Chapter 5: CPU Scheduling

Chapter 5: Outline

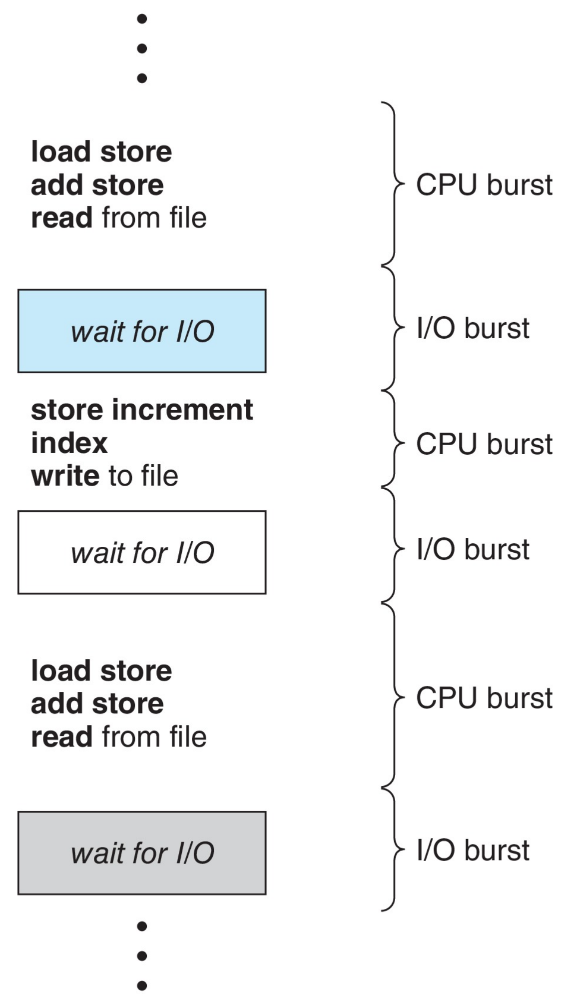
- Basic Concepts
- Scheduling Criteria
- Scheduling Algorithms
- Thread Scheduling
- Multi-Processor Scheduling
- Real-Time CPU Scheduling
- Operating Systems Examples
- Algorithm Evaluation

Objectives

- 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 **Windows**, **Linux**, and **Solaris** operating systems
- Apply *modeling* and *simulations* to evaluate CPU scheduling algorithms
- *Design a program* that implements several different CPU scheduling algorithms

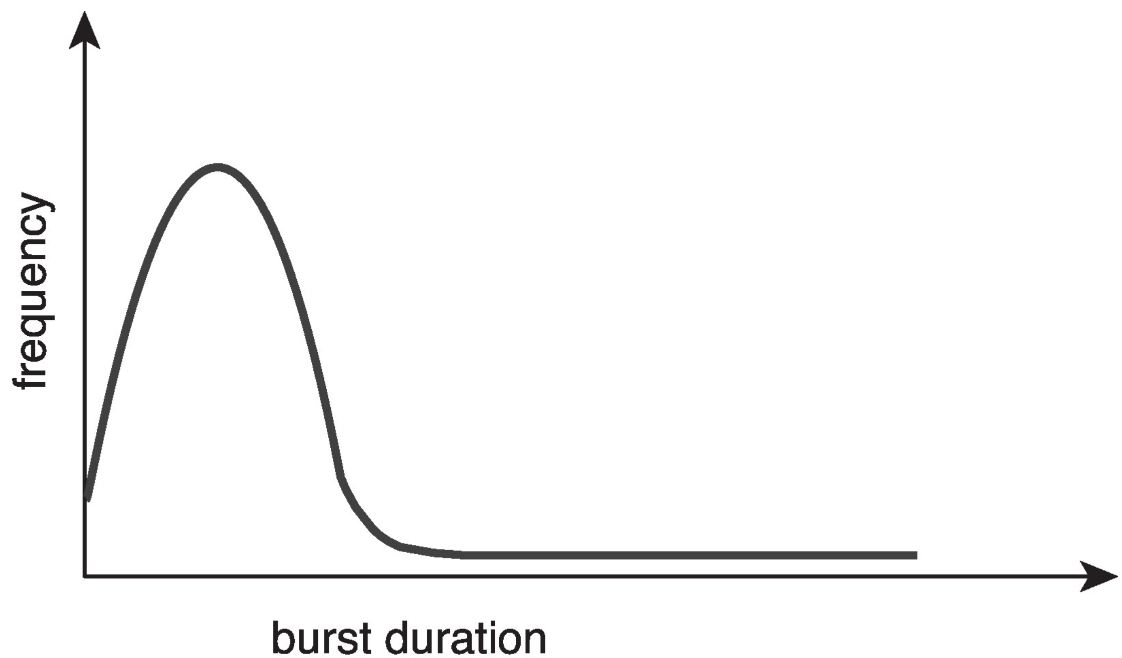
Basic Concepts

- Almost all computer resources are *scheduled* before use
- 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

