# Chapter 5:  CPU Scheduling

# Chapter 5: CPU Scheduling

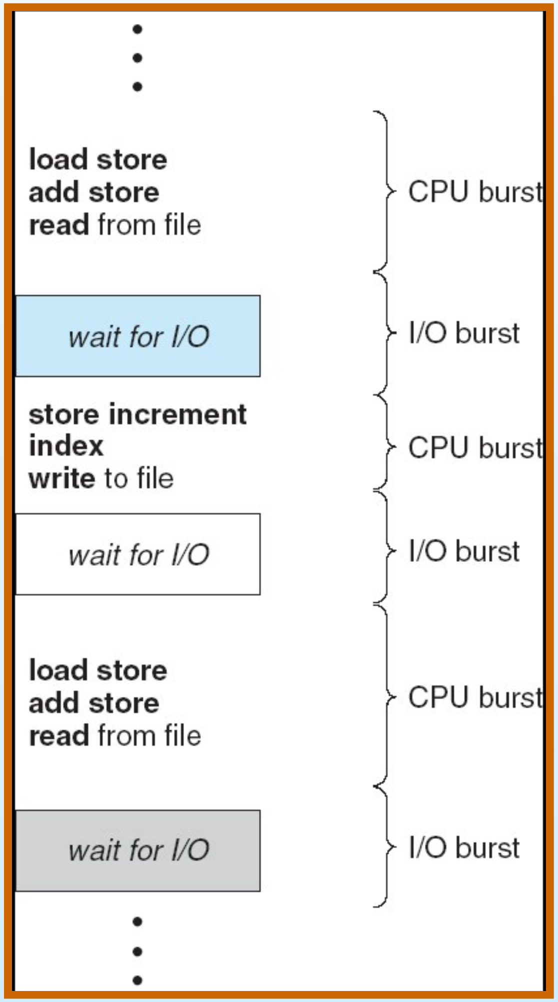
- Basic Concepts
- Scheduling Criteria
- Scheduling Algorithms
- Multiple-Processor Scheduling
- Operating Systems Examples

# 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


Silberschatz, Galvin and Gagne ©2005, Sunner©2008

# Alternating Sequence of CPU And I/O Bursts



# CPU Scheduler ( 调度 )

- Selects one of the processes in ready queue to run next
- 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

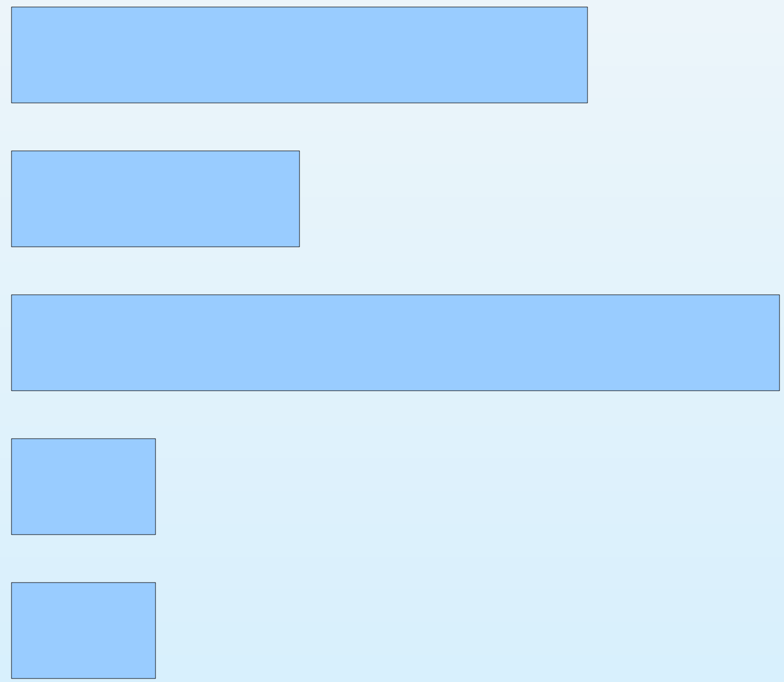
- Gives the CPU to the process selected by scheduler; this involves:
  - switching context
  - switching to user mode
  - jumping to the proper location in the user program to restart that program
- *Dispatch latency*
  - time it takes for the dispatcher to stop one process and start another running

# Scheduling Criteria

- CPU utilization – to max
  - keep the CPU as busy as possible
- Throughput – to max
  - # of processes that complete execution per time unit
- Turnaround time – to min
  - amount of time to execute a process
- Waiting time – to min
  - amount of time a process has been waiting in the ready queue
- Response time – to min
  - amount of time from a request was submitted to the first response is produced

