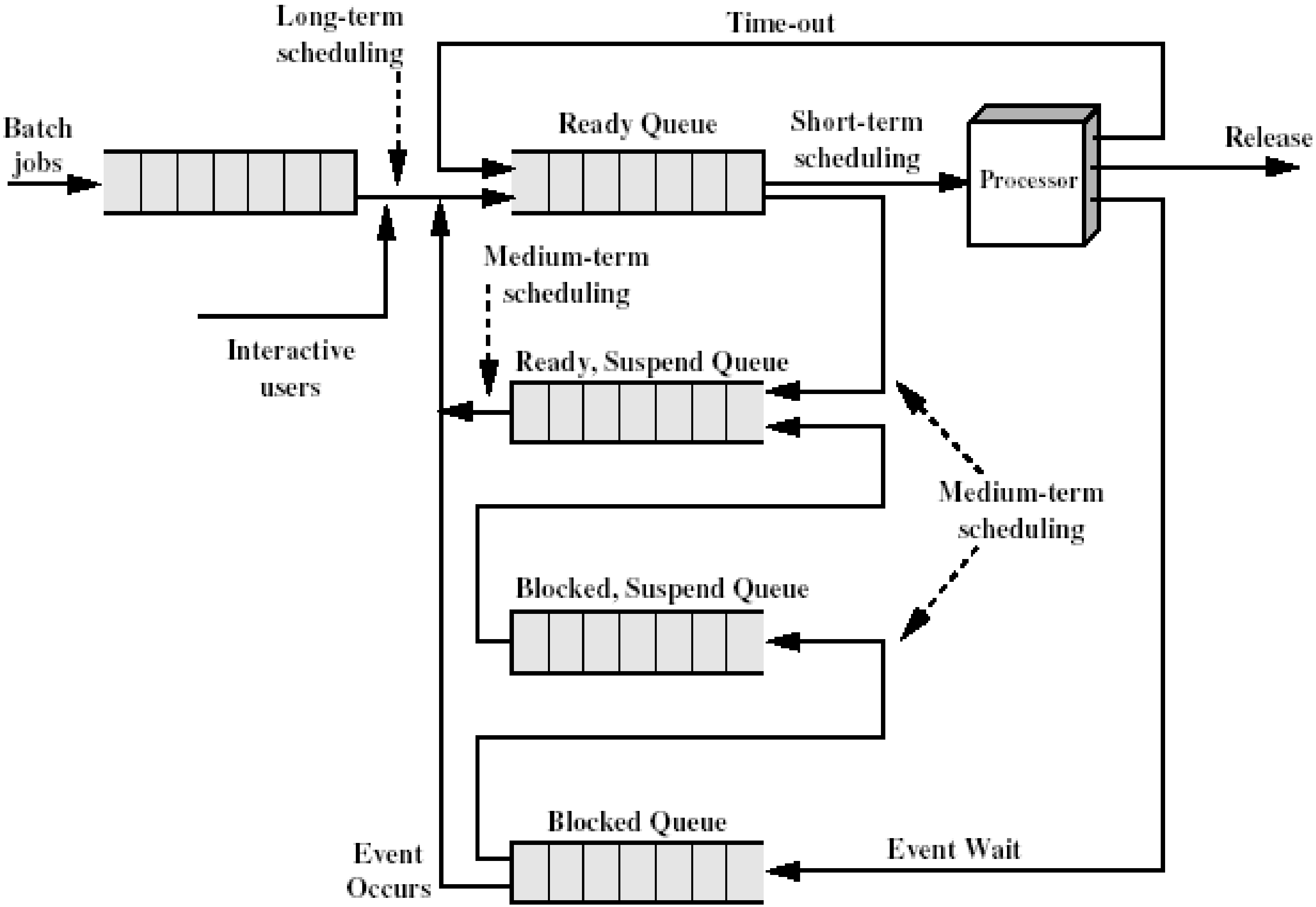
- As a process executes, it changes *state*
  - **new**: The process is being created
  - **running**: Instructions are being executed
  - **waiting**: The process is waiting for some event to occur
  - **ready**: The process is waiting to be assigned to a processor
  - **terminated**: The process has finished execution

Diagram of Process State



Scheduling Divisions

- *Long term scheduling*: which determines which programs are admitted to the system for execution and when, and which ones should be exited.
- *Medium term scheduling*: which determines when processes are to be suspended and resumed;
- *Short term scheduling* (or dispatching): which determines which of the ready processes can have CPU resources, and for how long.



Queung Diagram for Scheduling

Scheduling algorithms

- Preemptive – Non-preemptive
- Preemptive scheduling may provide for greater responsiveness to events, but at the expense of higher overhead caused by more frequent execution of the scheduler and associated process switching.

Scheduling algorithms

- Non- preemptive implies that the running process retains ownership of allocated resources, including the CPU, until the running process voluntarily surrenders the control to the OS. In other words, the running process cannot be forced to leave ownership of the CPU when a high-priority process becomes ready for execution.
- In case of preemptive scheduling the running process may be replaced by a higher-priority process at any time. Activating the scheduler whenever an event that changes the state of the system is detected does this.

