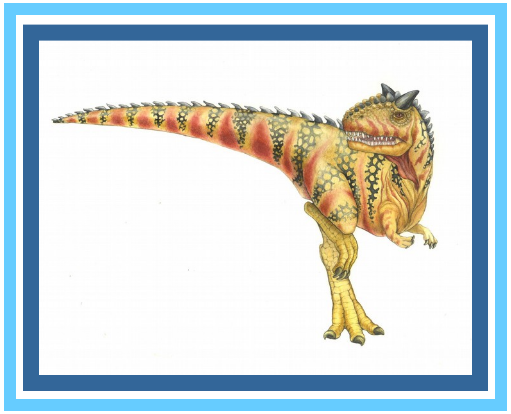
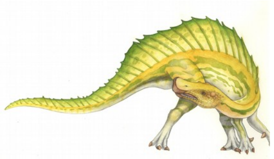
# Chapter 5: CPU Scheduling



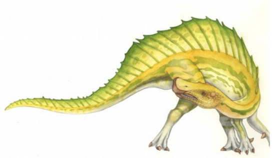
# Chapter 5:  CPU Scheduling

- Basic Concepts
- Scheduling Criteria
- Scheduling Algorithms
- Thread Scheduling
- Multi-Processor Scheduling
- Real-Time CPU Scheduling
- Operating Systems Examples
- Algorithm Evaluation



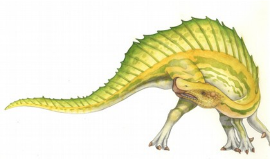
# Objectives

- Understand difference between different types of scheduling
- Describe various CPU scheduling algorithms
- Assess CPU scheduling algorithms based on scheduling criteria
- Explain the issues related to multiprocessor and multicore scheduling
- Describe various real-time scheduling algorithms
- Describe the scheduling algorithms used in the  Linux  operating system



# Medium Term Scheduling

- The **medium term scheduler** controls the degree of multi-programming.
- It admits processes into memory to compete for the CPU
- Also called a **swapper** when it is used to control the **process mix** by removing and restoring processes
- Goal of medium term scheduler is to keep a good mix of processes in memory so that the CPU is always kept busy.

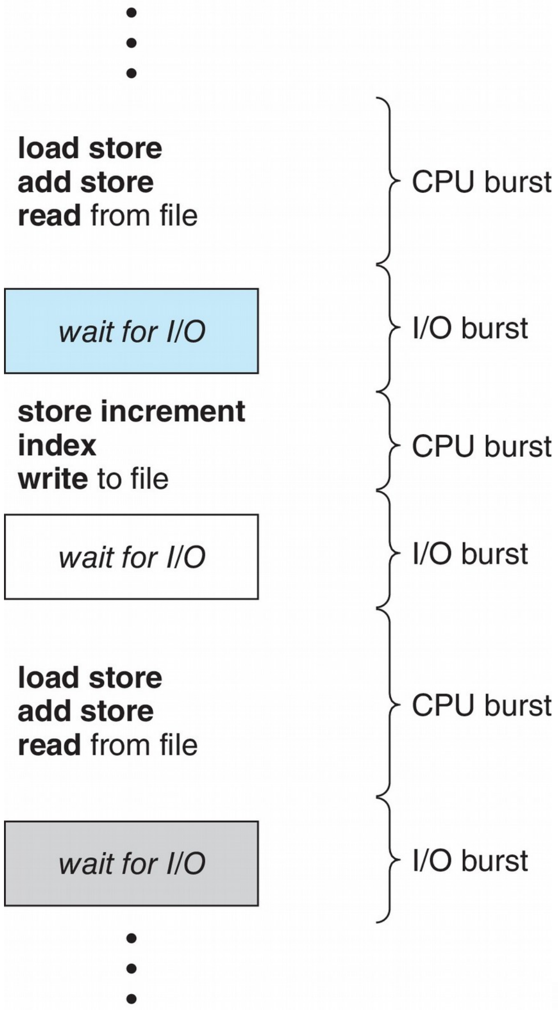


# Basic Problem

- Objective  of multi-programming is to have some process running at all times, to maximize CPU utilization.
- Need to keep many programs in memory
- Process executes on CPU until it executes some instruction for which it has to wait (e.g. I/O)
- Process is removed from CPU and another must be chosen to run
- This is purpose of CPU scheduler: which process should run?

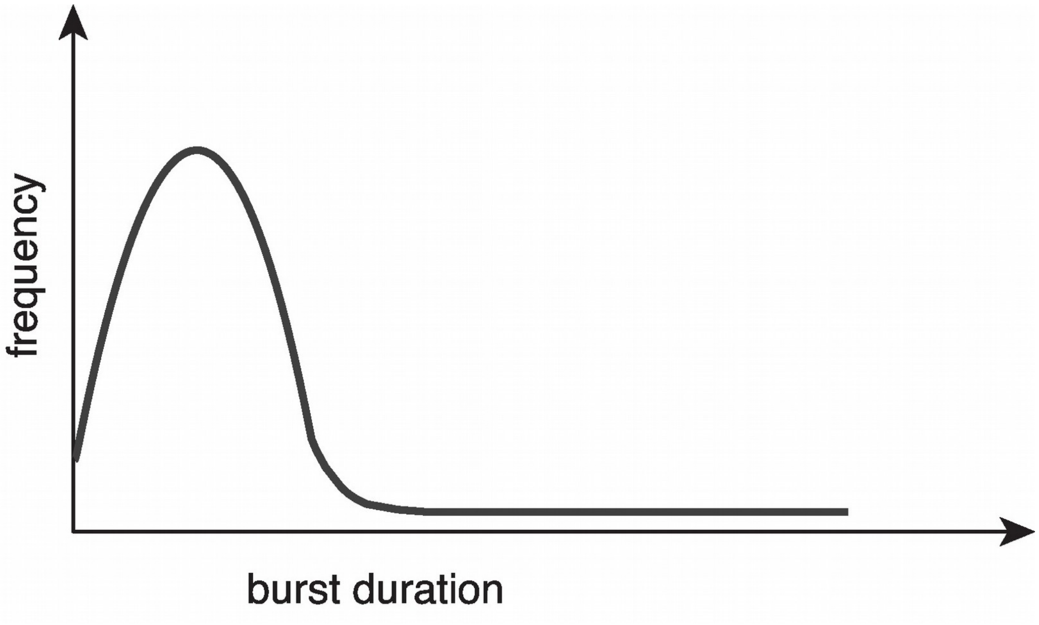
# Core Concept

- Every process has the same cycle of execution, called the **CPU–I/O Burst Cycle**
  - Process executes on CPU until it issues I/O, then waits for I/O to complete
  - The time on CPU is called a **CPU burst**
  - When it issues I/O request it is called an **I/O burst**
- It repeats this cycle over and over until it terminates.
- Lengths of CPU bursts affect decisions about scheduling.



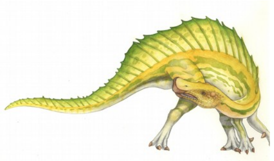
# Distribution of CPU Burst Times

- The duration of CPU bursts has been measured extensively.
  - Many burst times are short; few are long
  - Distribution is generally  exponential or hyperexponential,
  - An **I/O-bound** program has many short CPU bursts.
  - A **CPU-bound** program has a few long CPU bursts.



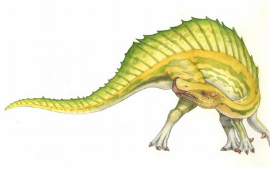
# CPU Scheduler

- When the running process is removed from CPU, the OS must choose a process to run. This is the role of the **CPU scheduler**.
- One or more queues of ***ready-to-run*** processes are kept in memory. These are called **ready queues**.
- The **CPU scheduler** selects from among the processes in ready queue, and allocates the a CPU core to one of them
  - Queue may be ordered in various ways – FIFO, priority queue, unordered, etc
  - Usually process control blocks or pointers to PCBs are kept in queues.