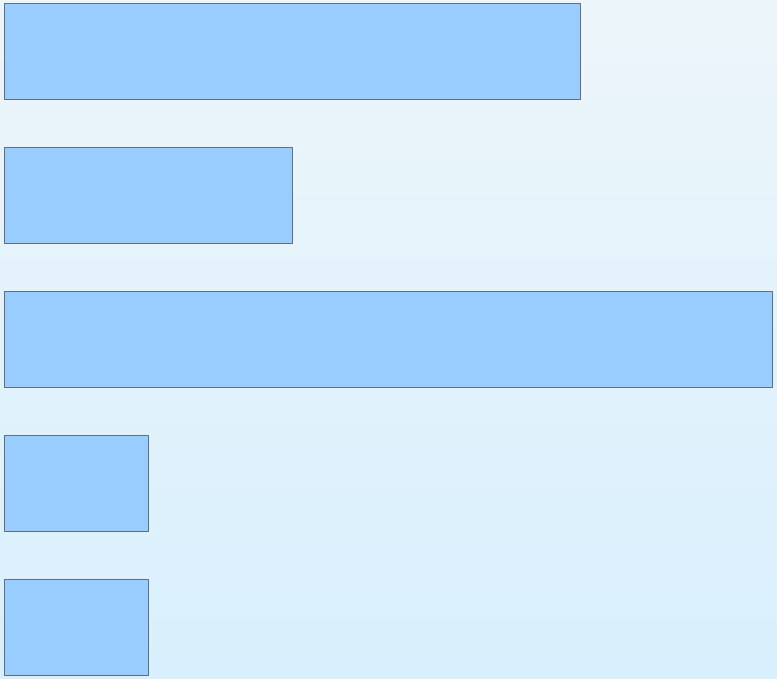
# How to Schedule?

| Process | Arrival time | Burst time |
|---|---|---|
| A | 0 | |
| B | 0 | |
| C | 0 | |
| D | 0 | |
| E | 0 | |



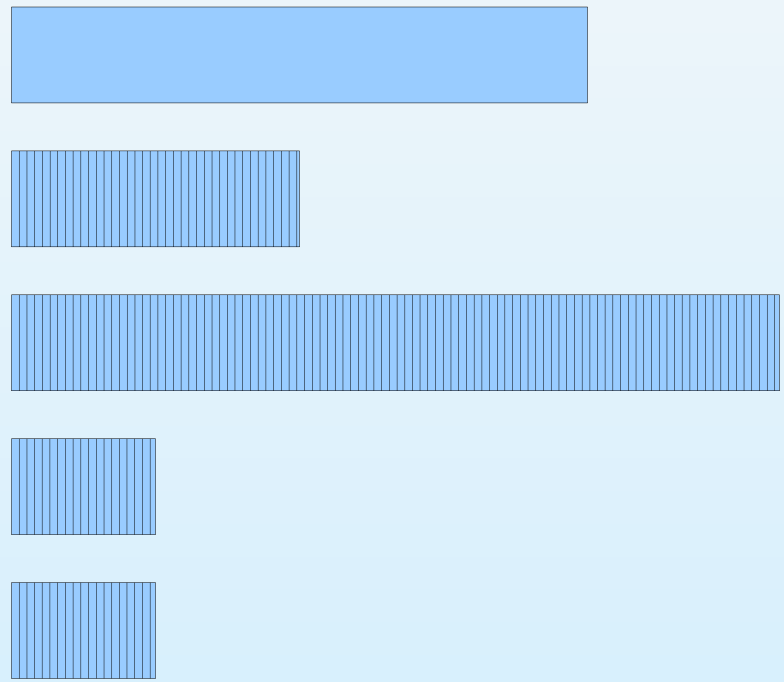
# How to Schedule?

| Process | Arrival time | Burst time |
|---|---|---|
| A | 0 | |
| B | 1 | |
| C | 2 | |
| D | 3 | |
| E | 4 | |



# How to Schedule?

| Process | Arrival time | Burst time |
|---|---|---|
| A | 0 | |
| B | 0 | |
| C | 0 | |
| D | 0 | |
| E | 0 | |