CPU Scheduler

- Selects from among the processes in memory that are ready to execute, and allocates the CPU to one of them

The events are:

- Clock ticks (Timer interrupt)
- Interrupts and I/O completions
- Most operational OS calls
- Sending and receiving of signals
- Activation of interactive program

Whenever one of these events occurs, the OS invokes the short-term scheduler to determine whether another process should be scheduled for execution.

Dispatcher

- Dispatcher module gives control of the CPU to the process selected by the short-term scheduler; this involves:
  - switching context
  - jumping to the proper location in the user program to restart that program
- **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

Scheduling Algorithms Classifications

- Interactive Algorithms
  - RR
  - Priority
  - Lottery

- Batch Scheduling
  - FCFS
  - SJF
  - SRTN
  - HRRN

- Real-Time Scheduling Algorithms
  - EDF

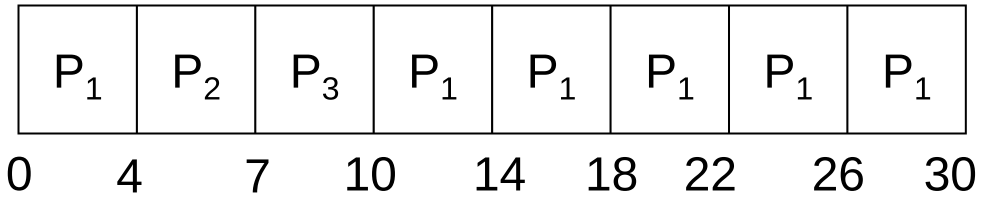
Round Robin (RR)

- Each process gets a small unit of CPU time (*time quantum*), 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.
- 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

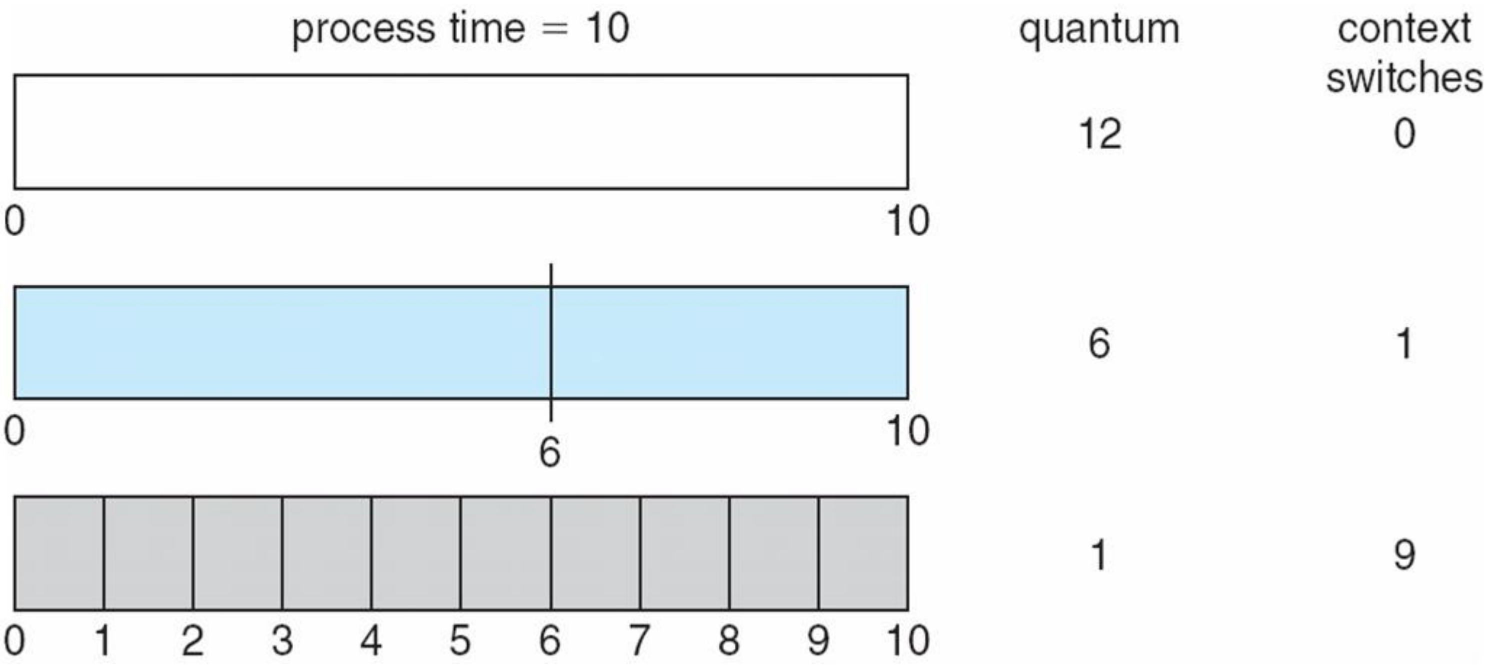
Process	Burst Time
P_1	24
P_2	3
P_3	3

- The Gantt chart is:



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

Time Quantum and Context Switch Time



process time = 10

quantum	context switches
12	0
6	1
1	9

Quantum?