# Scheduling Decisions

- CPU scheduling decisions may take place when a process:
  1. Switches from running to waiting state (e.g. by issuing an I/O request)
  2. Switches from running to ready state  (e.g., an interrupt occurs)
  3. Switches from waiting to ready (e.g.,  I/O completes)
  4. Terminates
- Condition 3 is called a signal or release event.
- If scheduling only occurs under conditions 1 and 4 it is called **nonpreemptive scheduling**
- If it occurs under any of the conditions it is called **preemptive scheduling**

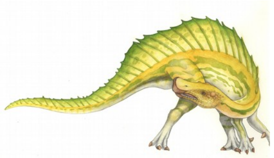


# Preemption

- Without preemption, process runs until it terminates or issues some request that causes it to wait such as I/O or other system call.
- Results in poor response time for other processes.
- **Preemptive scheduling** is the norm for modern operating systems.
- But preemptive scheduling causes  other problems:
  - Consider two processes that access  shared data
  - Consider preemption while in kernel mode – kernel data structures can become corrupted if kernel is not designed carefully
  - Consider interrupts occurring during crucial OS activities

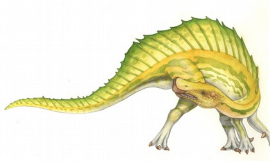
# Non-Preemptive Kernel

- During the processing of a system call, the kernel may be busy with an activity on behalf of a process.
- This can involve changing  a kernel data structure.
- If it is interrupted or preempted, the data structure can become corrupted.
- A non-preemptive kernel  waits for a system call to complete or for a process to block while waiting for I/O to complete to take place before doing a context switch.
- Makes the kernel structure  simple, since the kernel will not preempt a process while the kernel data structures are in an inconsistent state.
- But very inefficient and does not work for real-time operating systems.



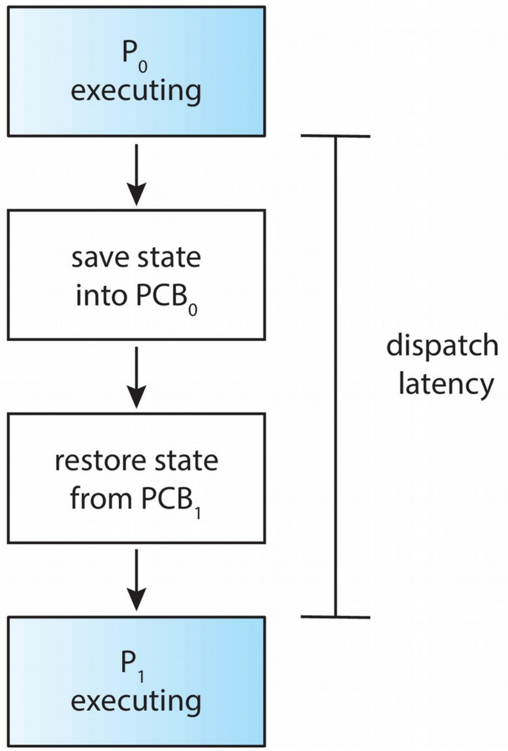
# Preemptive Kernels

- Kernel is allowed to be interrupted almost any time because they might be important and  cannot  be ignored by the kernel.
- The kernel needs to handle these interrupts otherwise, input might be lost or output overwritten.
- Solution: disable interrupts at entry and re-enable interrupts at exit of sections of code that handle them.
- Code sections must be very short.



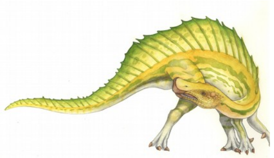
# Dispatcher

- **Dispatcher** is the component of the kernel that 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 (loading its PC)
- **Dispatch latency** – time it takes for the dispatcher to stop one process and start another running
- Need dispatcher to be very fast to minimize latency.



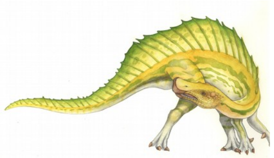
# CPU Scheduling Criteria

- **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, including wait time
- **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)



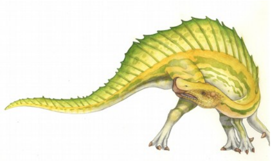
# Scheduling Algorithm Objectives

- Maximize  CPU utilization
- Maximize throughput
- Minimize turnaround time
- Minimize waiting time
- Minimize response time
- Cannot accomplish all, so it is an **optimization problem**.



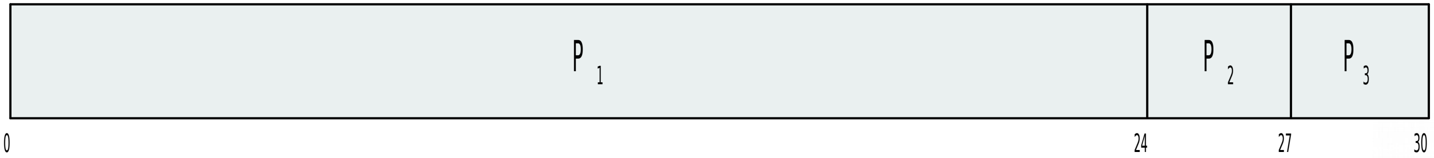
# Nuances of Optimization

- Do we optimize the maximum, minimum, or average measure of the property. For example,
  - do we optimize design so that maximum response time is minimal or average response time is minimal?
  - do we minimize maximum waiting time or average waiting time?
- In time-sharing systems, reliability is important, so minimizing variance in response time is often the objective rather than mean response time.
- For simple analysis, to assess scheduling algorithms, we can model processes by the lengths of CPU bursts and their average wait times.



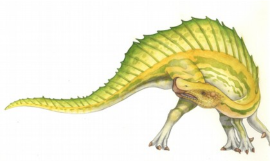
# Scheduling Algorithms

- We look at a few common scheduling algorithms.
- Each is classified as preemptive or non-preemptive.
- An algorithm that is non-preemptive is only used for non-preemptive scheduling; a preemptive algorithm is used for preemptive scheduling.
- We will use Gantt charts to analyze the algorithms.
- A Gantt chart is a horizontal bar chart in which the x-axis is time. A Gantt chart represents the order in which processes are scheduled and how much time each spends in the CPU.
- Example:
  - The chart below represents 3 processes, P1, P2, P3, scheduled at times 0, 26, and 27 respectively. P1 ran 26 units, P2, 1 unit, P3, 3 units.



# First-Come-First-Served Scheduling

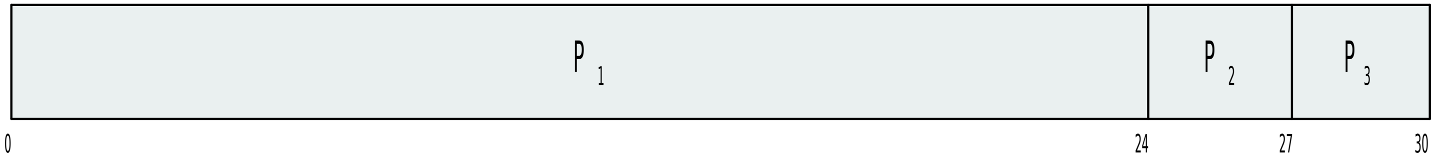
- The first is **first-come-first-served** (**FCFS**) which is a non-preemptive algorithm.
- In **first-come-first-served scheduling**, processes arrive and are placed at the rear of a FIFO queue.
- The process at the front of the queue is scheduled next.
- If a process issues a wait, it is placed at the rear of the queue again.



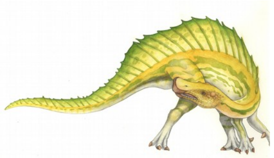
# FCFS Example

| 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

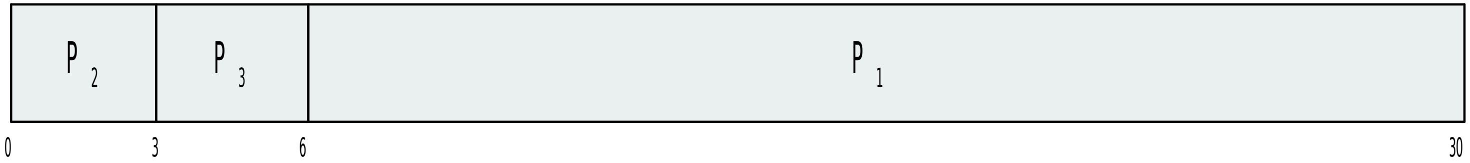


# FCFS Example (Cont.)

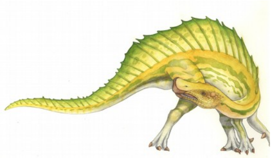
Suppose that the processes arrive in the order:

$P_2$ , $P_3$ , $P_1$

- 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
- Performance is affected by order of arrivals

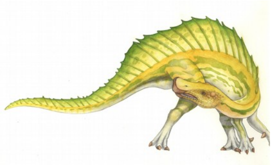


# Convoy Effect

- **Convoy effect** - short process behind long process
  - Consider one CPU-bound and many I/O-bound processes
  - CPU bound process runs, takes long time and short I/O bound processes wait.
  - They all run quickly, issue I/O requests and wait again for long CPU bound process. Their I/O is satisfied and they all move to ready queue, but they have to wait and wait and wait...
- I/O devices are poorly utilized and response times are poor.