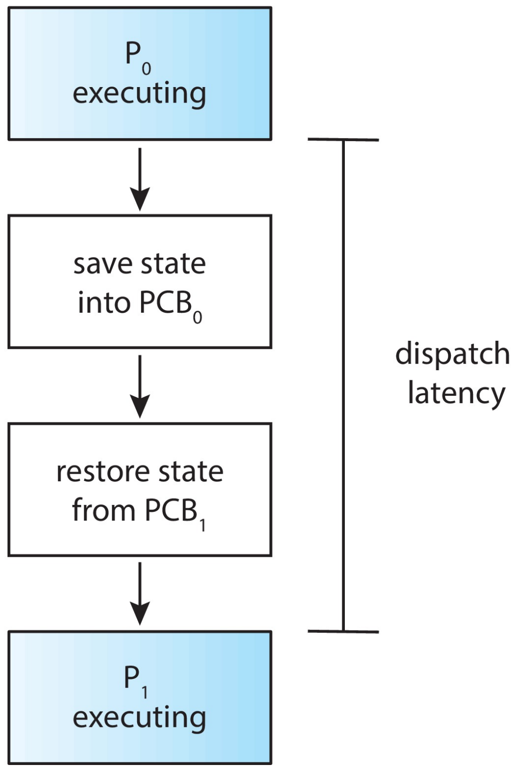
- Generally, frequency curve shows
  - Large number of *short bursts*
  - Small number of *longer bursts*



CPU Scheduler

- The *CPU scheduler* selects one process from among the processes in *ready queue*, and allocates the CPU core to it
  - Queue may be ordered in various ways: FIFO, priority, tree, linked list
- *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*
  - No choice in terms of scheduling
- All other scheduling is *preemptive*, and can result in *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
    - The number of context switches can be obtained by using the **#vmstat** command or the **/proc** file system for a given process
  - switching to user mode
  - jumping to the proper location in the user program to resume that program
- **Dispatch latency** – time it takes for the dispatcher to stop one process and start another running

#vmstat

Scheduling Criteria

- **CPU utilization** – keep the CPU as busy as possible
- **Throughput** – number 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 spends 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 outputting the response (for time-sharing environment or in an interactive system)

#top

Scheduling Algorithm Optimization Criteria

- Max *CPU utilization*
- Max *Throughput*
- Min *Turnaround time*
- Min *Waiting time*
- Min *Response time*

- In most cases, it is necessary to *optimize the average measure*
- For interactive systems (such as a PC desktop or laptop system), it is more important to *minimize the variance* in the response time

Note: *For next examples of the comparison of various CPU-scheduling algorithms*

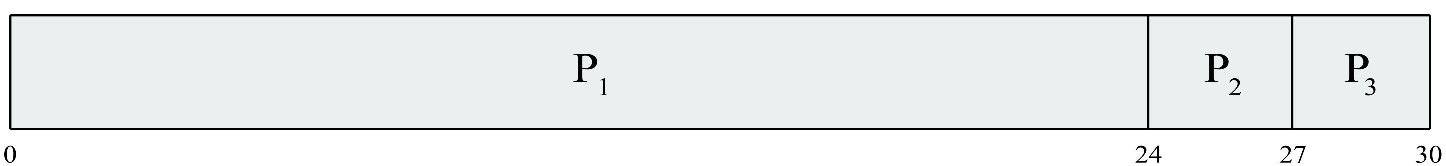
- Consider only *one CPU burst* (in milliseconds) per process
- The measure of comparison: **average waiting time**

First-Come, First-Served (FCFS) Scheduling

- **Motivation**: for simplicity, consider FIFO-like policy

Process	Burst Time (ms)
P_1	24
P_2	3
P_3	3

- Suppose that the processes arrive at time *0* 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: *P2 , P3 , P1*
- The *Gantt chart* for the schedule is:

  - Waiting time for *P1 = 6; P2 = 0; P3 = 3*
  - **Average waiting time** = *(6 + 0 + 3)/3 = 3*
  - Much better than previous case
- **Convoy effect** – short processes behind a long process, all the other processes wait for the one big process to get off the CPU
  - Consider one *CPU-bound* and *many I/O-bound* processes
  - Result in *lower* CPU and device utilization

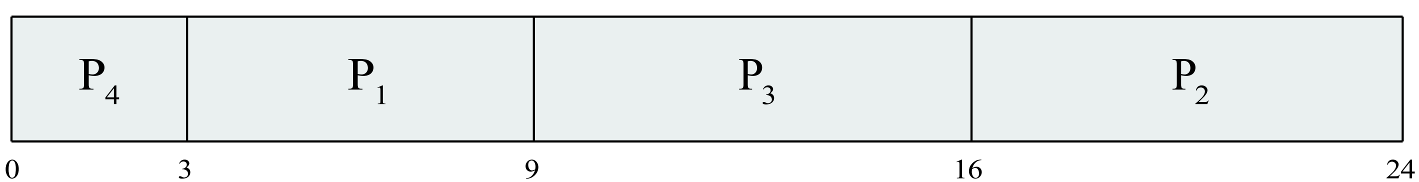
Shortest-Job-First (SJF) Scheduling

- **Motivation**: Moving a short process before a long one decreases the waiting time of the short process more than it increases the waiting time of the long process
  - The *shortest-next-CPU-burst* algorithm
- Associate with each process *the length of its next CPU burst*
  - When the CPU is available, it is assigned to the process that has the *smallest next CPU burst*
  - *FCFS scheduling* is used if the next CPU bursts of two processes are the same
- SJF is provably *optimal* – gives minimum average waiting time for a given set of processes
  - The difficulty is how to know the *length of the next CPU request*
  - Could ask the user