1. **Shorter time quantum**
    A. means the scheduler is invoked more often, which means more scheduling overheads—which reduce the time available to the processes.
2. **Longer time quantum**
    A. risks making the scheduler too “FCFS-like”: unresponsive, unfair, low I/O device utilization, etc.
3. **Best value**

Quanta above 100ms should be avoided. A frequently chosen compromise for the quantum is between 20ms and 50ms.

Advantages of Round Robin
simplicity and strict "first come, first served" nature.

Disadvantages include the absence of a priority system: lots of low privilege processes may starve on high privilege one.

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
- Problem ≡ **Starvation** – low priority processes may never execute
- Solution ≡ **Aging** – as time progresses increase the priority of the process

Priority Inversion

- If a low priority thread **locks some resource shared with a high** priority thread process, in a computer busy with high priority threads.
- if a **high priority thread uses the shared data structure, it will also works slowly because it has to wait for the lock - Priority inversion.**

High priority threads wait for low priority ones, just reverse of the normal case.

- Solution: **temporarily raise priority of processes with waited lock, to** that of the waiting process.

Difficulty: it is usually not immediately clear who is waiting for whom.

Lottery Scheduling

- Lottery Scheduling is a simple algorithm that statistically guarantees a variable fraction of processor time to each runnable process.
- The concept is much like a lottery. At each scheduling decision, each runnable process is given a number of "lottery tickets". Then a random number is generated, corresponding to a specific ticket. The process with that ticket gets the quantum.
- Although there is no absolute guarantee that processes will be treated equally, the frequency of scheduling events in a preemptive multitasking system means that it comes very close to doing so.
- The number of tickets given to a process divided by the total number of tickets is the statistical fraction of the quanta given to that process. For example, if these processes are given this number of tickets:

  foo - 5
  bar - 7
  bash - 4

- The fractions of processor time given to each should be:

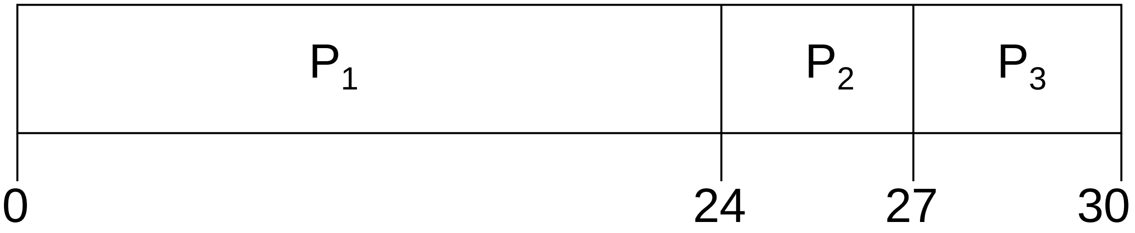
  foo - 5/16 - 31%
  bar - 7/16 - 44%
  bash - 4/16 - 25%

- Requires a reliable random number generator.

First-Come, First-Served (FCFS) Scheduling

Process	Burst Time
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:



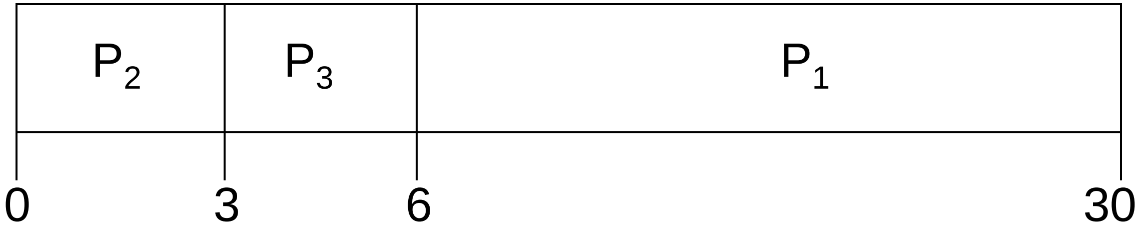
- Waiting time for P_1 = 0; P_2 = 24; P_3 = 27
- Average waiting time: (0 + 24 + 27)/3 = 17

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
- *Convoy effect* short process behind long process

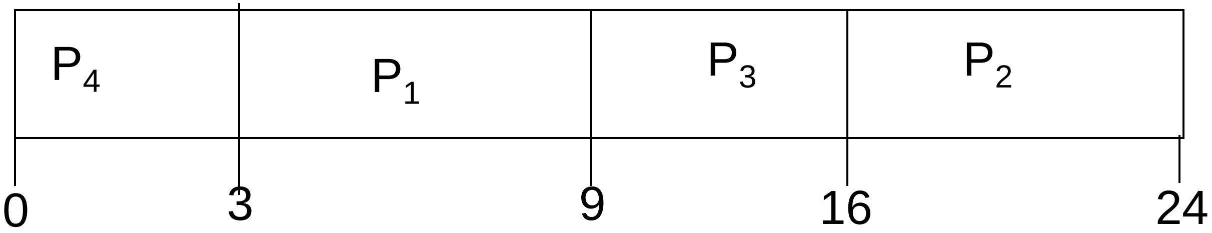
Shortest-Job-First (SJF) Scheduling

- Shortest-next-CPU-burst Algorithm
- 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.
  - Ties are broken by FCFS.