# 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

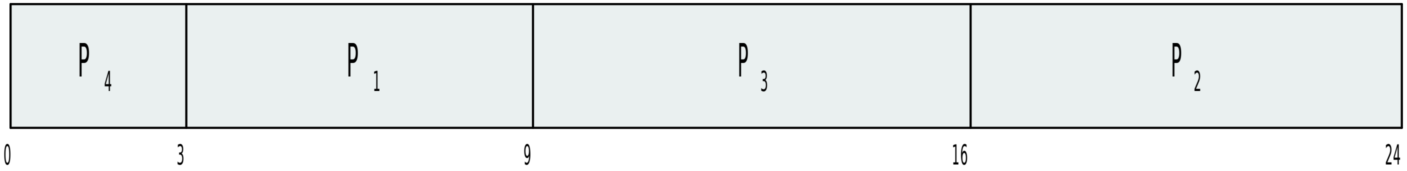


# Example of SJF

- Assume all processes arrive at time 0 and that CPU burst times are known and as shown below.

| Process | Burst Time |
|---|---|
| $P_1$ | 6 |
| $P_2$ | 8 |
| $P_3$ | 7 |
| $P_4$ | 3 |

- SJF scheduling chart



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

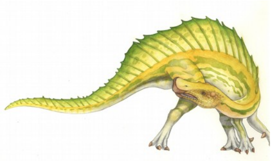
# Predicting Length of Next CPU Burst

- Can only estimate the length of next burst – should be similar to the previous one, or the the most recent ones.
  - Then pick process with shortest predicted next CPU burst

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

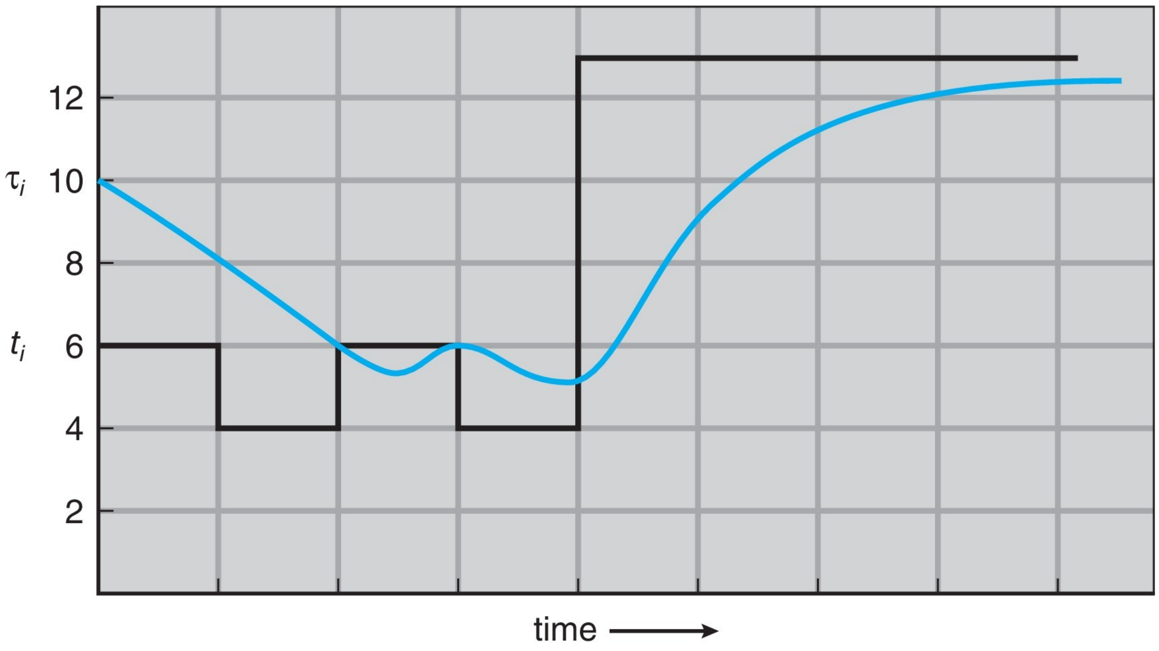
  1. $t_n = \text{actual length of } n^{th} \text{ CPU burst}$
  2. $\tau_{n+1} = \text{ predicted value for the next CPU burst}$
  3. $\text{For any } \alpha,\ 0 \le \alpha \le 1$
  4. $\text{define:} \qquad \tau_{n=1} = \alpha t_n + (1-\alpha)\tau_n.$

- Initally, first guess τ0 is any value.
- When α=0, guess never changes. When α = 1, guess is always last burst length. (upcoming slides illustrate ) Typically α is set to ½
- Preemptive version called **shortest-remaining-time-first**



# Prediction of the Length of the Next CPU Burst

(This graph is for α=0.5.)



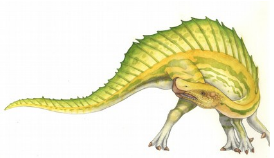
| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| CPU burst ($t_i$) | | 6 | 4 | 6 | 4 | 13 | 13 | 13 | . . . |
| "guess" ($\tau_i$) | 10 | 8 | 6 | 6 | 5 | 9 | 11 | 12 | . . . |

# Examples of Exponential Averaging

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

$$
\begin{aligned}
\tau_{n+1} = \alpha\, t_n &+ (1 - \alpha)\alpha\, t_{n-1} + \dots \\
&+ (1 - \alpha)^j \alpha\, t_{n-j} + \dots \\
&+ (1 - \alpha)^{n+1} \tau_0
\end{aligned}
$$

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



# Example of Shortest-Remaining-Time-First

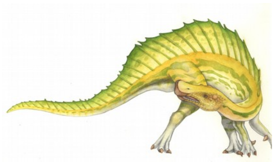
- When SJF is preemptive it is called **Shortest Remaining Time First,** so **SRTF = *Preemptive* SJF**
- A process runs, gets preempted and is moved to back of ready queue. Its new predicted burst time is the previous one minus the time it just spent on the CPU.
- This is why it is shortest remaining time first, because predicted burst is what is remaining.

# Example of SRTF

- 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 |

- To create a Gantt Chart, you need to simulate the scheduler.
- Scheduler needs to run when a new process arrives, and it picks shortest burst time remaining process.



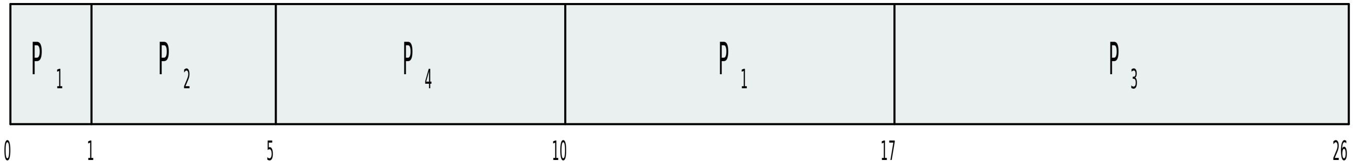
# SRTF Worked Example

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

- Time 0: P1 runs for 1 msec. P2 arrives and has shorter time so it is scheduled. P1 has 7 msec remaining time now.
- Time 1: P2 runs. P3 and P4 arrive at times 2 and 3, but both have longer times, so P2 stays on CPU until it finishes at time 5 (1+4)
- Time 5:  P4 has shortest remaining time (5 msec) so it gets CPU and runs to finish at time 10.
- Time 10: P1 has shorter time than P3, so P1 runs and finishes at time 17, then P3 runs.

# Example of Shortest-Remaining-Time-First

- SRTF Gantt Chart for this set of processes:

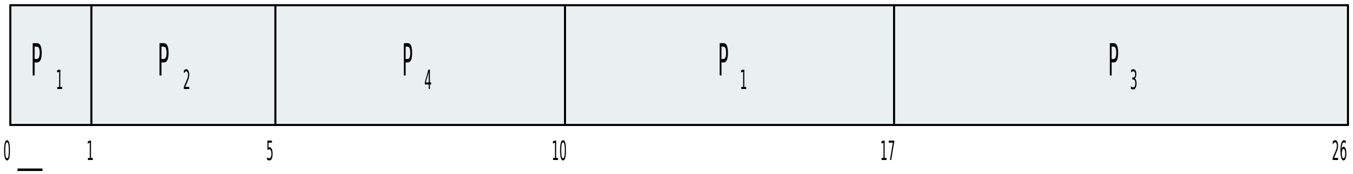


- What is the average waiting time?