Example of SJF scheduling

Process	Burst Time (ms)
P_1	6
P_2	8
P_3	7
P_4	3

- SJF scheduling *Gantt chart*



  - **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 *exponential averaging* of the measured lengths of previous CPU bursts as follows

$$\alpha \in [0,1]$$

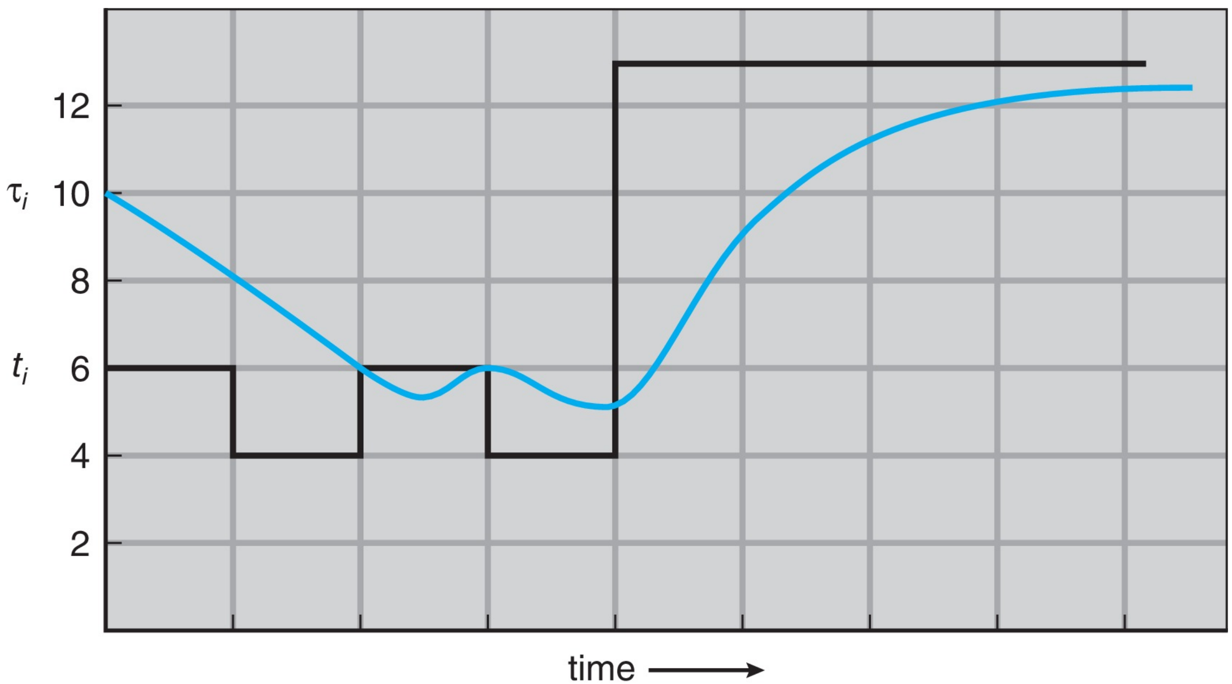
τ_n: predicted value for the next CPU burst

t_n: actual length of n^{th} CPU burst

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

- Commonly, *α* controls the relative weight of recent and past history in the prediction and *sets to ½*
- *Preemptive* version called *Shortest-Remaining-Time-First* (SRTF)

- An exponential average with $\alpha = 1/2$ and $\tau_0 = 10$



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

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

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

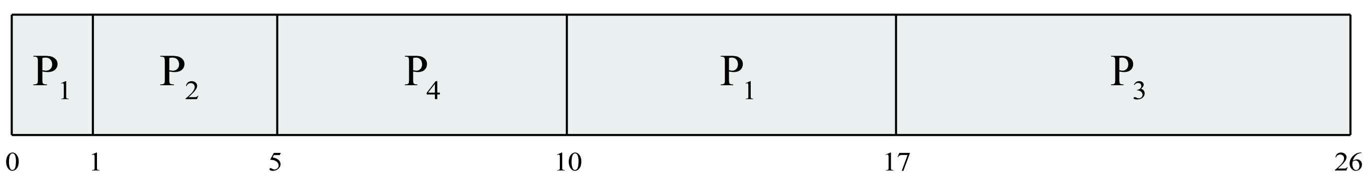
- Since both α and $(1 - \alpha)$ are less than or equal to 1, each successive term has less weight than its predecessor

Shortest-Remaining-Time-First (SRTF)

- **Motivation**: now, we add the concepts of *varying arrival times* and *preemption* to the analysis

Process	Arrival Time	Burst Time (ms)
P1	*0*	*8*
P2	*1*	*4*
P3	*2*	*9*
P4	*3*	*5*

- Preemptive SJF *Gantt Chart*



  - **Average waiting time** = *[(10-1)+(1-1)+(17-2)+(5-3)]/4 = 6.5*

- The value for *nonpreemptive SJF scheduling*?