Example of SJF

Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3

- SJF scheduling chart



P_4 P_1 P_3 P_2

0 3 9 16 24

- Average waiting time (SJF) = (3 + 16 + 9 + 0) / 4 = 7
- Average waiting time (FCFS) = (0+6+14+21)/4 = 10.25

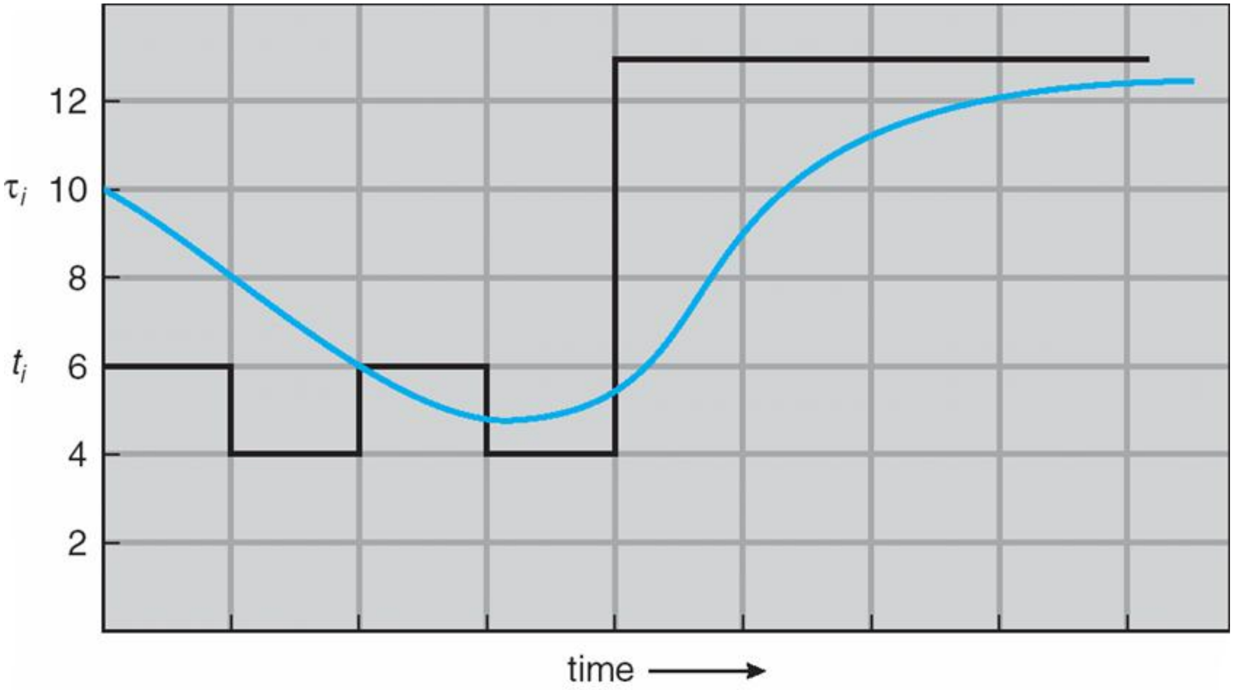
Determining Length of Next CPU Burst

1. $t_n =$ actual length of n^{th} CPU burst
2. $\tau_{n+1} =$ predicted value for the next CPU burst
3. $\alpha, 0 \le \alpha \le 1$
4. Define :

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

- Can estimate the length
- Can be done by using the length of previous CPU bursts, using exponential averaging

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

- 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

Shortest-Job-First (SJF) Scheduling

- Preemptive and Non-preemptive versions.
- Preemptive – Shortest Remaining Time First/Next (SRTF/ SRTN)

Q.1 – Solve using SRTN. Calculate Avg. Waiting time.

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

Highest Response Ratio Next

- the priority of each job is dependent on its estimated run time, and also the amount of time it has spent waiting. Jobs gain higher priority the longer they wait which prevents indefinite postponement or process starvation.

$$Priority = \frac{waiting\ time + estimated\ run\ time}{estimated\ run\ time} = 1 + \frac{waiting\ time}{estimated\ run\ time}$$

Earliest Deadline First

Theorem

- Given a task set of periodic or sporadic tasks, with relative deadlines equal to periods, the task set is schedulable by EDF if and only if

$$U = \sum_{i=1}^{N} \frac{C_i}{T_i} \leq 1$$

Corollary

EDF is an optimal algorithm, in the sense that if a task set is schedulable, then it is schedulable by EDF.