# Waiting Time for SRTF

- Waiting time for each process is

  *(time it finishes) – (burst length) – (arrival time)*



| Process | Arrival Time | Burst Time | End Time |
|---|---|---|---|
| $P_1$ | 0 | 8 | 17 |
| $P_2$ | 1 | 4 | 5 |
| $P_3$ | 2 | 9 | 26 |
| $P_4$ | 3 | 5 | 10 |

- Average waiting time =
  - ((17-8-0) +(5-4-1) + (26-9-2) + (10-5-3))/4 =
  - (9+0+15+2)/4 = 6.5

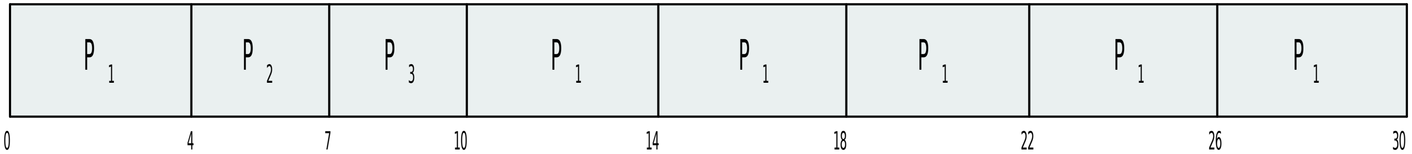
# Round Robin (RR)

- 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. ***This is a scheduled event.***
- 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 to run again.
- Might be less if processes issue I/O requests.
- Timer interrupts at end of quantum to schedule next process
- Performance
  - *q* large ⇒ FIFO
  - *q* small ⇒ *q* must be large with respect to context switch, otherwise overhead is too high

# Example of RR with Time Quantum = 4

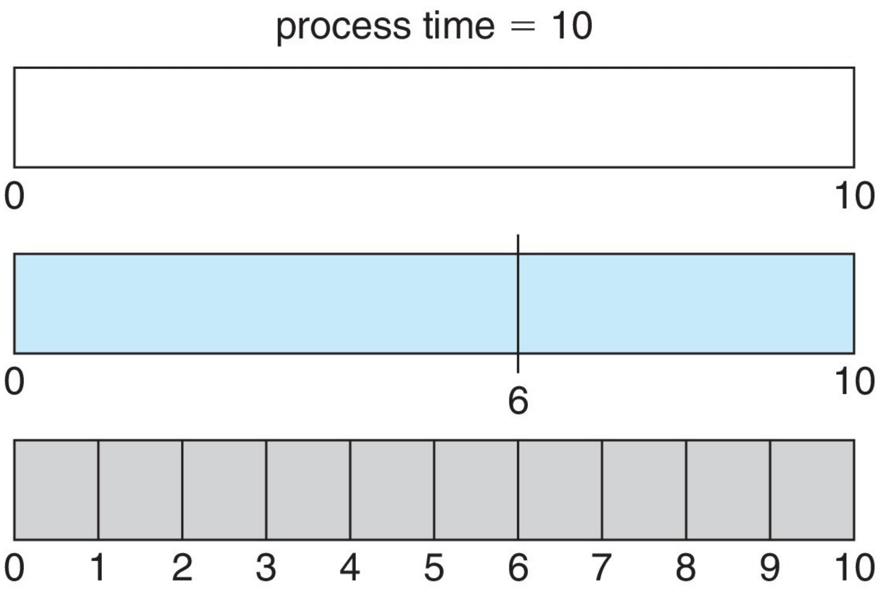
| Process | Burst Time |
|---|---|
| $P_1$ | 24 |
| $P_2$ | 3 |
| $P_3$ | 3 |

- The Gantt chart is:



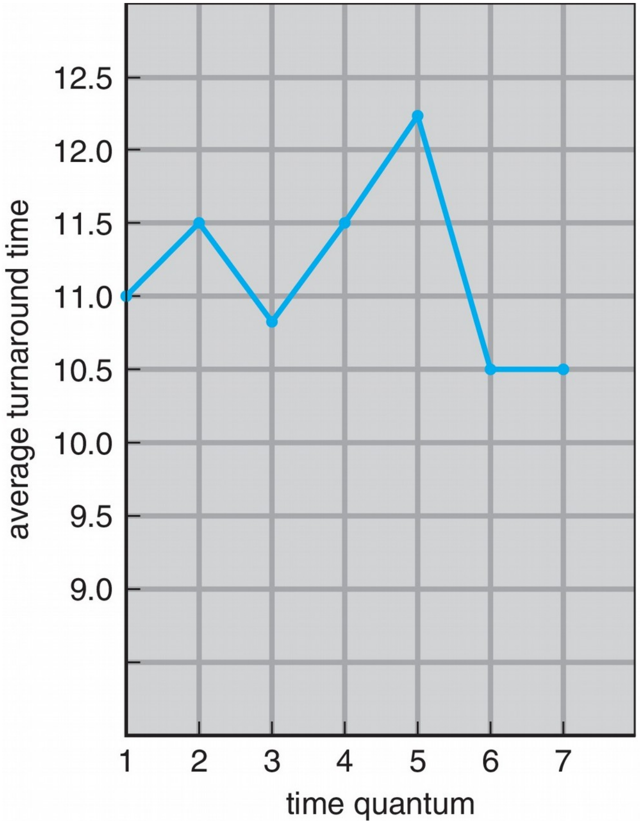
- Turnaround time average here = ( 30 + 10 + 7 )/3 = 15.7
- If using SJF, turnaround would be (30 + 6 + 3)/3 = 13
- Typically, higher average turnaround than SJF, but better ***response time***
- q should be large compared to context switch time; q usually 10ms to 100ms, context switch < 10 usec

# Time Quantum and Context Switch Time



| quantum | context switches |
|---|---|
| 12 | 0 |
| 6 | 1 |
| 1 | 9 |

# Turnaround Time Varies With The Time Quantum



| process | time |
|---|---|
| $P_1$ | 6 |
| $P_2$ | 3 |
| $P_3$ | 1 |
| $P_4$ | 7 |

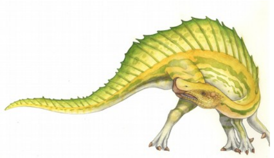
**Rule of thumb for choosing quantum size:**

*80% of CPU bursts should be shorter than q*

| | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $P_1$ | $P_2$ | $P_3$ | $P_4$ | $P_1$ | $P_2$ | $P_4$ | $P_1$ | $P_2$ | $P_4$ | $P_1$ | $P_4$ | $P_1$ | $P_4$ | $P_1$ | $P_4$ | $P_4$ | |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 |

# Priority Scheduling

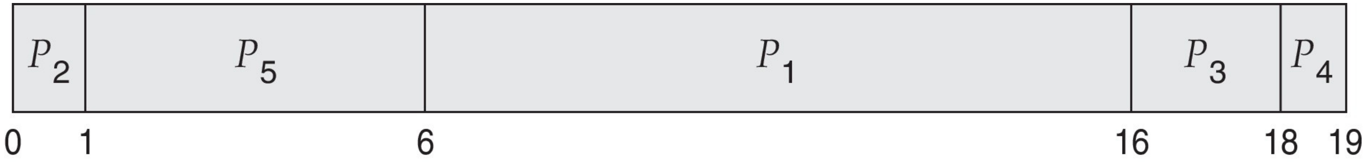
- A priority number (integer) is associated with each process
- The CPU is allocated to the process with the highest priority (smallest integer ≡ highest priority)
- Can be either
  - Preemptive   - high priority arriving process preempts lower one
  - Nonpreemptive
- SJF is priority scheduling where priority is the inverse of predicted next CPU burst time
- Problem ≡ **Starvation** – (also called **indefinite blocking**)
  - – low priority processes may never execute
- Solution ≡ **Aging** – as time progresses increase the priority of the process



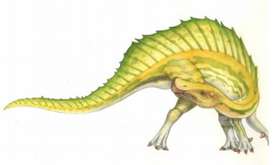
# Example of Priority Scheduling

| 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

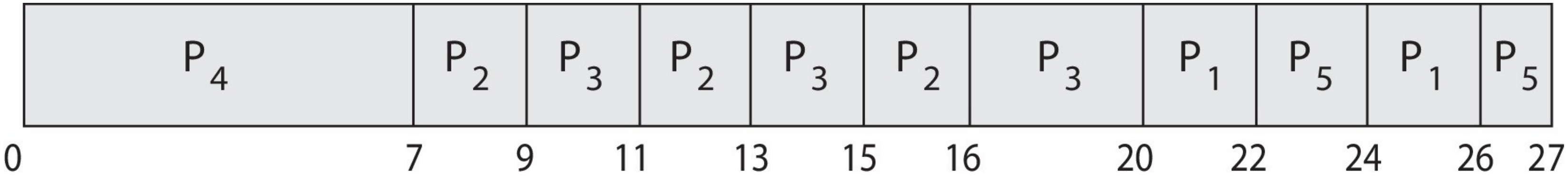


# Priority Scheduling w/ Round-Robin

| Process | Burst Time | Priority |
|---|---|---|
| $P_1$ | 4 | 3 |
| $P_2$ | 5 | 2 |
| $P_3$ | 8 | 2 |
| $P_4$ | 7 | 1 |
| $P_5$ | 3 | 3 |