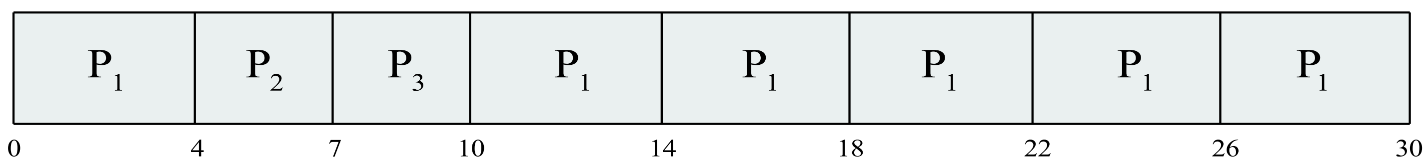
Round Robin (RR) Scheduling

- **Motivation**: try scheduling algorithm similar to *FCFS scheduling*, but *preemption* is added to enable the system to switch between processes
- 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

Example of RR with Time Quantum q = 4

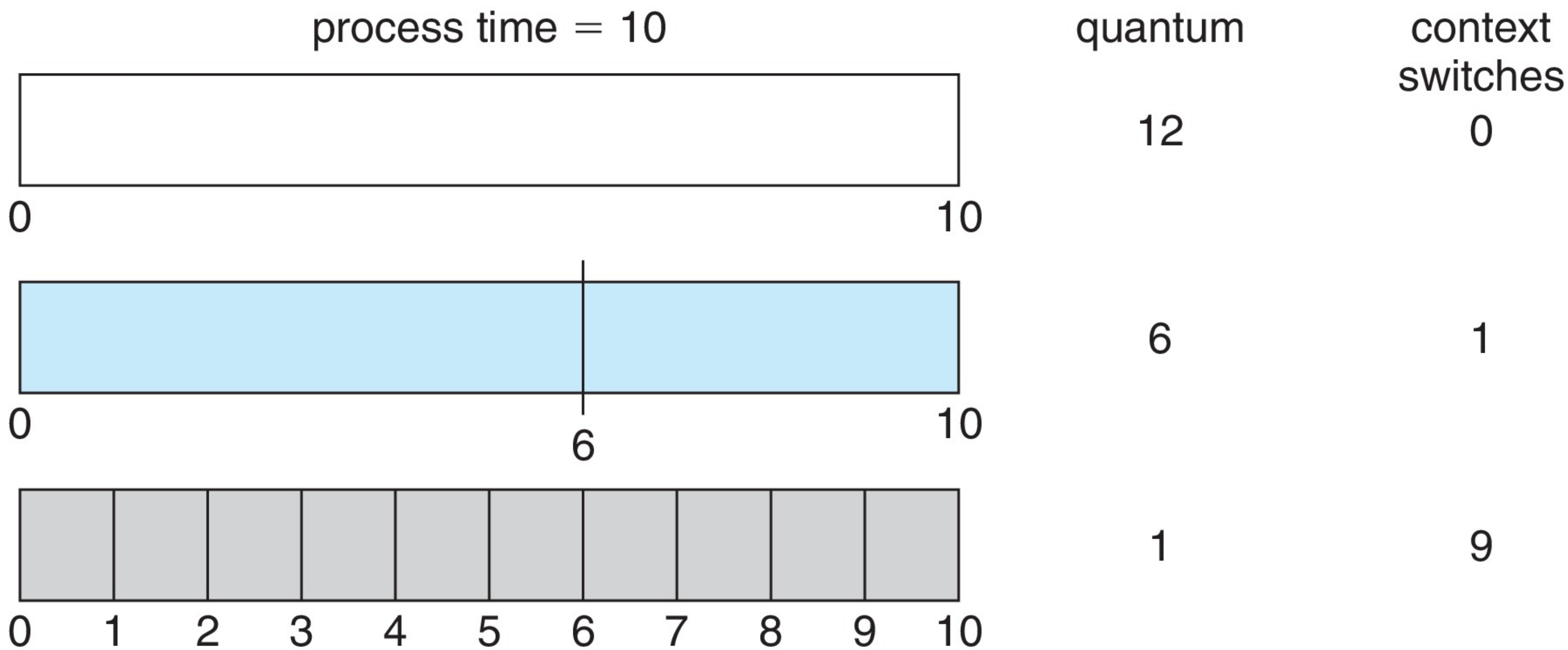
Process	Burst Time
P_1	24
P_2	3
P_3	3

- The *Gantt chart* is:

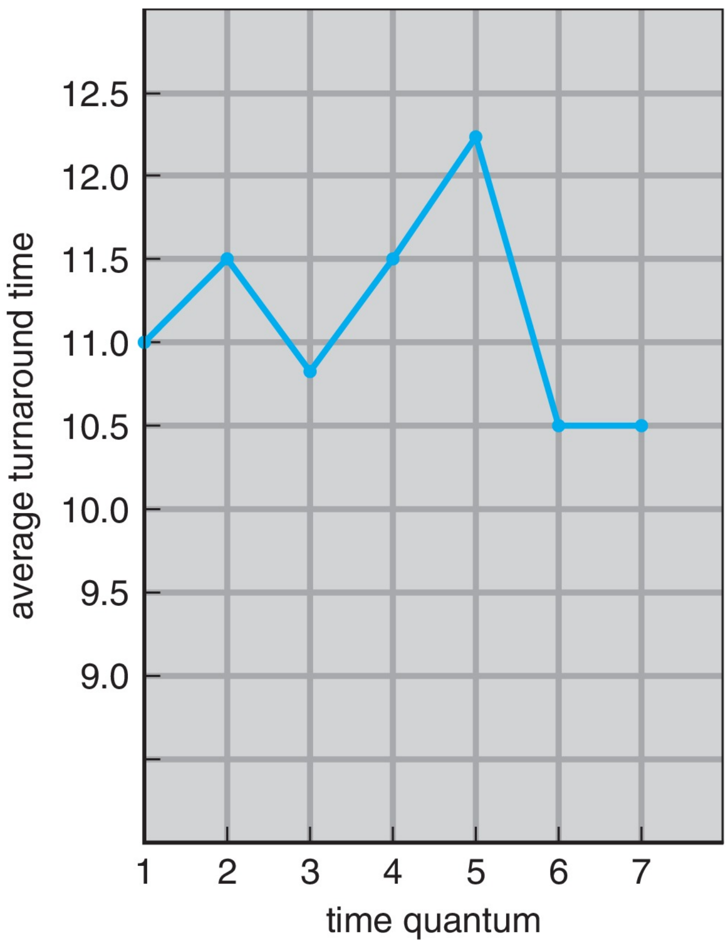


  - **Average waiting time** = ?
- Typically, *higher average turnaround* than SJF, but *better response*
- ***q*** should be large compared to context switch time
- ***q*** usually *10ms* to *100ms*, context switch *< 10µsec*

Time Quantum and Context Switch Time



Turnaround Time Varies With The Time Quantum



process	time
P_1	6
P_2	3
P_3	1
P_4	7

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

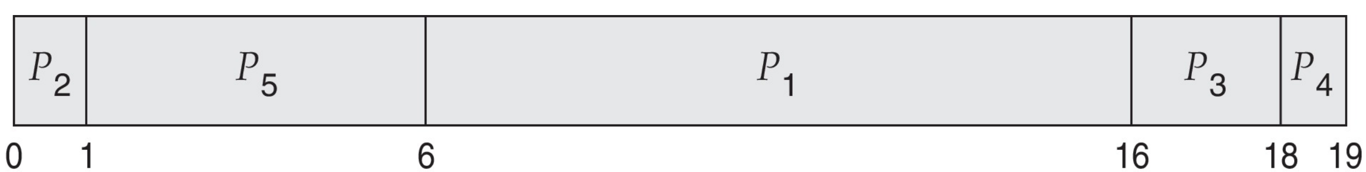
Priority Scheduling

- **Motivation**: A *priority number* (*integer*) is associated with each process
- The CPU is allocated to the process with the *highest priority* (*smallest integer ≡ highest priority*). Equal-priority processes are scheduled in **FCFS** or **RR**
  - 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
  - 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*