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

| Process | Burst Time |
|---|---|
| $P_1$ | 24 |
| $P_2$ | 3 |
| $P_3$ | 3 |

- **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**

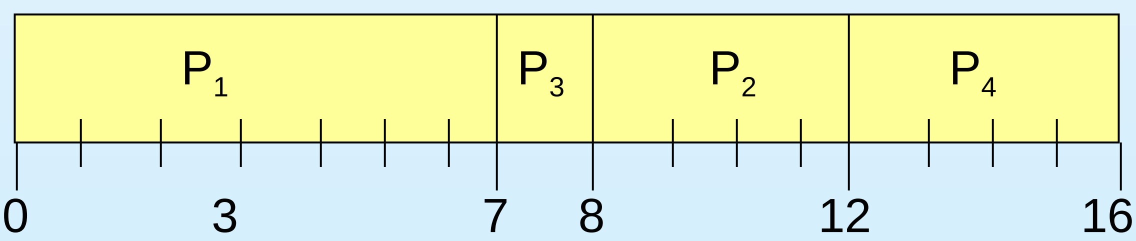
# Shortest-Job-First (SJF) Scheduling

- Schedule the process who has the shortest next CPU burst
- Two schemes:
  - Nonpreemptive
    - Process cannot be preempted until completes its CPU burst
  - Preemptive
    - if a new process arrives with CPU burst length less than remaining time of current executing process, preempt.
    - Shortest-Remaining-Time-First (SRTF)
- SJF is optimal
  - minimum average waiting time

# Example of Non-Preemptive SJF

| Process | Arrival Time | Burst Time |
|---|---|---|
| $P_1$ | 0.0 | 7 |
| $P_2$ | 2.0 | 4 |
| $P_3$ | 4.0 | 1 |
| $P_4$ | 5.0 | 4 |

- SJF (non-preemptive)

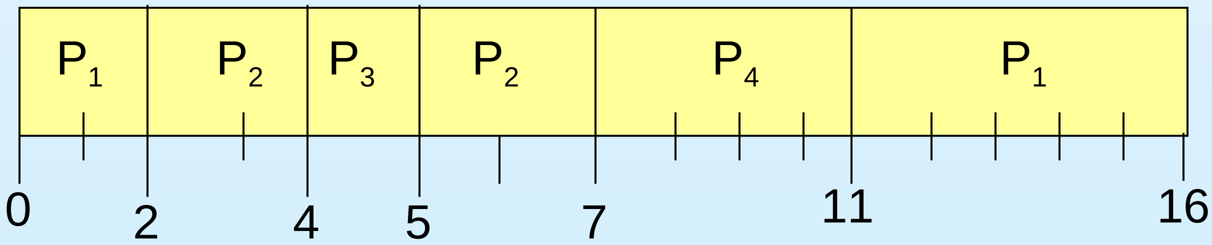



- Average waiting time = (0 + 6 + 3 + 7)/4 = 4

# Example of Preemptive SJF

| Process | Arrival Time | Burst Time |
|---|---|---|
| $P_1$ | 0.0 | 7 |
| $P_2$ | 2.0 | 4 |
| $P_3$ | 4.0 | 1 |
| $P_4$ | 5.0 | 4 |

- SJF (preemptive)




- Average waiting time = (9 + 1 + 0 +2)/4 = 3

# How to Know Length of Next CPU Burst?

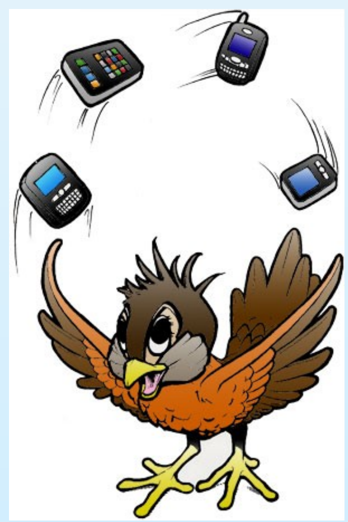
- Can only estimate the length
  - using the length of previous CPU bursts

# Priority Scheduling

- A priority number is associated with each process
- The CPU is allocated to the process with the highest priority (smallest integer ≡ highest priority)
  - Preemptive
  - nonpreemptive
- SJF is a priority scheduling where priority is the predicted next CPU burst time
- Problem ≡ Starvation
  - low priority processes may never execute
- Solution ≡ 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.