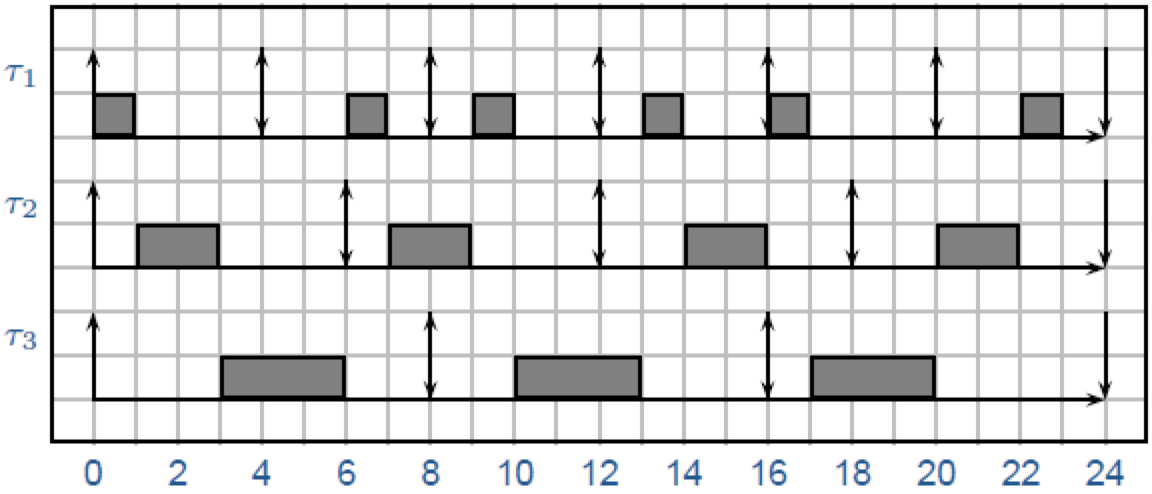
- In fact, if U > 1 no algorithm can successfully schedule the task set;
- if U <=1, then the task set is schedulable by EDF (and maybe by other algorithms also).

Example

$\tau_1 = (1,4)$, $\tau_2 = (2,6)$, $\tau_4 = (3,8)$.

$U = \frac{1}{4} + \frac{2}{6} + \frac{3}{8} = \frac{23}{24}$

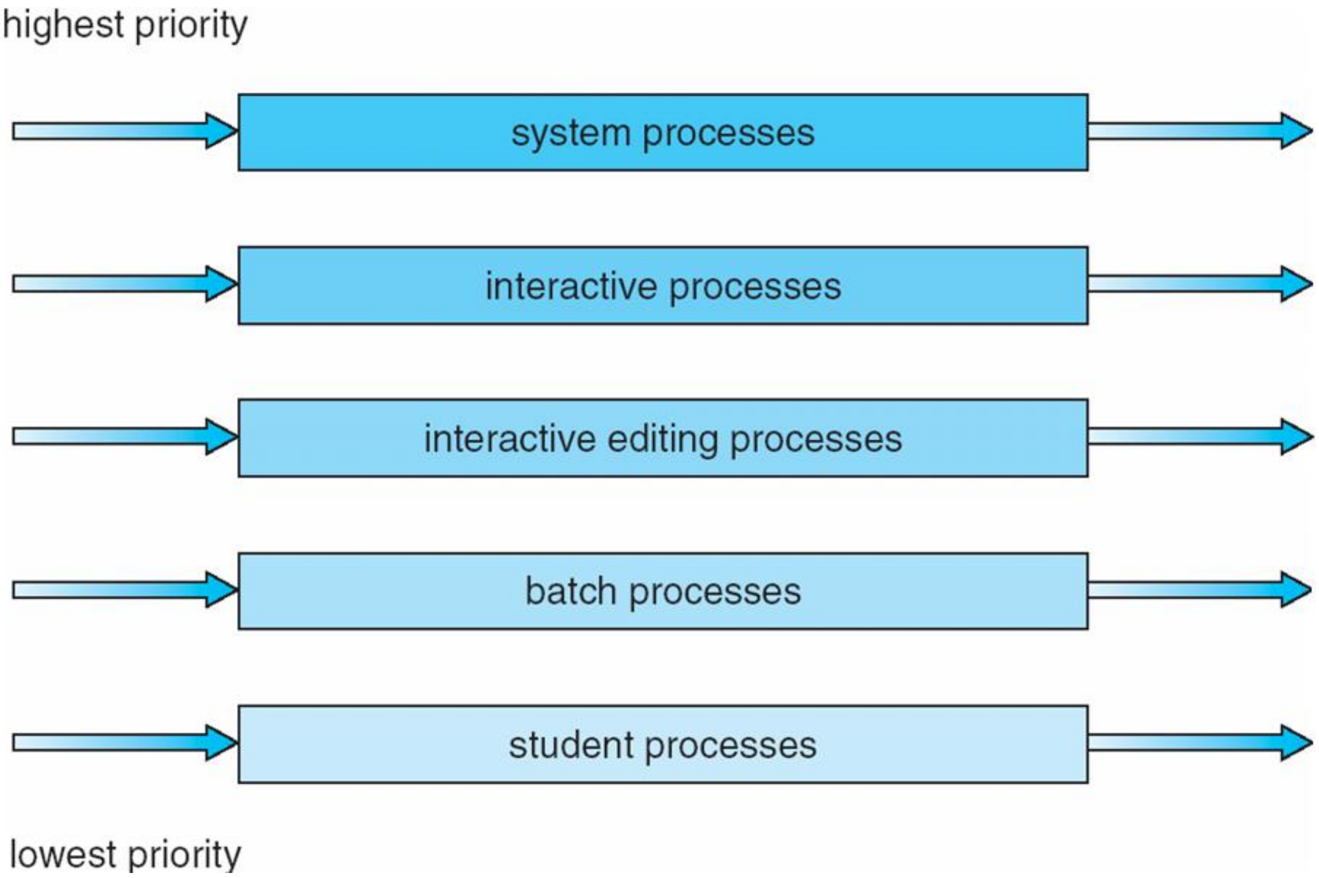
Again, the utilization is very high. However, no deadline miss in the hyperperiod.



