##### Lecture 14,15,16,17

##### Topics To Be Covered

##### CPU Scheduling

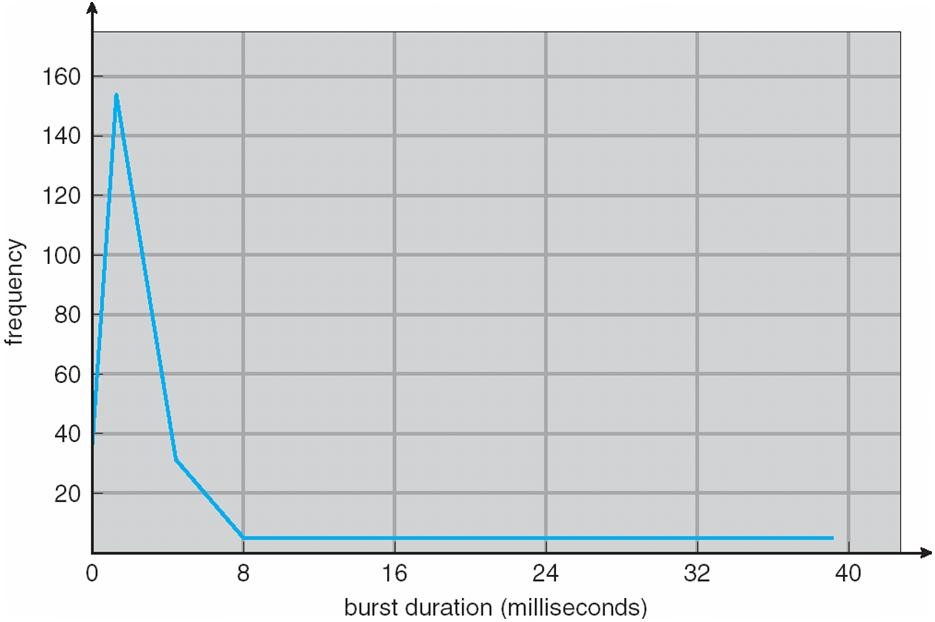
To introduce CPU scheduling, which is the basis for multiprogrammed operating systems To describe various CPU-scheduling algorithms

To discuss evaluation criteria for selecting a CPU-scheduling algorithm for a particular system Maximum CPU utilization obtained with multiprogramming

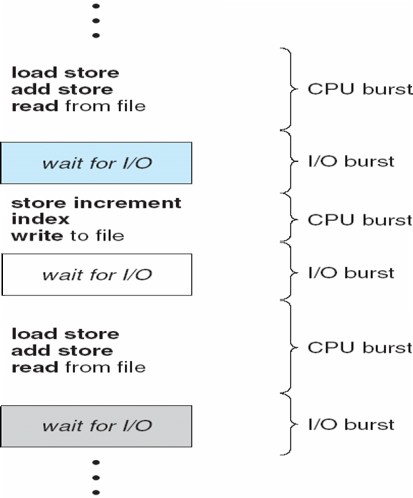
CPU–I/O Burst Cycle – Process execution consists of a *cycle* of CPU execution and I/O wait

**CPU burst** distribution

##### Histogram of CPU-burst Times



**Alternating Sequence of CPU And I/O Bursts**



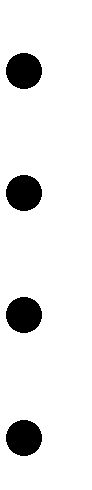
**CPU Scheduler**

Selects from among the processes in memory that are ready to execute, and allocates the CPU to one of them 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** All other scheduling is **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

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

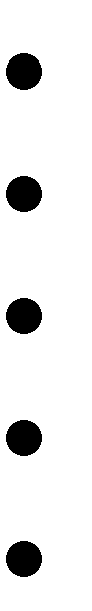
##### Scheduling Criteria

**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)

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

**Following are the main scheduling algorithm**

##### First-Come, First-Served (FCFS) Scheduling

Process Burst Time

*P1* 24

*P2* 3

*P3* 3

Suppose that the processes arrive in the order: *P1* , *P2* , *P3*

The Gantt Chart for the schedule is:

|  |  |  |
| --- | --- | --- |
| P1 | P2 | P3 |
|  |  |  |

## 0 24 27 30

Waiting time for *P1* = 0; *P2* = 24; *P3* = 27 Average waiting time: (0 + 24 + 27)/3 = 17 Suppose that the processes arrive in the order