# Example of RR with Time Quantum = 20

| Process | Burst Time |
|---|---|
| $P_1$ | 53 |
| $P_2$ | 17 |
| $P_3$ | 68 |
| $P_4$ | 24 |

- The Gantt chart is:

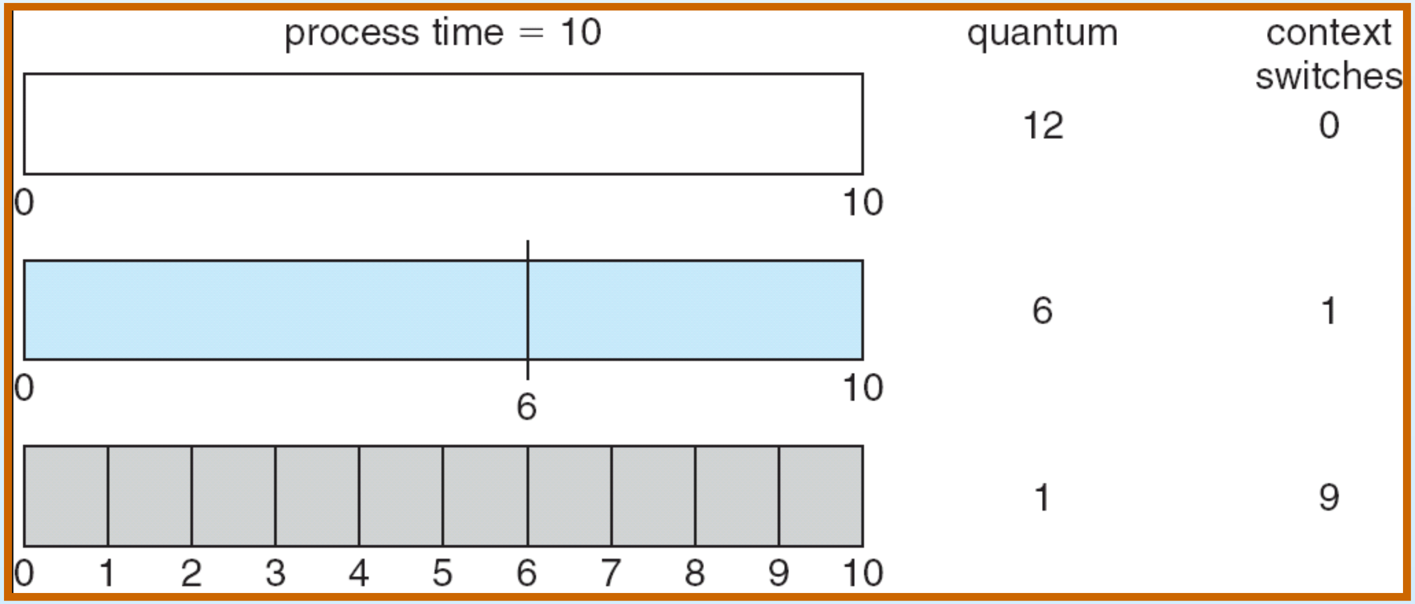
| $P_1$ | $P_2$ | $P_3$ | $P_4$ | $P_1$ | $P_3$ | $P_4$ | $P_1$ | $P_3$ | $P_3$ |
|---|---|---|---|---|---|---|---|---|---|

0 20 37 57 77 97 117 121 134 154 162

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

# Time Quantum and Context Switch Time

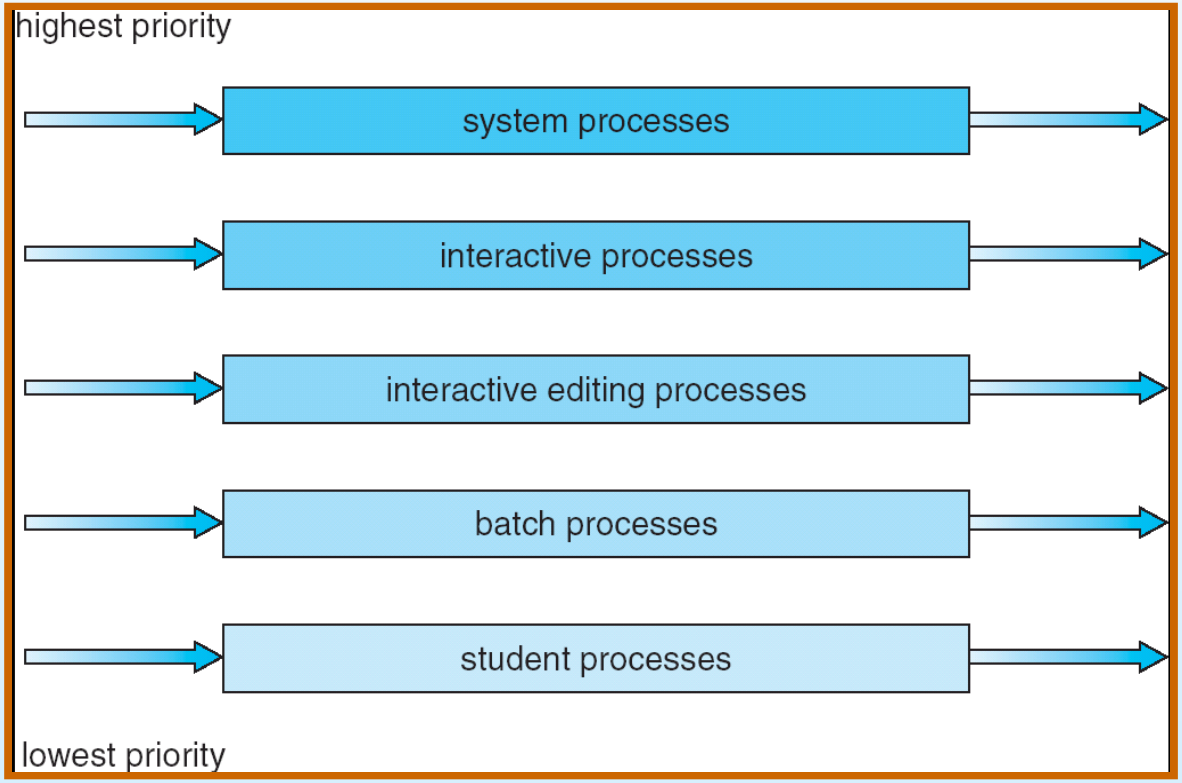
- Performance
  - *quantum* large ⇒ FCFS
  - *quantum* small ⇒ *q* must be large with respect to context switch, otherwise overhead is too high