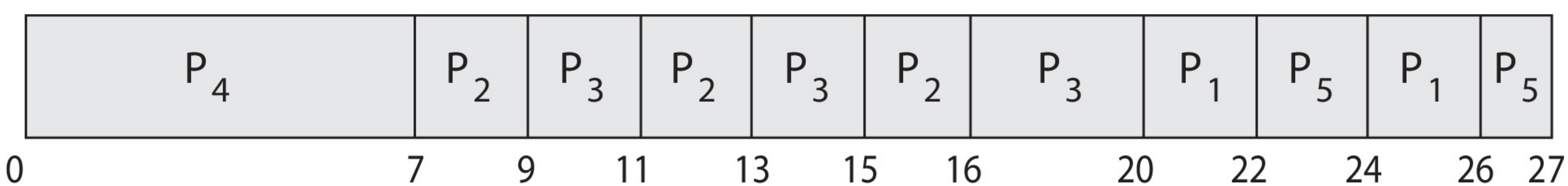


  - **Average waiting time** = *8.2*

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

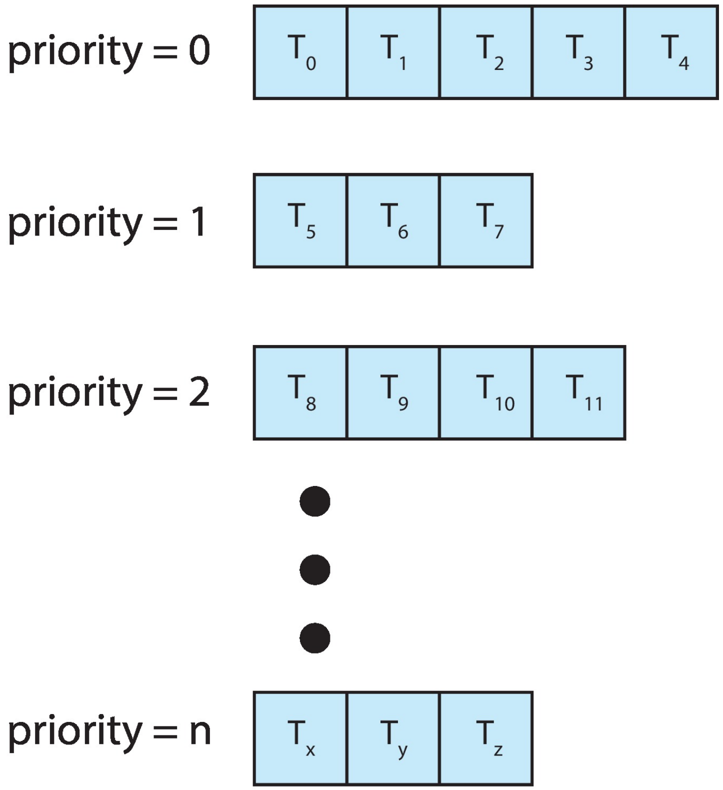
- Run the process with the highest priority. Processes with the same priority run Round-Robin
- *Gantt Chart* with time quantum $q = 2$ ms



  - **Average waiting time** = ?

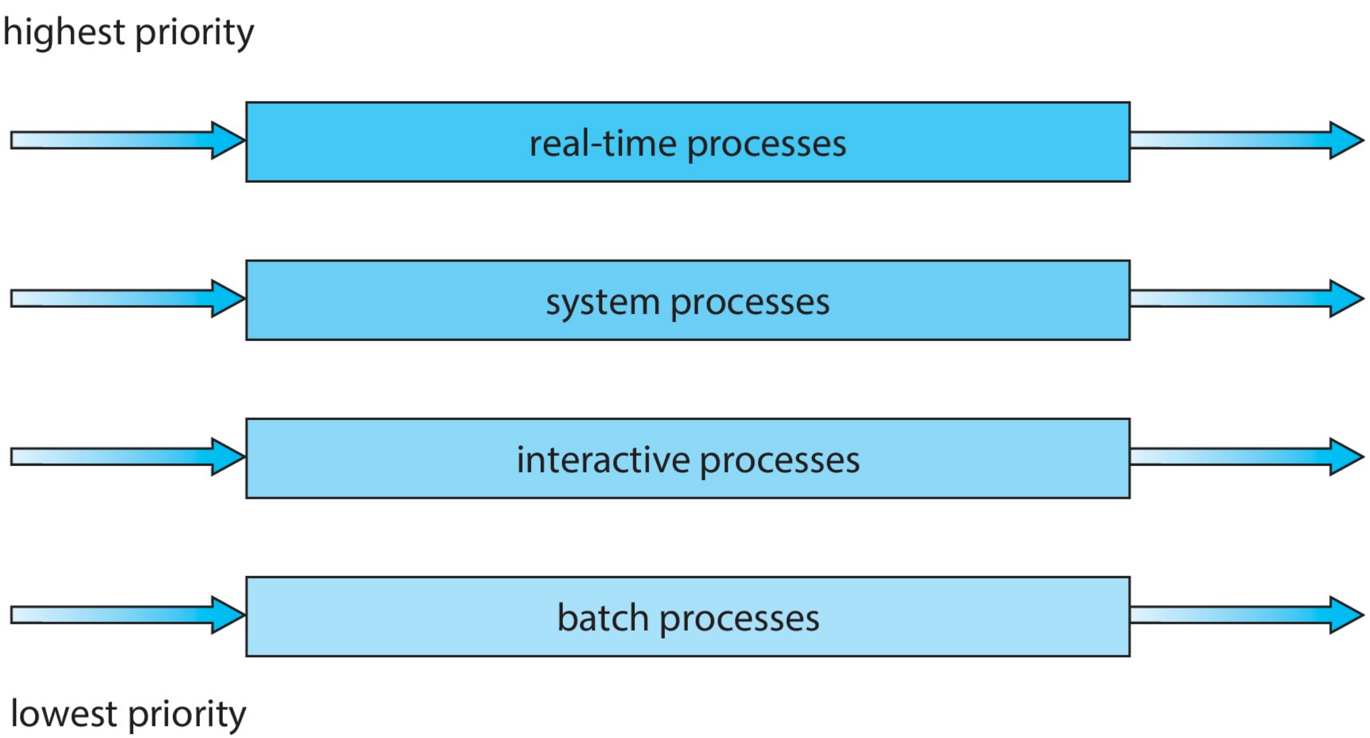
Multilevel Queue

- **Motivation**: with priority scheduling, have separate queues for each priority
- Schedule the process in the highest-priority queue!