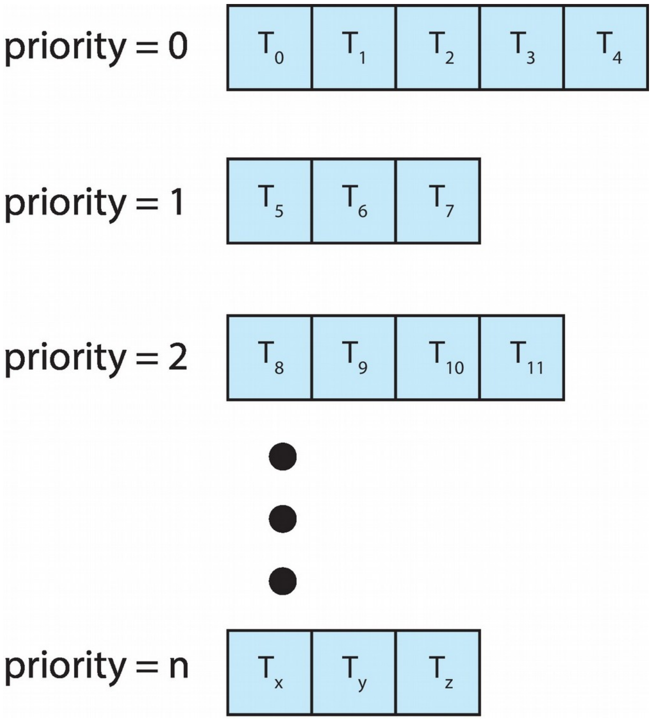
- Run the process with the highest priority. Processes with the same priority run round-robin

- Gantt Chart with 2 ms time quantum



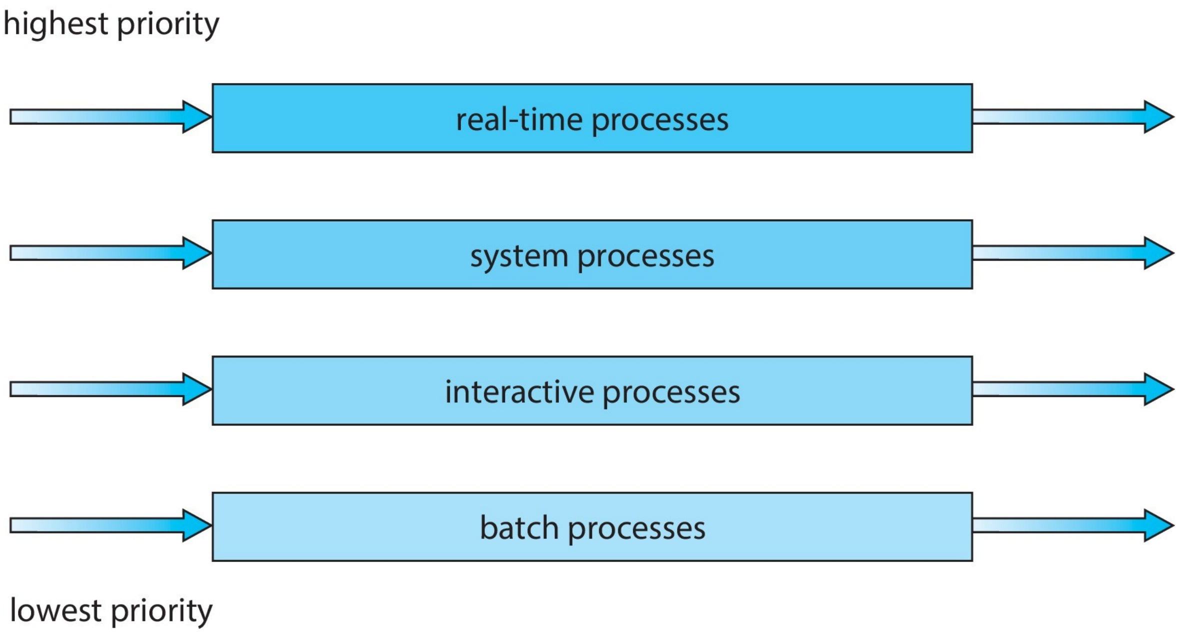
# Multilevel Queue

- With priority scheduling, have separate queues for each priority.
- Schedule the process in the highest-priority queue!



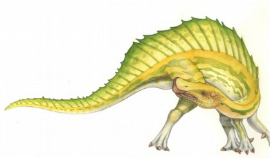
# Multilevel Queues

- Prioritization based upon process type



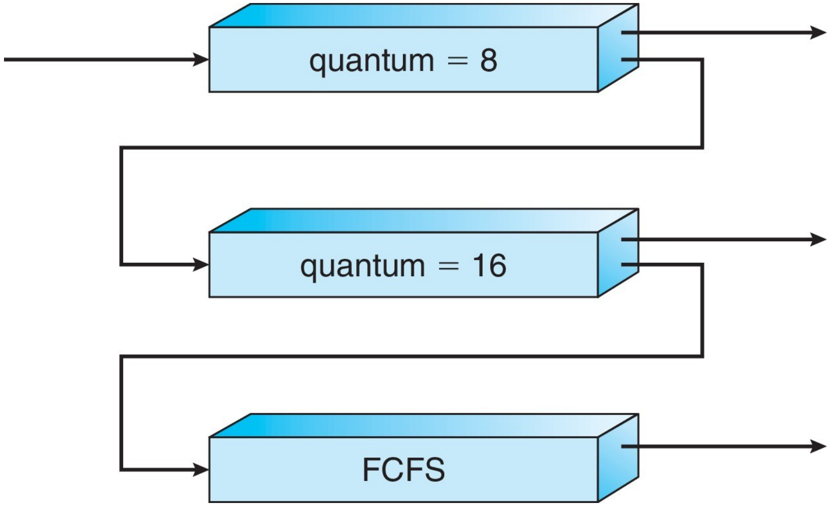
# Multilevel Feedback Queue

- 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



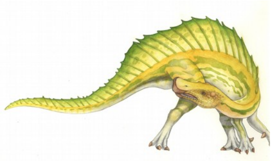
- Scheduling
  - A new job enters queue $Q_0$ which is served RR with q= 8
    - When it gains CPU, job receives 8 milliseconds
    - If it does not finish in 8 milliseconds, job is moved to queue $Q_1$
  - At $Q_1$ job is again served RR with q=16 and receives 16 additional milliseconds
    - If it still does not complete, it is preempted and moved to queue $Q_2$

# Thread Scheduling: Contention Scope

- User-level and kernel-level threads are scheduled differently.
- When threads are supported by the kernel, threads are scheduled, not processes.
- Remember – user threads are mapped to kernel threads.
- Question – do all threads of all processes compete for CPUs equally, or do the threads of each process compete against each other for the CPU, with each process being given a share of CPU time?
  - This is the question of thread **contention scope** – which threads does a thread contend with (i.e., compete against) for the CPU?
  - Two types: **process-contention** and **system-contention** scopes.

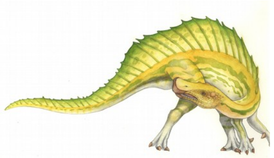
# Process Contention Scope

- With the many-to-one and many-to-many models, the thread library schedules user-level threads to run on LWPs.
  - Implies that all threads in a single process get scheduled onto fixed set of LWPs, which in turn get scheduled onto CPUs.
  - So threads within a process compete for LWPs.
  - Known as **process-contention scope** (**PCS**) since scheduling competition is within the process.
  - Typically done via priority set by programmer



# System Contention Scope

- Kernel threads are scheduled onto available CPUs, so all kernel threads compete with each other across the entire system.
- This is called **system-contention scope** (**SCS**) – competition among all threads in the system.
- The one-to-one thread model (used by Windows and Linux) uses SCS only.