# Multilevel Queue

- Ready queue is partitioned into separate queues
  - i.e., 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

# Multilevel Queue Scheduling



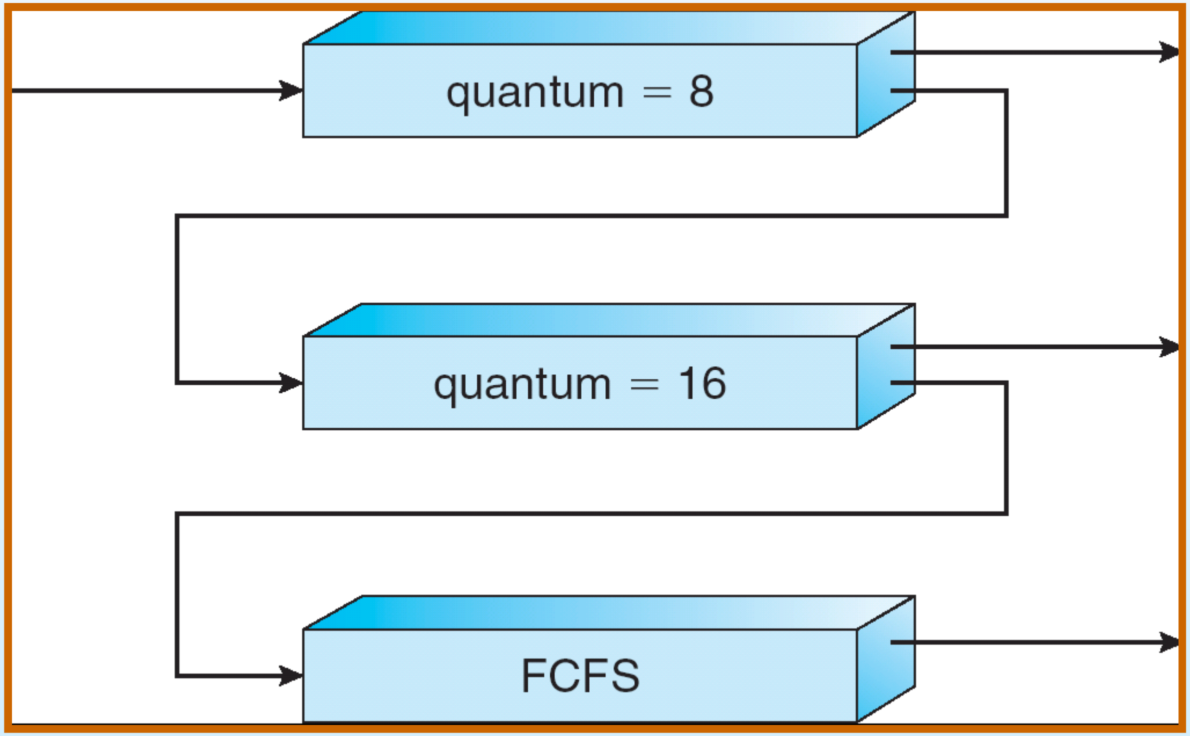
# Multilevel Feedback Queue

- A process can move between the 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
    - when to demote a process
    - which queue a process will enter when that process needs I/O