Example of Multilevel Queue

- Prioritization based upon process type

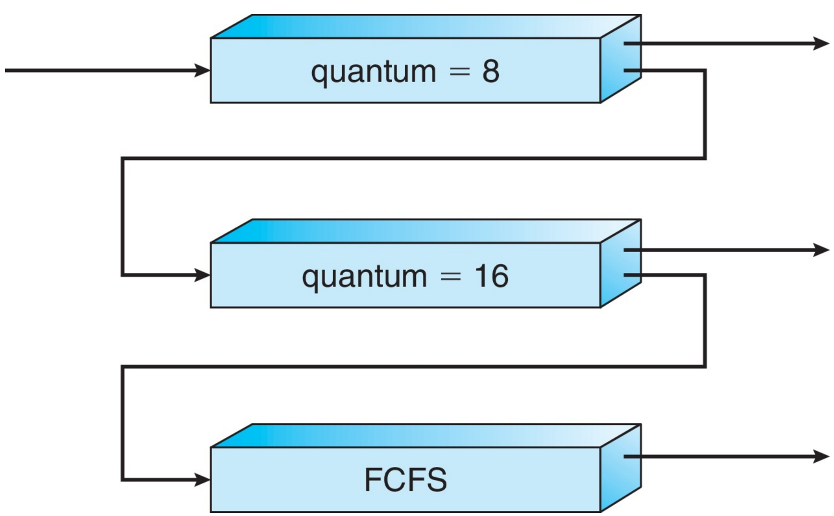


Multilevel Feedback Queue

- **Motivation**: 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
- This scheme leaves I/O-bound and interactive processes — which are typically characterized by short CPU bursts — in the higher-priority queues and a process that waits too long in a lower-priority queue *may be moved to a higher-priority queue*

Example of Multilevel Feedback Queue

- Three queues:
  - **Q0** – RR with time quantum 8 milliseconds
  - **Q1** – RR with time quantum 16 milliseconds
  - **Q2** – 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 **Q1**
  - At **Q1** job is again served FCFS and receives *16* additional milliseconds
    - If it still does not complete, it is preempted and moved to queue **Q2**

Thread Scheduling

- Distinction between *user-level* and *kernel-level* threads
- When threads supported, *threads scheduled, not processes*
- Many-to-one and many-to-many models, thread library schedules user-level threads to run on *Light-Weight Process* (**LWP**)
  - Known as *Process-Contention Scope* **(PCS)** since scheduling competition is within the process
  - Typically done via *priority set by programmer*
- Kernel thread scheduled onto available CPU is *System-Contention Scope* **(SCS)** – competition among all threads in system

POSIX Pthread Scheduling

- 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
- Can be limited by OS – Linux and macOS only allow **PTHREAD_SCOPE_SYSTEM**
- Pthread IPC (Inter-process Communication) provides two functions for setting
  - `pthread attr setscope(pthread attr t *attr, int scope)`
  - `pthread attr getscope(pthread attr t *attr, int *scope)`

Pthread Scheduling API

```
#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;

   /* get the default attributes */

   pthread_attr_init(&attr);

   /* first inquire on 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("PTHREAD_SCOPE_PROCESS");

      else if (scope == PTHREAD_SCOPE_SYSTEM)

         printf("PTHREAD_SCOPE_SYSTEM");

      else
         fprintf(stderr, "Illegal scope value.\n");

   }
```

Pthread Scheduling API (Cont.)