*P2* , *P3* , *P1*

The Gantt chart for the schedule is:nnnnWaiting time for *P1 =* 6*; P2* = 0*; P3 =* 3nAverage waiting time: (6 + 0 + 3)/3 = 3

Much better than previous case

*Convoy effect* short process behind long process

## 0 3 6 30

|  |  |  |
| --- | --- | --- |
| P2 | P3 | P1 |
|  |  |  |

##### 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

|  |  |  |
| --- | --- | --- |
| Process | Arrival Time | Burst Time |
| *P1* | 0.0 | 6 |
| *P2* | 2.0 | 8 |
| *P3* | 4.0 | 7 |
| *P4* | 5.0 | 3 |

SJF scheduling chart

average waiting time = (3 + 16 + 9 + 0) / 4 = 7the length of the next CPU request

## 0 3 9 16 24

|  |  |  |  |
| --- | --- | --- | --- |
| P4 | P1 | P3 | P2 |
|  |  |  |  |

##### Determining Length of Next CPU Burst

* 1. *tn*

# actual

length of

*nth*

CPU

# burst

* 1. *n* 1



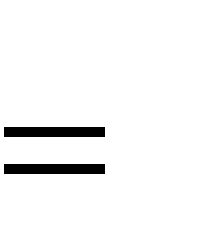
# predicted value

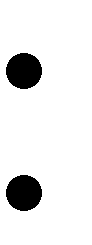
for

# the next

CPU

# burst

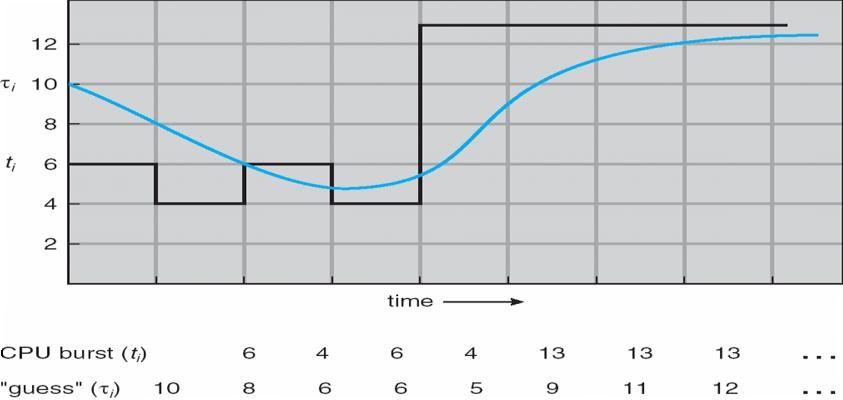
3. , 0 1

4. Define :

Can only 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



**Examples of Exponential Averaging**

a =0 tn+1 = tn

Recent history does not count a =1

tn+1 = a *t*n

Only the actual last CPU burst counts If we expand the formula, we get: t*n*+1 = a t*n*+(1 *-* a*)*a *tn* -1 + …

*+(*1 - a *)j* a *tn* -*j* + …

*+(*1 - a *)n* +1 t0

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

##### Priority Scheduling

A priority number (integer) is associated with each process

The CPU is allocated to the process with the highest priority (smallest integer º highest priority)

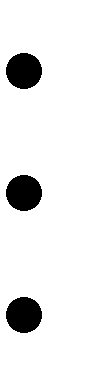
Preemptive nonpreemptive

SJF is a priority scheduling where priority is the predicted next CPU burst time Problem º **Starvation** – low priority processes may never execute

Solution º **Aging** – as time progresses increase the priority of the process

##### 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 Þ FIFO

*q* small Þ *q* must be large with respect to context switch, otherwise overhead is too high