Multilevel Queue

- Classify processes acc. to their resource needs and behavior.
- Ready queue is partitioned into separate queues:
  foreground (interactive)
  background (batch)
- Each queue has its own scheduling algorithm
  - foreground – RR
  - background – FCFS
- Scheduling must be done between the queues
  - Option 1 : Fixed priority preemptive scheduling; (i.e., serve all from foreground then from background). Possibility of starvation.
  - Option 2 : 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 and 20% to background in FCFS

Multilevel Queue Scheduling



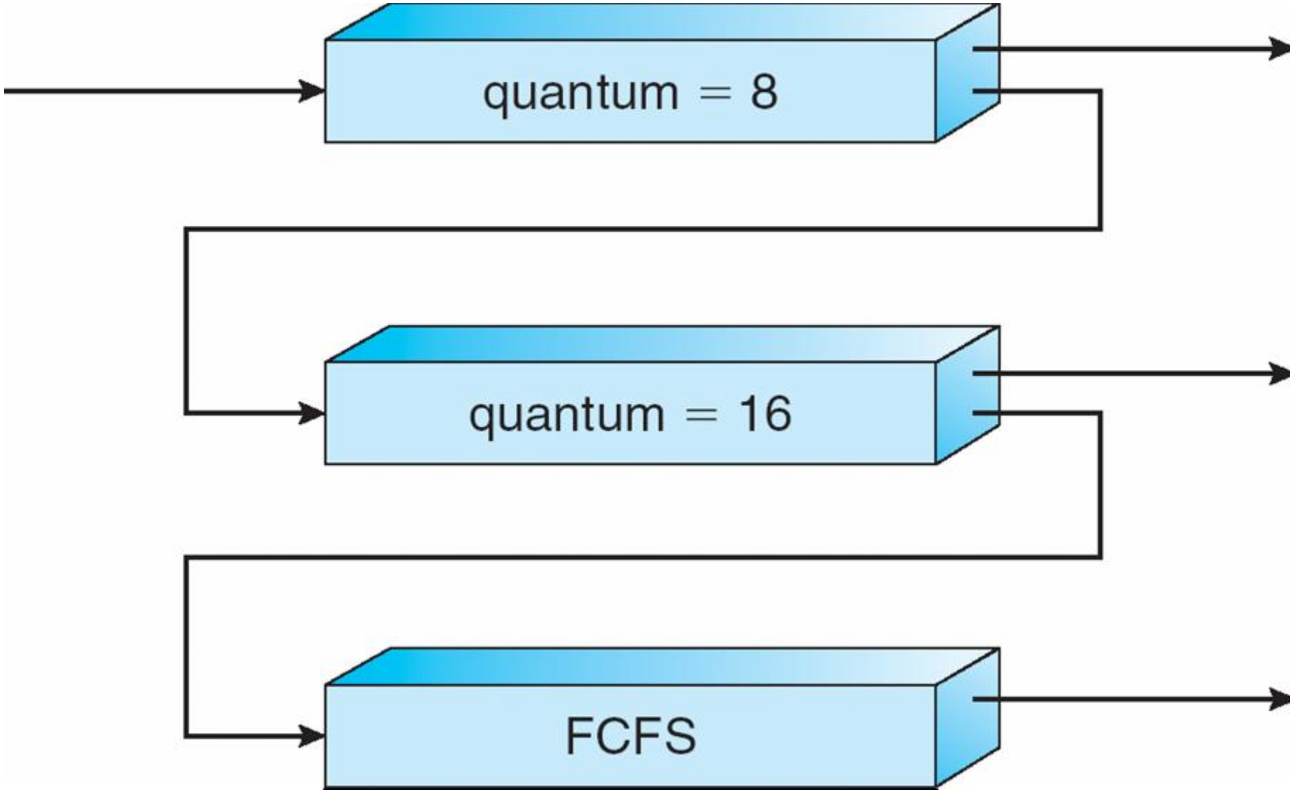
Multilevel Feedback Queue

- 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 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 and receives 16 additional milliseconds. If it still does not complete, it is preempted and moved to queue Q_2.

Multilevel Feedback 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
- **Processor affinity** – process has affinity for processor on which it is currently running
  - **soft affinity - Inter-process Migration**
  - **hard affinity – No Inter Process Migration**
- **Load Balancing**
  - **Per Processor Private Queue**
  - **Push and Pull Migration**

Operating System Examples

- Solaris scheduling
- Windows XP scheduling
- Linux scheduling

Linux Scheduling

- Constant order *O*(1) scheduling time.
  - It means that it will take a constant time to schedule n tasks.
- Two priority ranges: time-sharing and real-time
- **Real-time** range from 0 to 99 and **nice** value from 100 to 140

Objectives

- Response time
- Throughput for batch
- No starvation
- Accommodate high AND low priority

General Process Classifications

- Two schemes for classifying:
  - Traditional classes
    - IO bound
    - CPU bound
  - Alternative classes
    - Interactive
    - Batch
    - Real-time
      - Video, robotics, sensor-based

How to determine?