# Pthread Scheduling

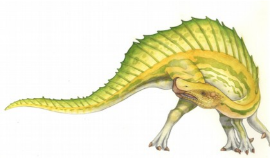
- The `Pthread` API allows specifying either PCS or SCS during thread creation
  - `PTHREAD_SCOPE_PROCESS` schedules threads using PCS scheduling
  - `PTHREAD_SCOPE_SYSTEM` schedules threads using SCS scheduling
- Although the API allows the choice, can be overridden by OS – Linux and macOS only allow `PTHREAD_SCOPE_SYSTEM`

# Pthread Scheduling API Example

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5
int main(int argc, char *argv[]) {
   int i, scope;
   pthread_t tid[NUM THREADS];
   pthread_attr_t attr;
   pthread_attr_init(&attr); /* apply  default attributes */
   /* inquire about the current scope */
   if (pthread_attr_getscope(&attr, &scope) != 0)
       fprintf(stderr, "Unable to get scheduling scope\n");
   else {
      if (scope == PTHREAD_SCOPE_PROCESS)
          printf("thread has PTHREAD_SCOPE_PROCESS");
      else if (scope == PTHREAD_SCOPE_SYSTEM)
          printf("thread has PTHREAD_SCOPE_SYSTEM");
      else
          fprintf(stderr, "Illegal scope value.\n");
   }
```

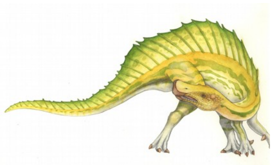
# Pthread Scheduling API