##### Example of RR with Time Quantum = 4

|  |  |
| --- | --- |
| Process | Burst Time |
| *P1* | 24 |
| *P2* | 3 |
| *P3*  The Gantt chart is: | 3 |

Typically, hig

her avera

ge turnar

ound than

SJF, but

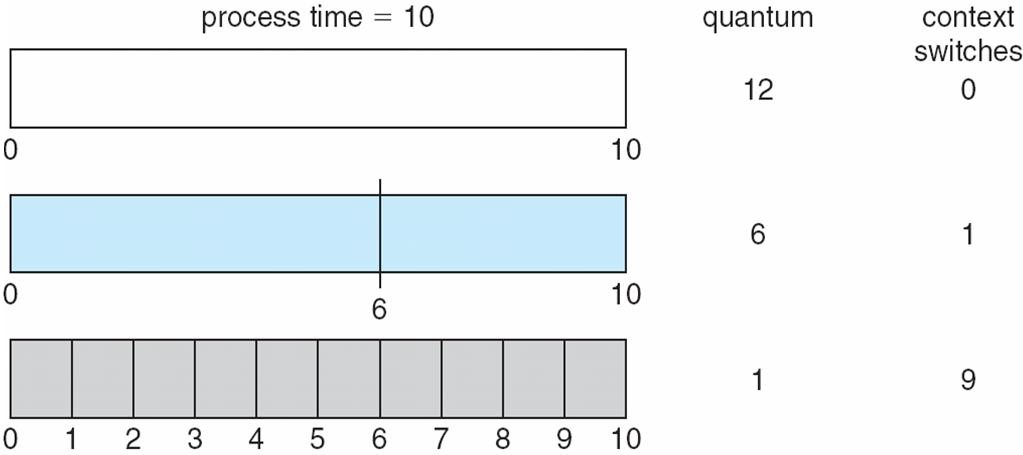
better *re*

*sponse*

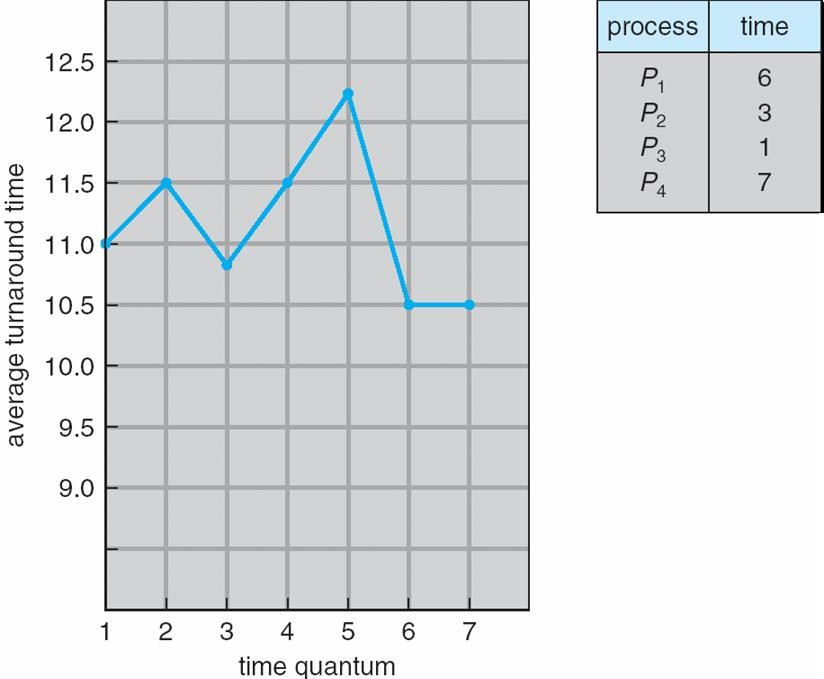
|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| P1 | P2 | P3 | P1 | P1 | P1 | P1 | P1 |

0 4 7 10 14 18 22 26 30

##### Time Quantum and Context Switch Time

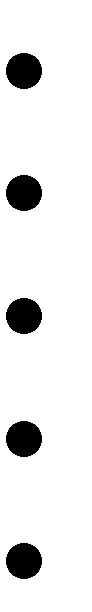


**Turnaround Time Varies With The Time Quantum**



**Multilevel Queue**

 Ready queue is partitioned into separate queues: foreground (interactive)

background (batch)