Source - DJ Foreman 3/8/2009

Linux Scheduling Classes

- FIFO FIFO real-time process
- RR Round Robin real-time process
- Normal non-real-time process

Real-time
Details

Real-time Rules

- Priority 0(highest) - 99(lowest)
  - Program adjustable:
    - *sched_setparam()*
    - *sched_setscheduler()*
- FIFO - non-interruptible except
  - Higher priority
  - Blocked
  - Yields
- RR
  - Adds a time-slice to each thread
  - Suspends at end of slice

Non-real-time Scheduling

(Conventional processes)

Rules

- Every process has a “static” priority
  - 100(highest) – 139 (lowest)
  - Variable via *nice*() and *setpriority*()
- Base quantum (not a constant!)
  - If sp<120
    - 20*(140-static priority)
  - If sp>=120
    - 5*(140-static priority)
- Dynamic priority (also 100-139)
  - Max(100, min(sp-bonus+5,139))
  - 0<=bonus<=10

Nice Value vs. static priority and Quantum

	Static Priority	NICE	Quantum
High Priority	100	-20	800 ms
	110	-10	600 ms
	120	0	100 ms
	120	+10	50 ms
Low Priority	139	+19	5 ms

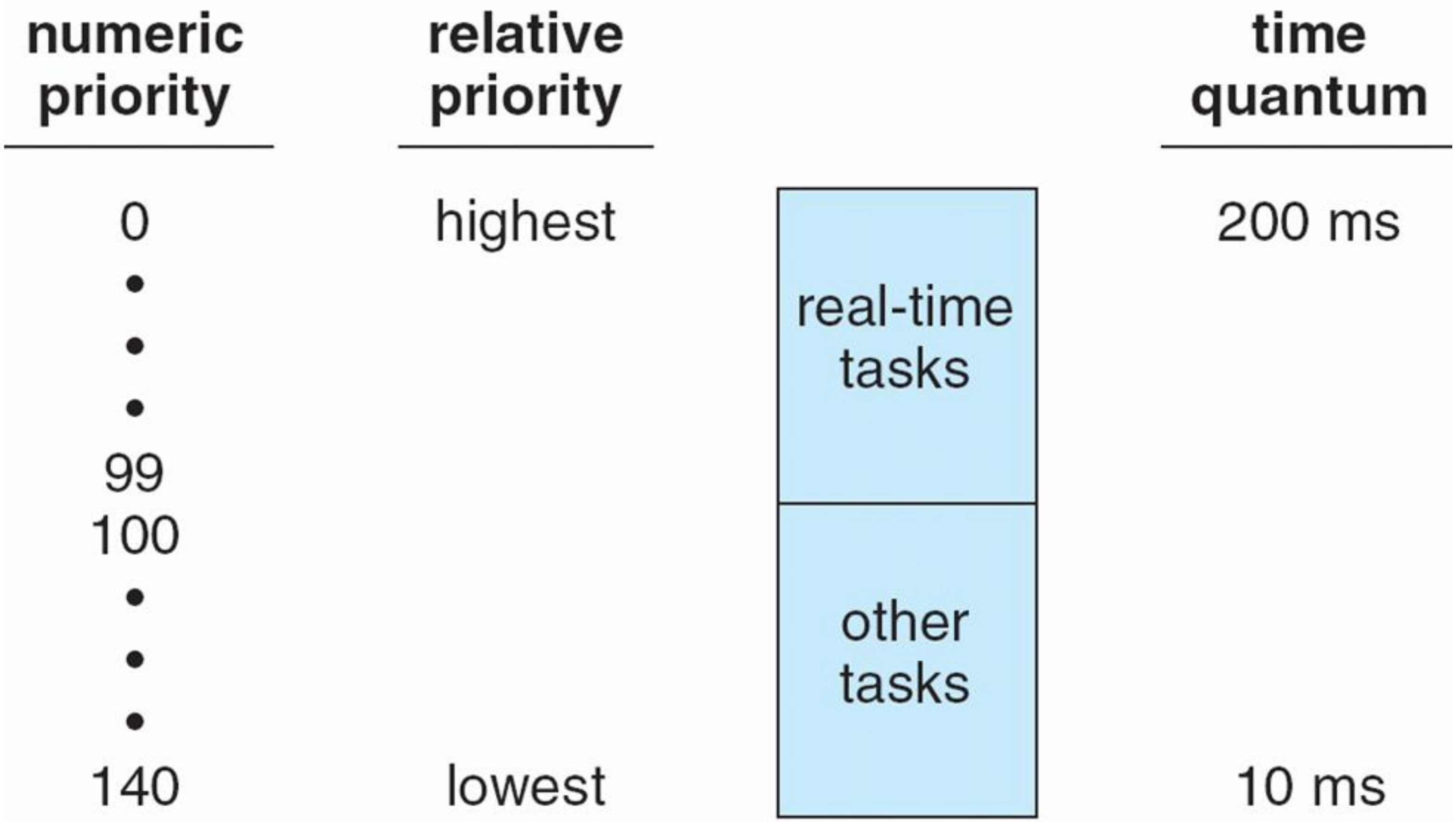
$$\text{Quantum} = \begin{cases} (140 - \text{SP}) \times 20 & \text{if SP} < 120 \\ (140 - \text{SP}) \times 5 & \text{if SP} \geq 120 \end{cases}$$