```
    /* set the scheduling algorithm to PCS */
   pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);
   /* create the threads */
   for (i = 0; i < NUM_THREADS; i++)
       pthread_create(&tid[i],&attr,runner,NULL);
   /* now join on each thread */
   for (i = 0; i < NUM_THREADS; i++)
       pthread_join(tid[i], NULL);
}
/* Each thread will begin control in this function */
void *runner(void *param)
{
    /* do some work ... */
    pthread_exit(0);
}
```



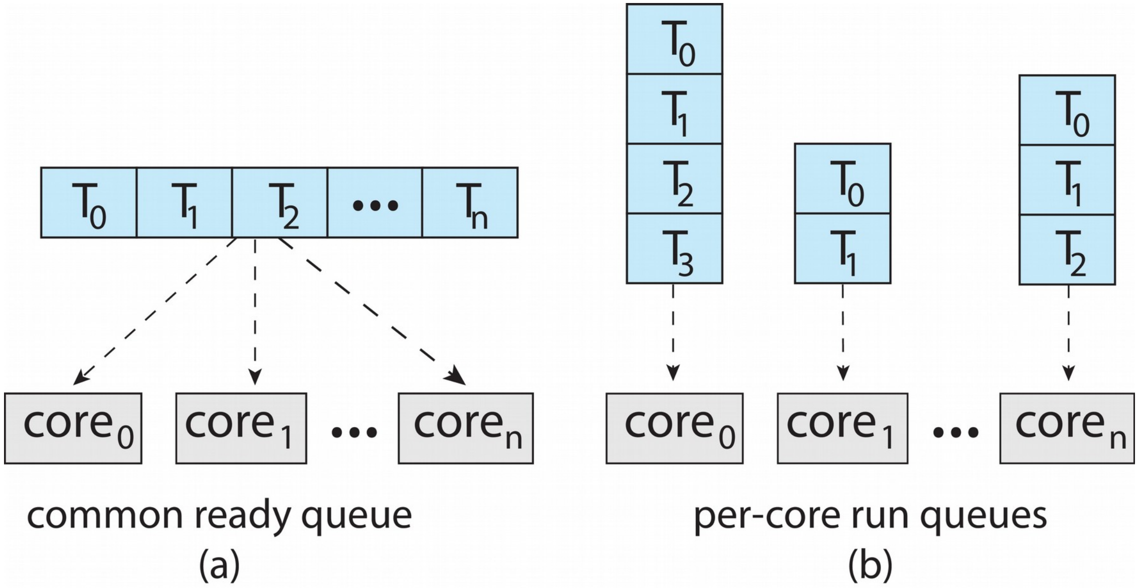
# Multiple-Processor Scheduling

- CPU scheduling more complex when multiple CPUs are available
- A multiprocessor may be any one of the following architectures:
  - Multi-core processor
  - Multi-threaded cores
  - A multi-core processor with multi-threaded cores
  - NUMA systems
  - Heterogeneous multiprocessing



# Multiple-Processor Scheduling (2)

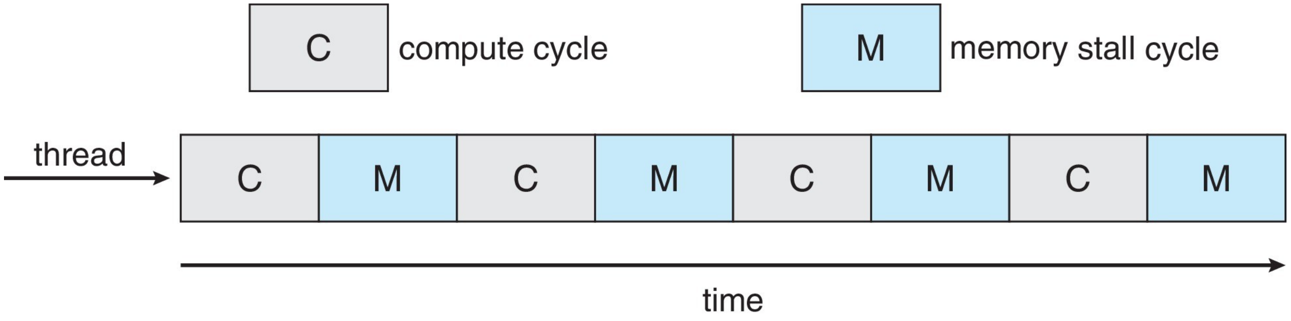
- Symmetric multiprocessing (SMP) is where each processor is self scheduling. Two scheduling models:
  - All threads may be in a shared ready queue (a)
  - Each processor may have its own private queue of threads (b)



  - Shared queue has race condition and needs synchronization/locking

# Multi-core Processors

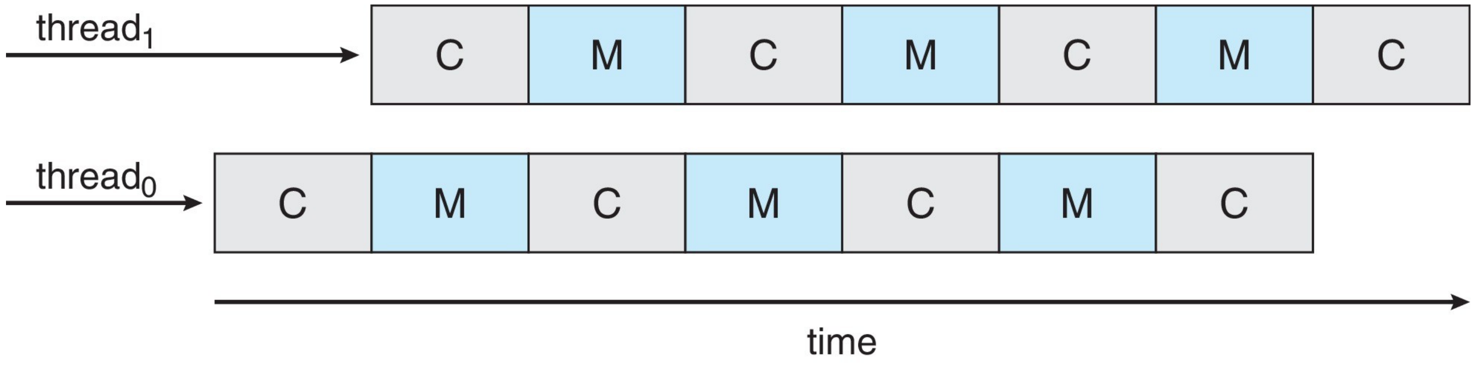
- Multiple processor cores on same physical chip
- Faster and consumes less power
- Multiple threads per core also growing
  - Sometimes called hyperthreading
  - Takes advantage of memory stall to make progress on another thread while memory retrieve happens



# Multi-threaded Multi-core System

Each core has > 1 hardware threads.

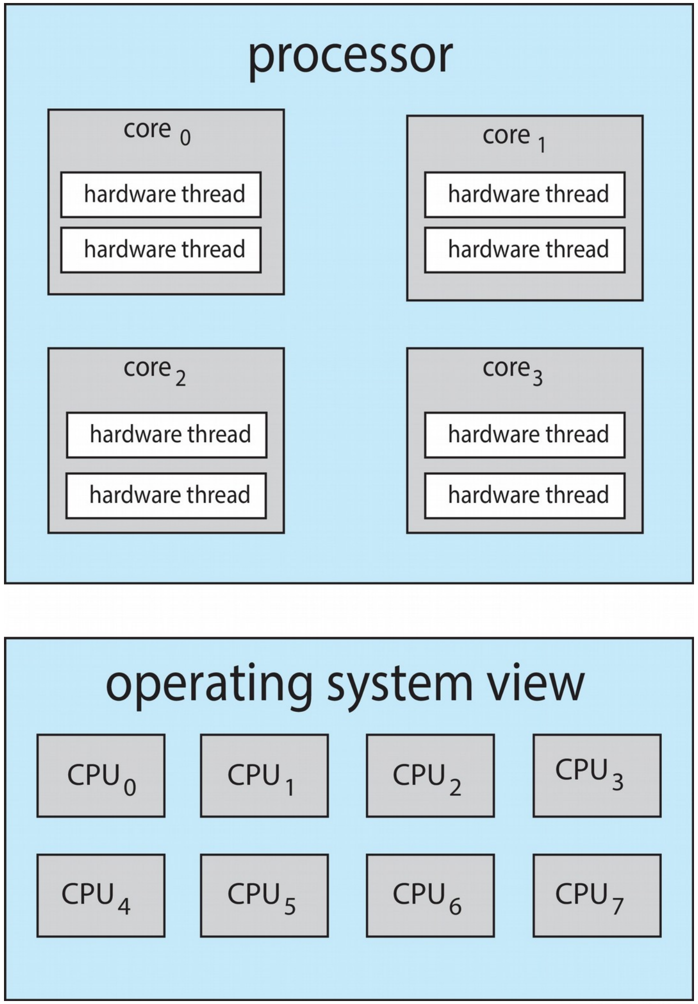
If one thread has a memory stall, switch to another thread!



# Multi-threaded Multi-core System

- **Chip-multithreading** (CMT) assigns each core multiple hardware threads. (Intel refers to this as **hyperthreading**.)

- On a quad-core system with 2 hardware threads per core, the operating system sees 8 logical processors.

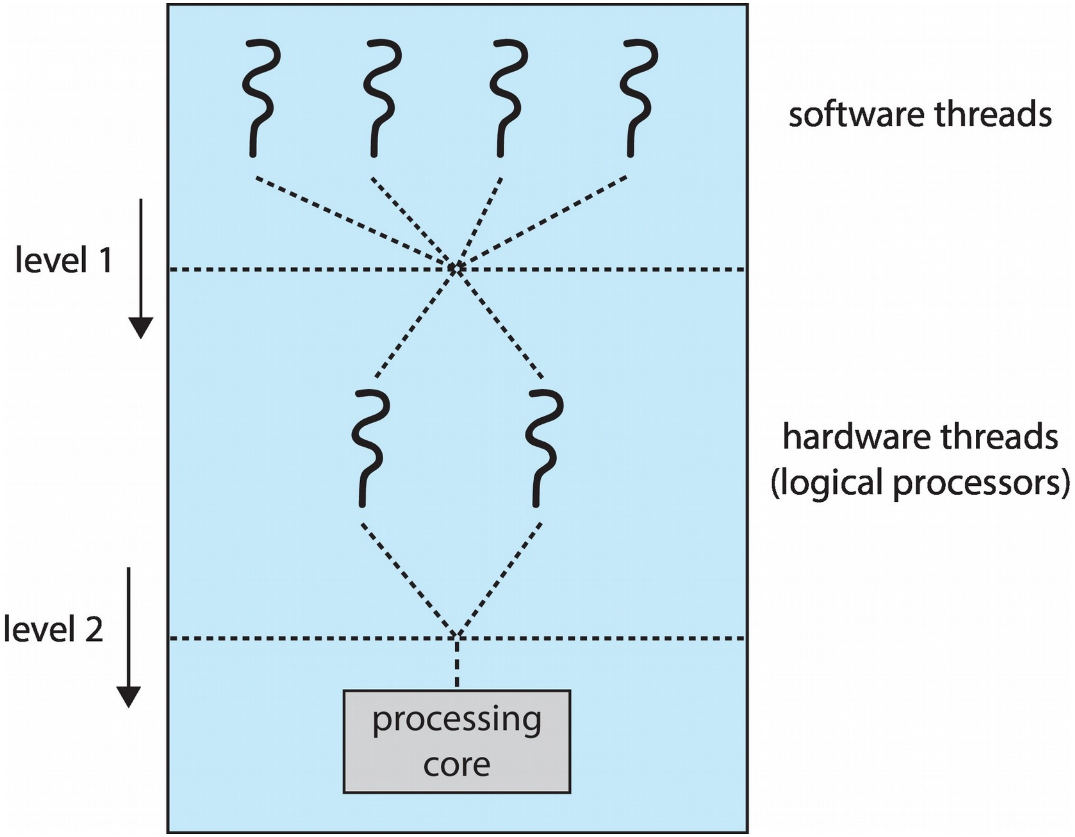


# Multi-threaded Multi-core System

- Two levels of scheduling:

1. The operating system deciding which software thread to run on a logical CPU

2. How each core decides which hardware thread to run on the physical core.

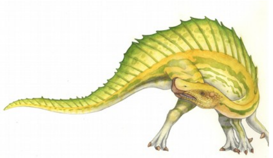


# Multiple-Processor Scheduling – Load Balancing

- If SMP, need to keep all CPUs loaded for efficiency
- **Load balancing** attempts to keep workload evenly distributed
- Two paradigms:
  - **Push migration** – periodic task checks load on each processor, and if found pushes task from overloaded CPU to other CPUs
  - **Pull migration** – idle processors pull waiting task from busy processor
  - Both can be used in combination (Linux CFS does)

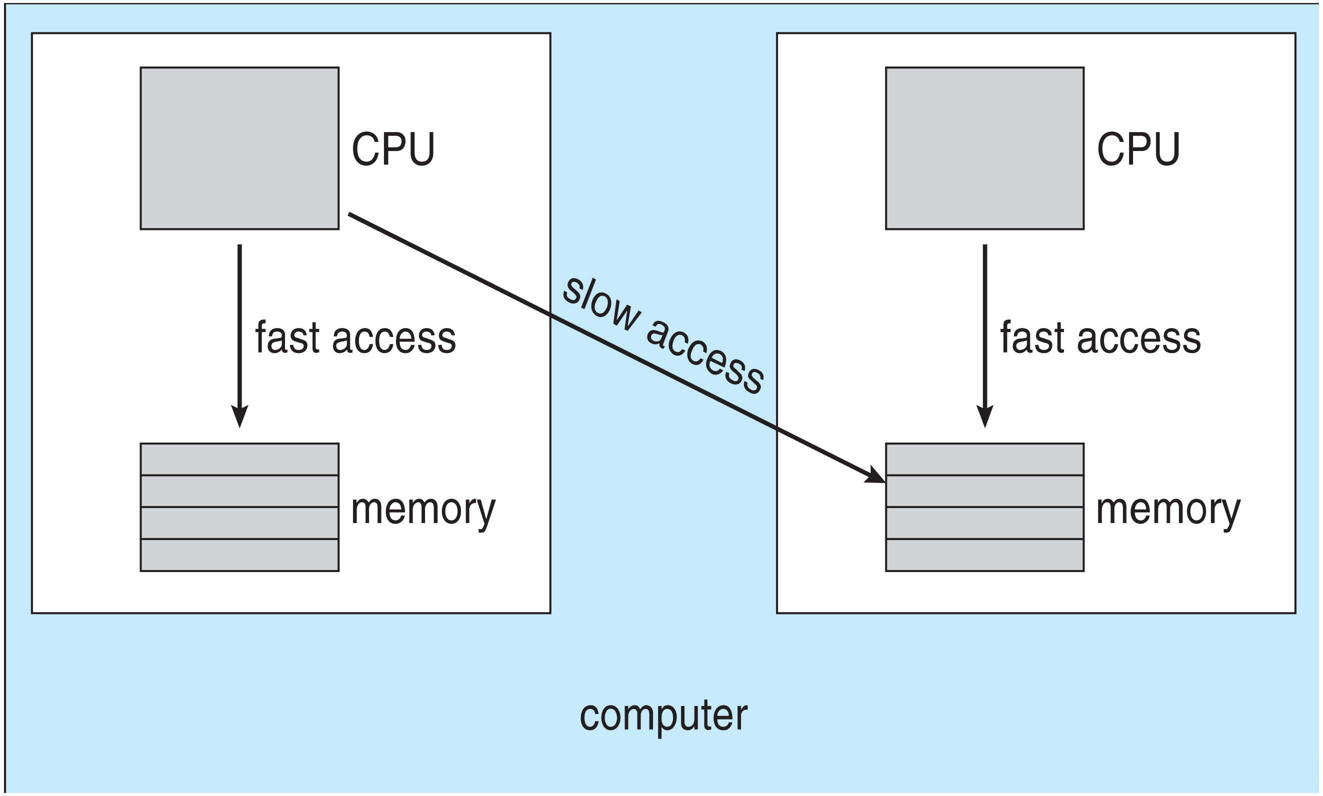
# Multiple-Processor Scheduling – Processor Affinity

- When a thread has been running on one processor, the cache contents of that processor stores the memory accesses by that thread.
- We refer to this as a thread having affinity for a processor (i.e. “processor affinity”)
- Load balancing may affect processor affinity as a thread may be moved from one processor to another to balance loads, yet that thread loses the contents of what it had in the cache of the processor it was moved off of.