# 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 $Q_0$, receives 8 ms.
  - If it does not finish in 8 ms, job is moved to queue $Q_1$.
  - At $Q_1$ job 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

- Multiple CPUs give us
  - *Load sharing*
  - More complex CPU scheduling
- We discuss *homogeneous processors* only

# Approaches to Multiple-Processor Scheduling

- *Asymmetric multiprocessing*
  - Only one processor runs OS
  - Other processors execute only user code
  - Reduce the need for data sharing
- *Symmetric multiprocessing (SMP)*
  - All processes are equal to run processes
    - Every processor has a private ready queue (common way)
    - All processors share one ready queue
  - The scheduler must be programmed carefully
  - Supported by Windows XP/2000, Solaris, Linux and Mac OS X

# Processor Affinity ( 亲和度 )

- Avoid migrating a process from CPU 0 to CPU 1
  - Invalidate cache of CPU 0
  - Re-populate cache of CPU 1
- *Processor affinity*
  - A process has an affinity for the processor on which it is running
- *Soft affinity*
  - Attempting to keep affinity but not guaranteeing
- *Hard affinity*
  - Specifying a process which will never migrate
    - `sched_setaffinity()` in Linux,
    - `SetProcessAffinityMask()` in Windows 2000/XP/Vista

# Load Balancing

- Attempting to keep the workload distributed across all processors
- *Push migration*
  - A specific task periodically pushes process from overloaded to idle or less-busy processors
- *Pull migration*
  - An idle processor pulls a waiting task from a busy processor
- Some OSes implement both
  - i.e., Linux and FreeBSD
- Load balancing vs. processor affinity