Rules (continued)

- Bonus
  - 0-5 is a penalty
  - 6-10 is a prize
  - Related to average sleep time
    - OF THIS PROCESS (not of all Processes)

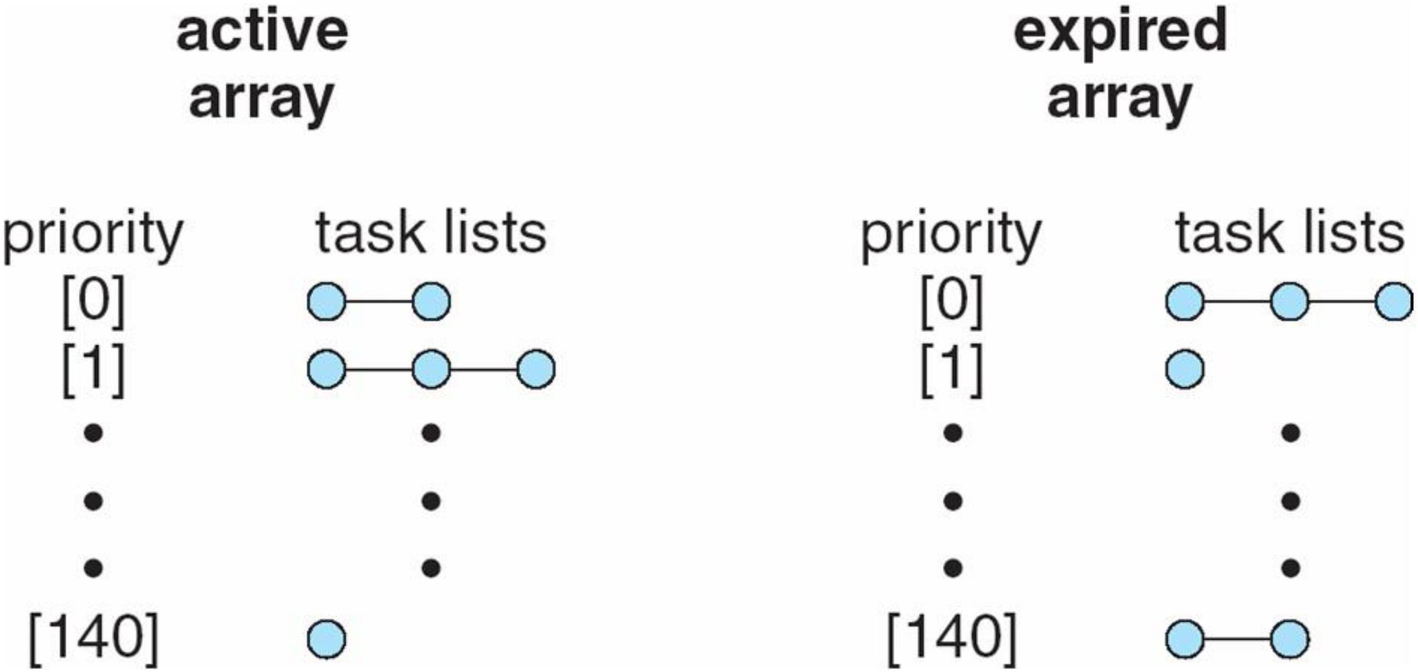
Average sleep time	Bonus
0<=N<100ms	0
300<=N<400ms	3
600<=N<700	6
900<=N<1000	9

Priorities and Time-slice length



numeric priority	relative priority		time quantum
0	highest	real-time tasks	200 ms
•			
•			
•			
99			
100		other tasks	
•			
•			
•			
140	lowest		10 ms

List of Tasks Indexed According to Priorities



System Calls Related to Scheduling

http://oreilly.com/catalog/linuxkernel/chapter/ch10.html

System Call	**Description**
nice( )	Change the priority of a conventional process.
getpriority( )	Get the maximum priority of a group of conventional processes.
setpriority( )	Set the priority of a group of conventional processes.
sched_getscheduler( )	Get the scheduling policy of a process.
sched_setscheduler( )	Set the scheduling policy and priority of a process.
sched_getparam( )	Get the scheduling priority of a process.
sched_setparam( )	Set the priority of a process.
sched_yield( )	Relinquish the processor voluntarily without blocking.
sched_get_ priority_min( )	Get the minimum priority value for a policy.
sched_get_ priority_max( )	Get the maximum priority value for a policy.
sched_rr_get_interval( )	Get the time quantum value for the Round Robin policy.

Algorithm Evaluation

- Deterministic modeling – takes a particular predetermined workload and defines the performance of each algorithm for that workload
- Benchmarks like SPEC2000
- Queuing models
- Implementation

nice/renice

- Q2. How to invoke nice/renice?
- To higher/lower priority of a process
- Requires root.

The "niceness" of a process is a numeric hint to the kernel about how the process should be treated in relation to other processes contending for the CPU. The strange name is derived from the fact that it determines how nice you are going to be to other users of the system. A high nice value means a low priority for your process: you are going to be nice. A low or negative value means high priority: you are not very nice. The range of allowable niceness values is -20 to +19.

More

http://oreilly.com/catalog/linuxkernel/chapter/ch10.html