- **Soft affinity** – the operating system attempts to keep a thread running on the same processor, but no guarantees.
- **Hard affinity** – allows a thread to specify a set of processors it may run on.



# NUMA and CPU Scheduling

If the operating system is **NUMA-aware**, it will assign memory closes to the CPU the thread is running on.



# Real-Time CPU Scheduling

- Can present obvious challenges
- **Soft real-time systems** – Critical real-time tasks have the highest priority, but no guarantee as to when tasks will be scheduled
- **Hard real-time systems** – task must be serviced by its deadline
- We will not cover real-time scheduling in this class

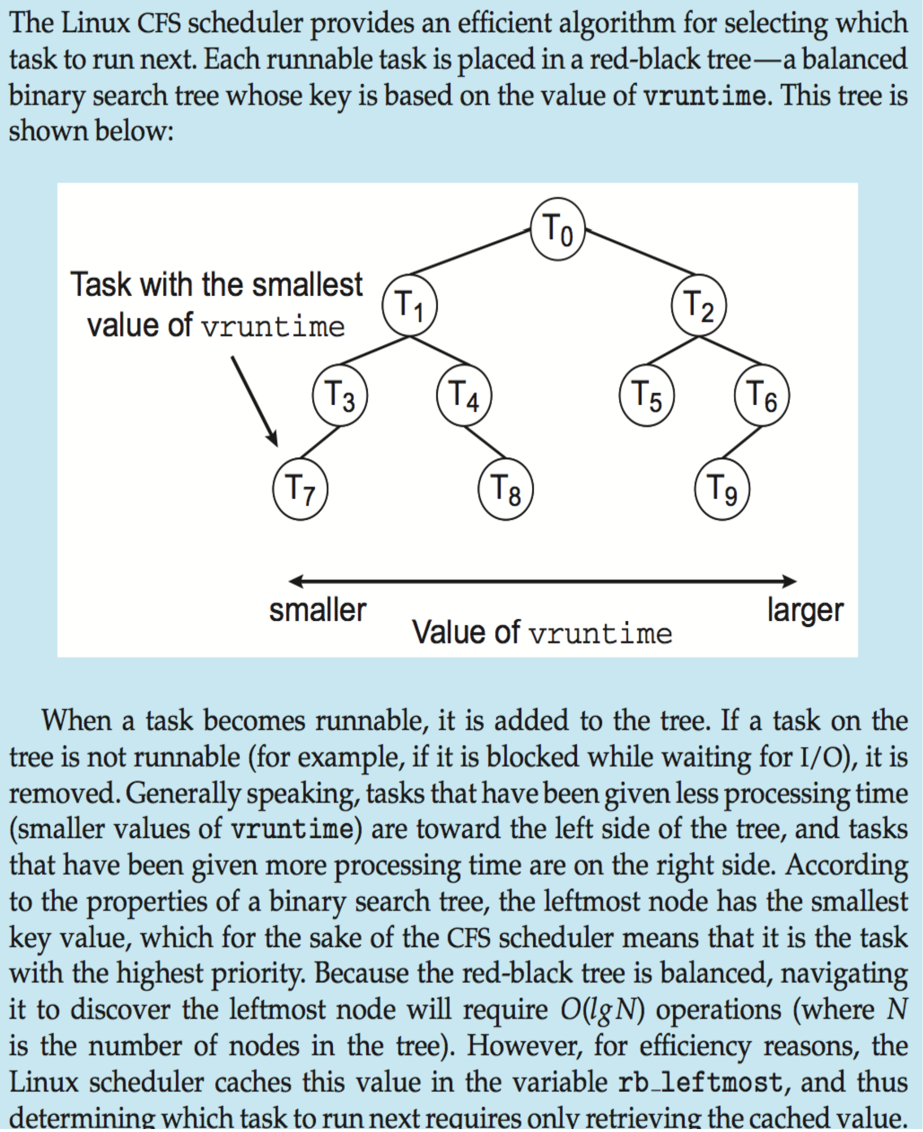
# Linux Scheduling Through Version 2.5

- 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
  - Map into  global priority with numerically lower values indicating higher priority
  - Higher priority gets larger q
  - Task run-able as long as time left in time slice (**active**)
  - If no time left (**expired**), not run-able until all other tasks use their slices
  - All run-able tasks tracked in per-CPU **runqueue** data structure
    - Two priority arrays (active, expired)
    - Tasks indexed by priority
    - When no more active, arrays are exchanged
  - 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
  - Rather than quantum based on fixed time allotments, based on proportion of CPU time
  - 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
  - Calculates **target latency** – interval of time during which task should run at least once
  - Target latency can increase if say number of active tasks increases
- CFS scheduler maintains per task **virtual run time** in variable **`vruntime`**
  - Associated with decay factor based on priority of task – lower priority is higher decay rate
  - Normal default priority yields virtual run time = actual run time
- To decide next task to run, scheduler picks task with lowest virtual run time

# CFS Performance



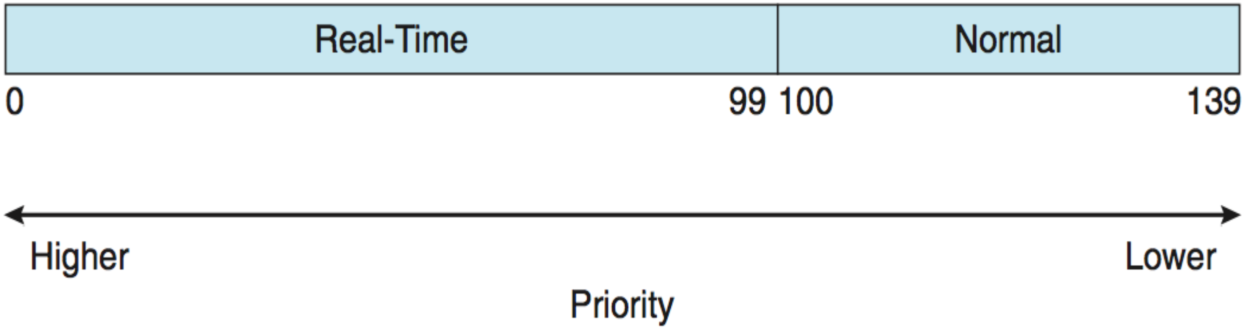
The Linux CFS scheduler provides an efficient algorithm for selecting which task to run next. Each runnable task is placed in a red-black tree—a balanced binary search tree whose key is based on the value of **vruntime**. This tree is shown below:

Task with the smallest value of vruntime
$T_0$
$T_1$
$T_2$
$T_3$
$T_4$
$T_5$
$T_6$
$T_7$
$T_8$
$T_9$
smaller
larger
Value of vruntime

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 $O(lgN)$ 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.

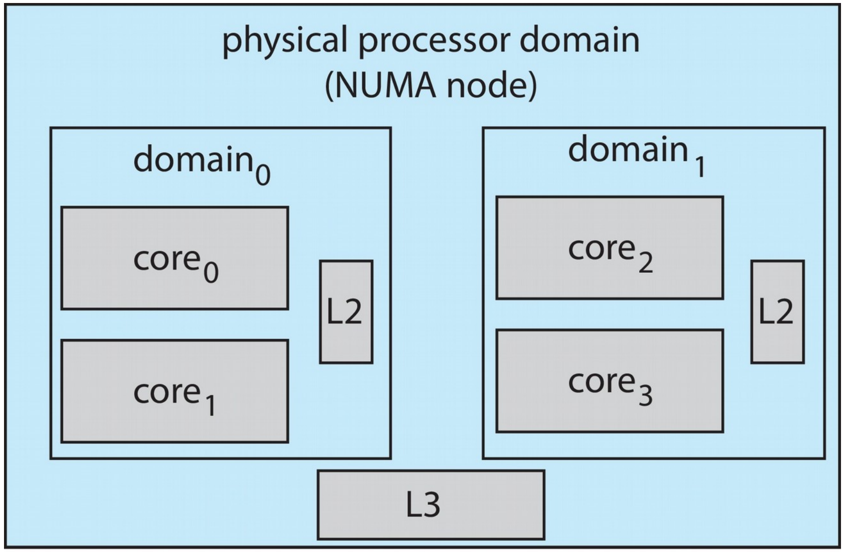
# Linux Scheduling (Cont.)

- 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



# Linux Scheduling (Cont.)

- Linux supports load balancing, but is also NUMA-aware.
- **Scheduling domain** is a set of CPU cores that can be balanced against one another.
- Domains are organized by what they share (i.e. cache memory.) Goal is to keep threads from migrating between domains.


physical processor domain
(NUMA node)
domain$_0$
core$_0$
core$_1$
L2
domain$_1$
core$_2$
core$_3$
L2
L3