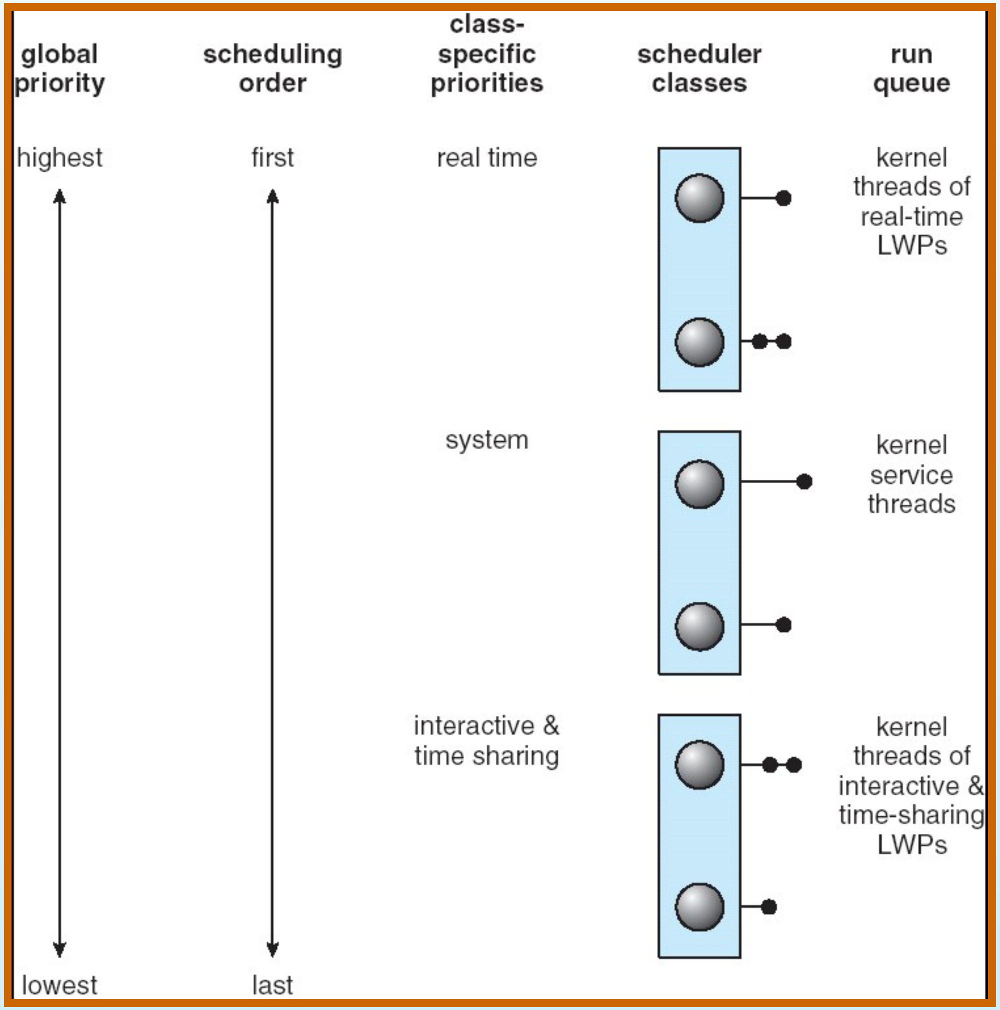
# Operating System Examples

- Solaris scheduling
- Windows XP scheduling
- Linux scheduling


Silberschatz, Galvin and Gagne ©2005, Sunner©2008

# Solaris Scheduling

- **Priority-based thread scheduling**



# Solaris Dispatch Table

| 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 Priorities

- Schedule threads using a priority-based, preemptive scheduling algorithm
- A thread will run until
  - Preempted by a higher-priority thread
  - Terminates
  - Time quantum ends
  - Calls a blocking system call
- Priorities
  - Variable class: 1-15
  - Real-time class: 16-31
  - Memory management: 0
  - Idle thread: no priority

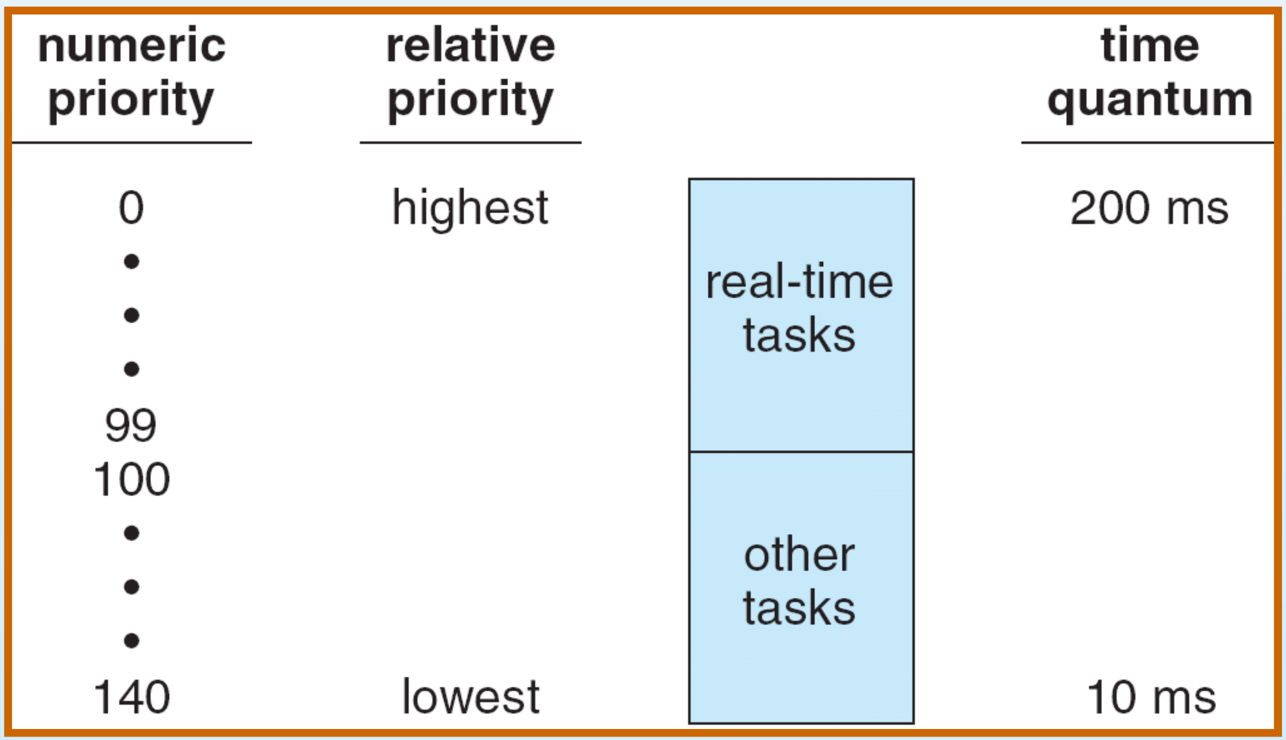
# Windows XP Priorities

| | 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 |

- **For threads in variable class, the priorities**
  - **Lowered after time quantum's running out**
  - **Never lowered below base priority (normal) of the class**
  - **Boosted when "waiting -> ready"**
    - **Interactive threads (active window) get more boost**
- **Gives foreground processes 3x quantum**