Each queue has its own scheduling algorithm foreground – RR

background – FCFS

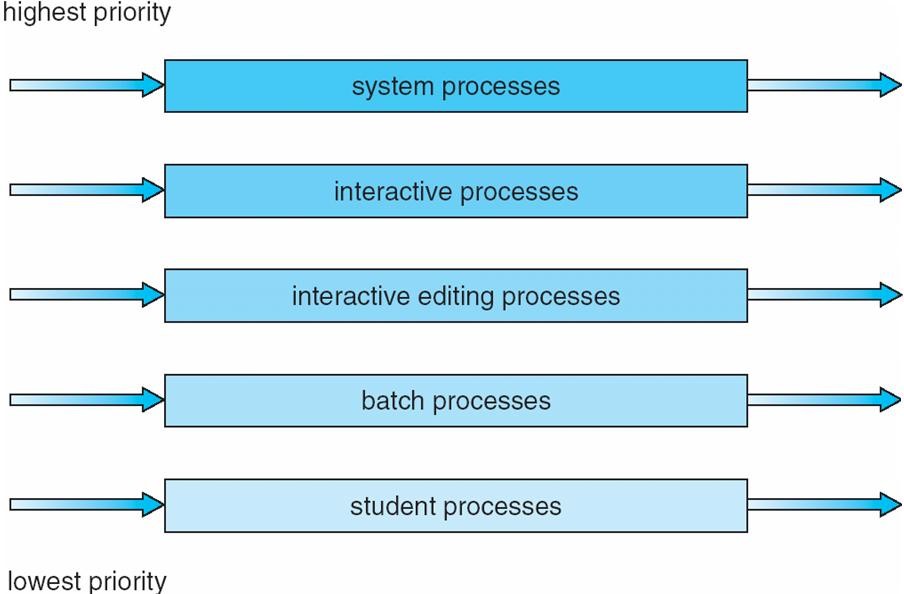
Scheduling must be done between the queues

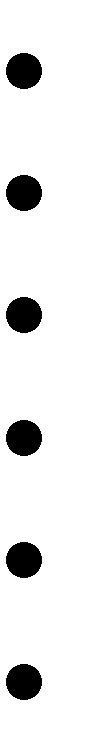
Fixed priority scheduling; (i.e., serve all from foreground then from background). Possibility of starvation.

 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

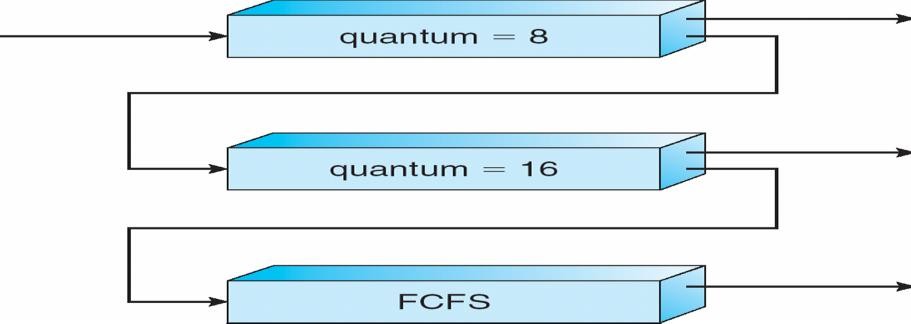
*Q*2 – FCFS

Scheduling

A new job enters queue *Q0* 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.