# Linux Scheduling

- A preemptive, priority-based algorithm
- Priorities
  - Real-time: 0-99
  - Nice: 100-140

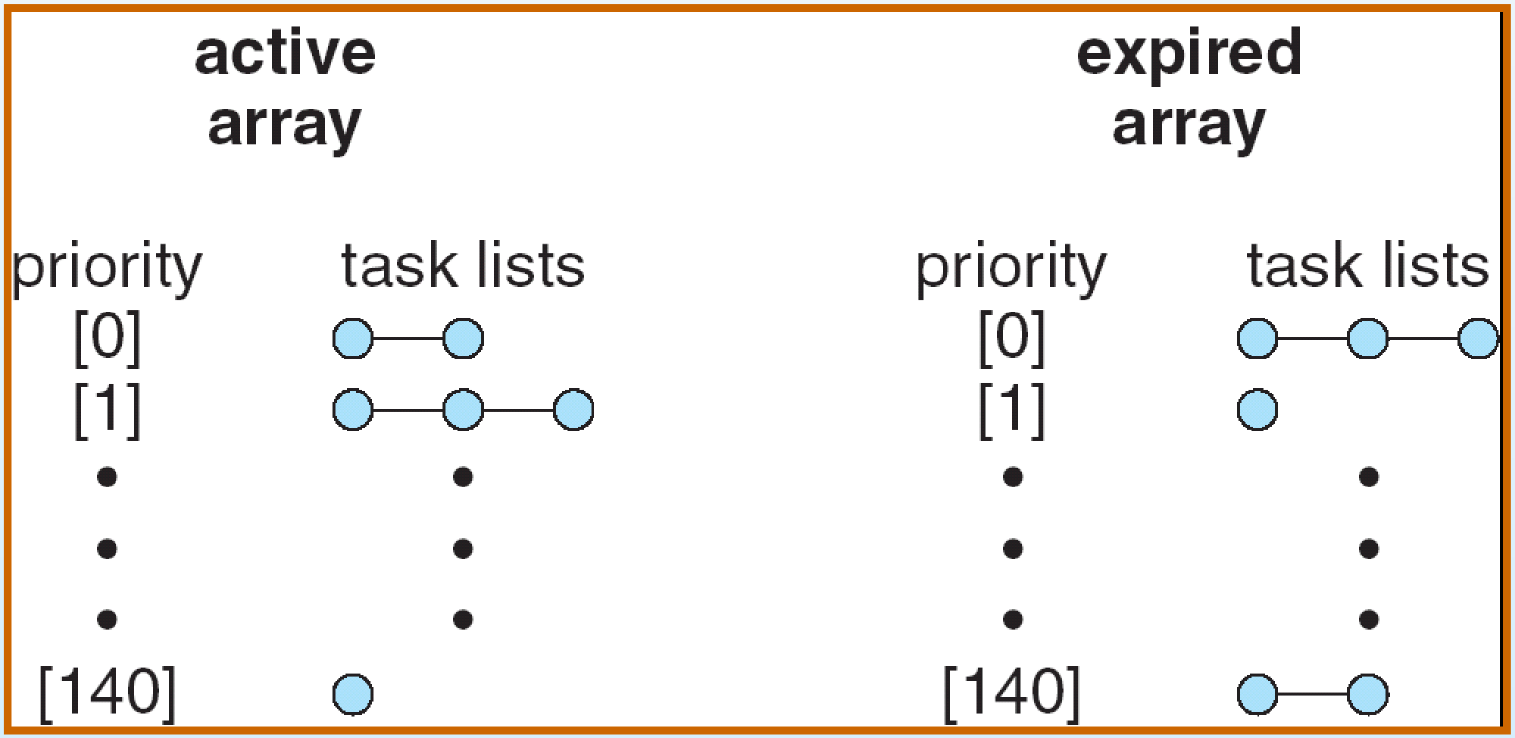


# Linux Scheduling (cont.)

- Each processor has one *runqueue* which contains all runnable tasks distributed to it
- Each runqueue contains two arrays
  - *Active* array contains all tasks with time remaining in their time slices
  - *Expired* array contains all expired tasks
- Task with the highest priority in the active array is chosen to run
- When active array is empty, the expired array becomes active array
  - And adjust priorities


Silberschatz, Galvin and Gagne ©2005, Sunner©2008

# List of Tasks Indexed According to Priorities



# Adjust Priorities

- A task's priority can be nice, nice – 5 or nice + 5
- Determined by how long it has been waiting
  - Interactive tasks, nice – 5
  - CPU-bound tasks, nice + 5
- Priorities are updated when moving tasks from active array to expired array

# End of Chapter 5