##### Multilevel Feedback Queues

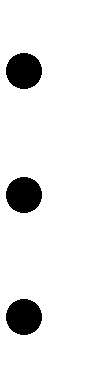


##### Multiple-Processor Scheduling

CPU scheduling more complex when multiple CPUs are available

**Homogeneous processors** within a multiprocessor

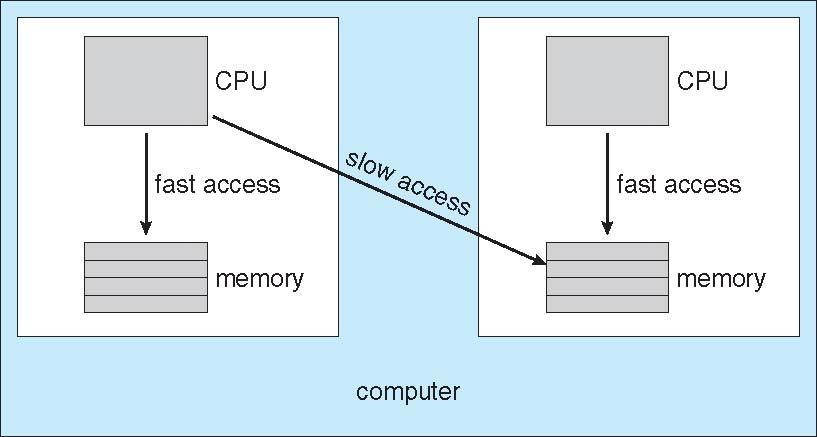
**Asymmetric multiprocessing** – only one processor accesses the system data structures, alleviating the need for data sharing

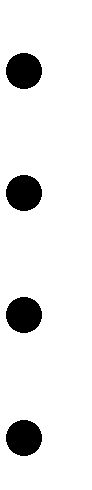
 **Symmetric multiprocessing (SMP)** – each processor is self-scheduling, all processes in common ready queue, or each has its own private queue of ready processes

**Processor affinity** – process has affinity for processor on which it is currently running

##### soft affinity hard affinity

**NUMA and CPU Scheduling**



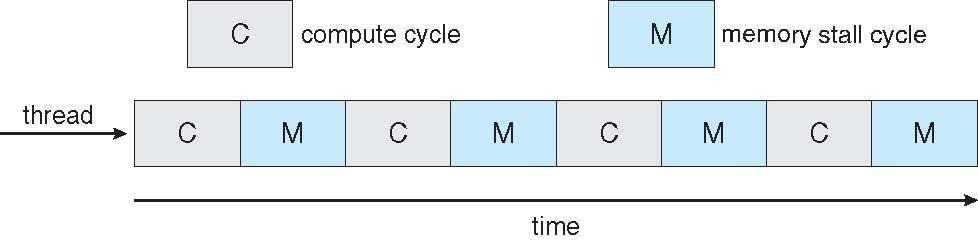
**Multicore Processors**

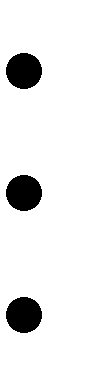
Recent trend to place multiple processor cores on same physical chip Faster and consume less power

Multiple threads per core also growing

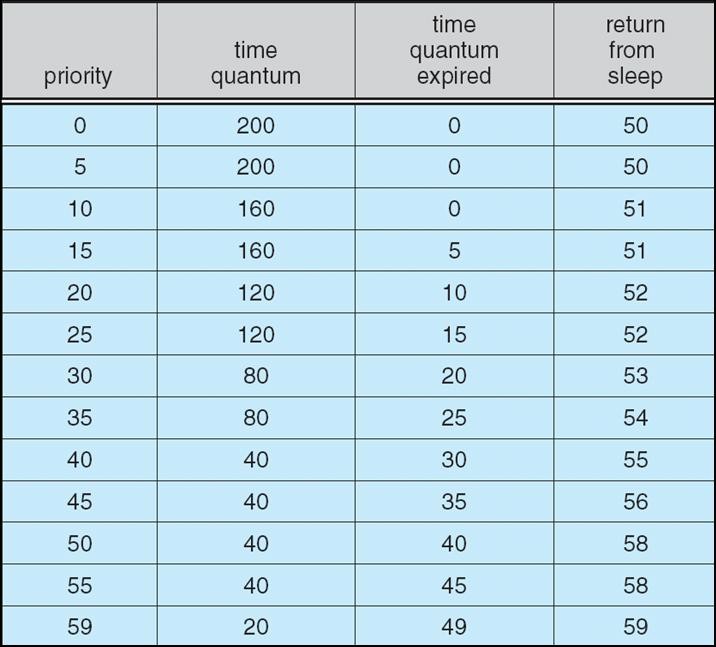
Takes advantage of memory stall to make progress on another thread while memory retrieve happens

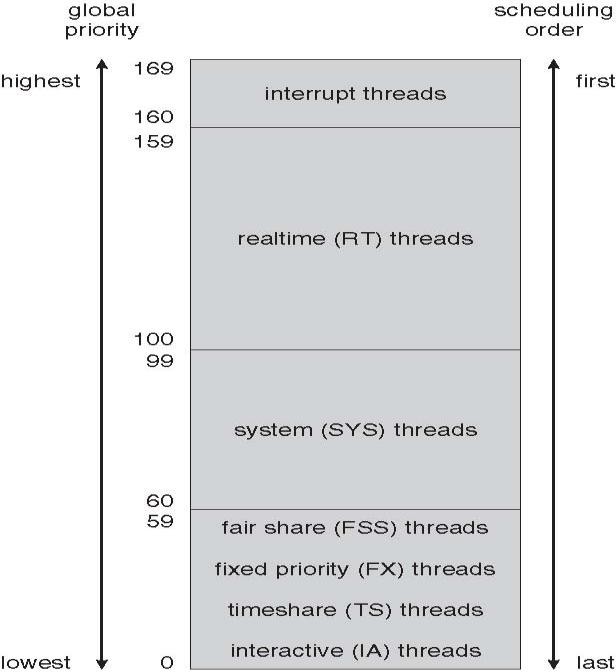
##### Multithreaded Multicore System



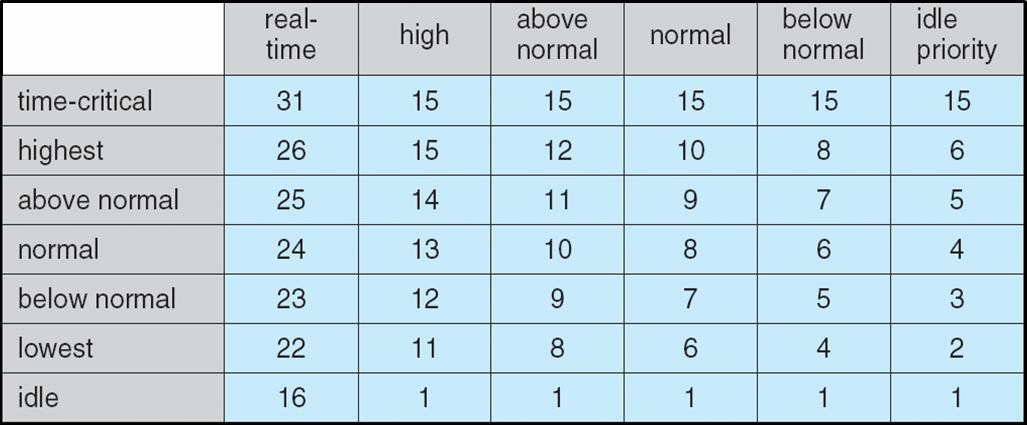
**Operating System Examples** Solaris scheduling Windows XP scheduling Linux scheduling

##### Solaris Dispatch Table

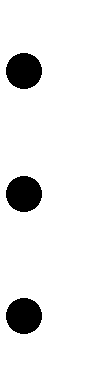


**Solaris Scheduling**

**Windows XP Priorities**



**Linux Scheduling**

Constant order *O*(1) scheduling time

Two priority ranges: time-sharing and real-time

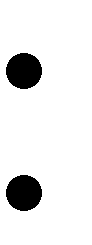
**Real-time** range from 0 to 99 and **nice** value from 100 to 140

##### Priorities and Time-slice length

**List of Tasks Indexed According to Priorities**



**Algorithm Evaluation**

 Deterministic modeling – takes a particular predetermined workload and defines the performance of each algorithm for that workload

Queuing models Implementation

##### Evaluation of CPU schedulers by Simulation