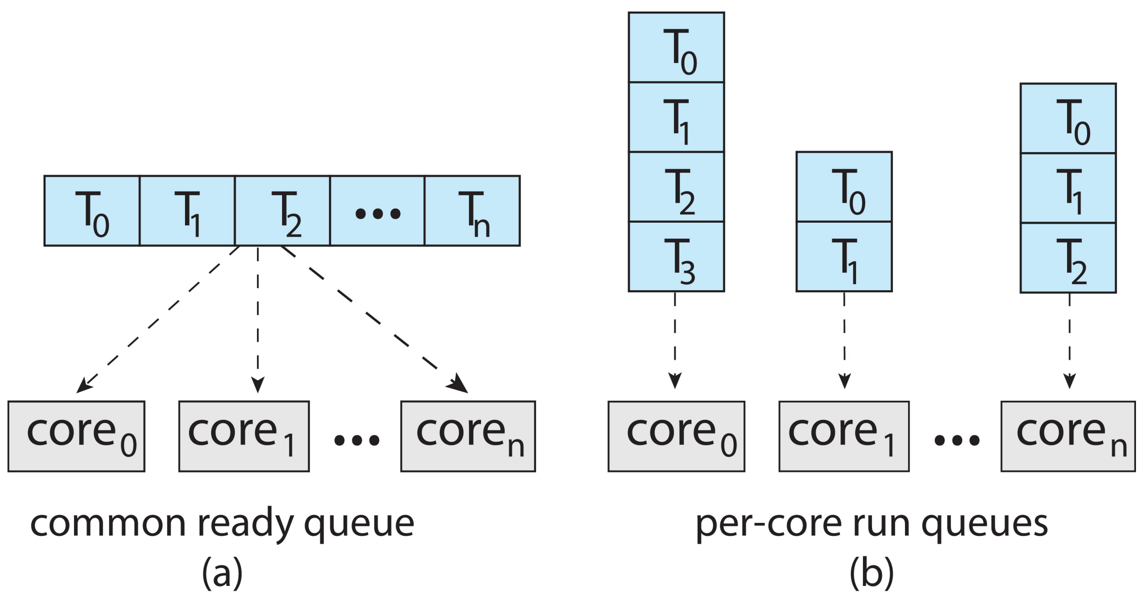
```
    /* set the scheduling algorithm to PCS or SCS */

    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
- *Multiprocessor* may be any one of the following architectures:
  - Multicore CPUs
  - Multithreaded cores
  - NUMA systems
  - Heterogeneous multiprocessing
- *Multiprocessor scheduling*
  - There is no one best solution

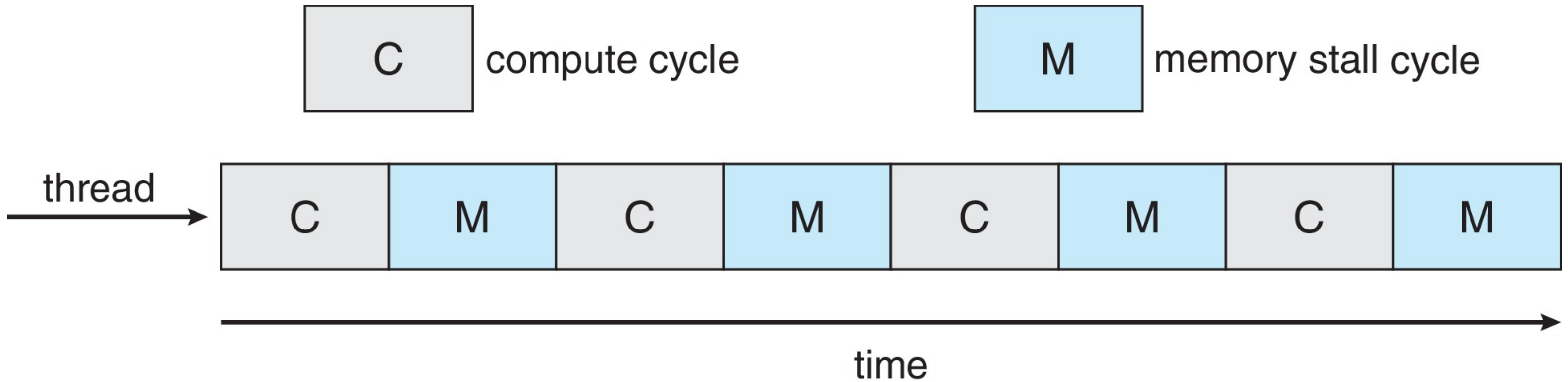
Multiple-Processor Scheduling (Cont.)

- *Symmetric multiprocessing* (**SMP**) is where each processor is self-scheduling
- Two possible strategies
  - All threads may be in a common ready queue (Fig. a)
  - Each processor may have its own private queue of threads (Fig. b)



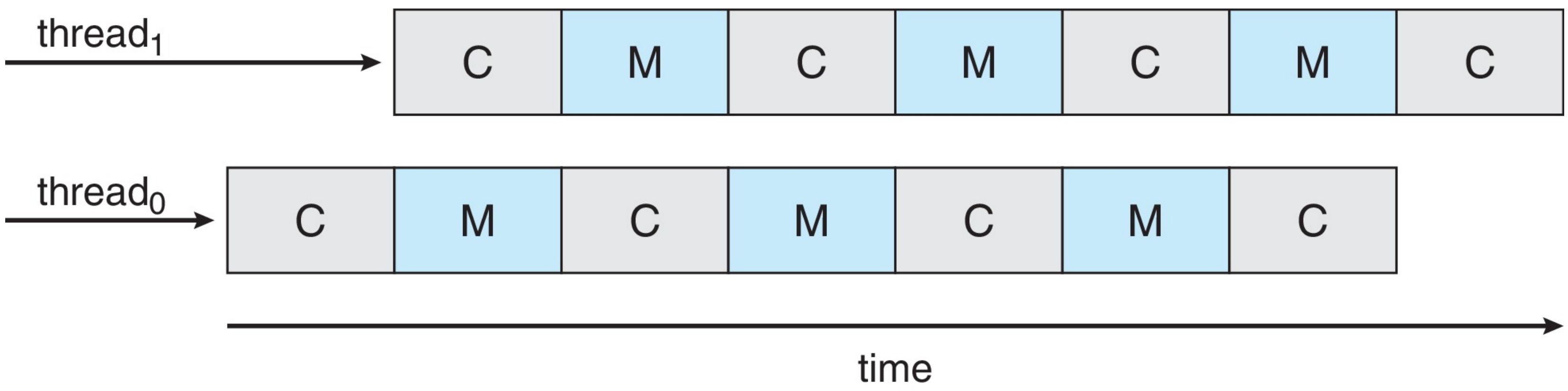
Multicore Processors

- 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

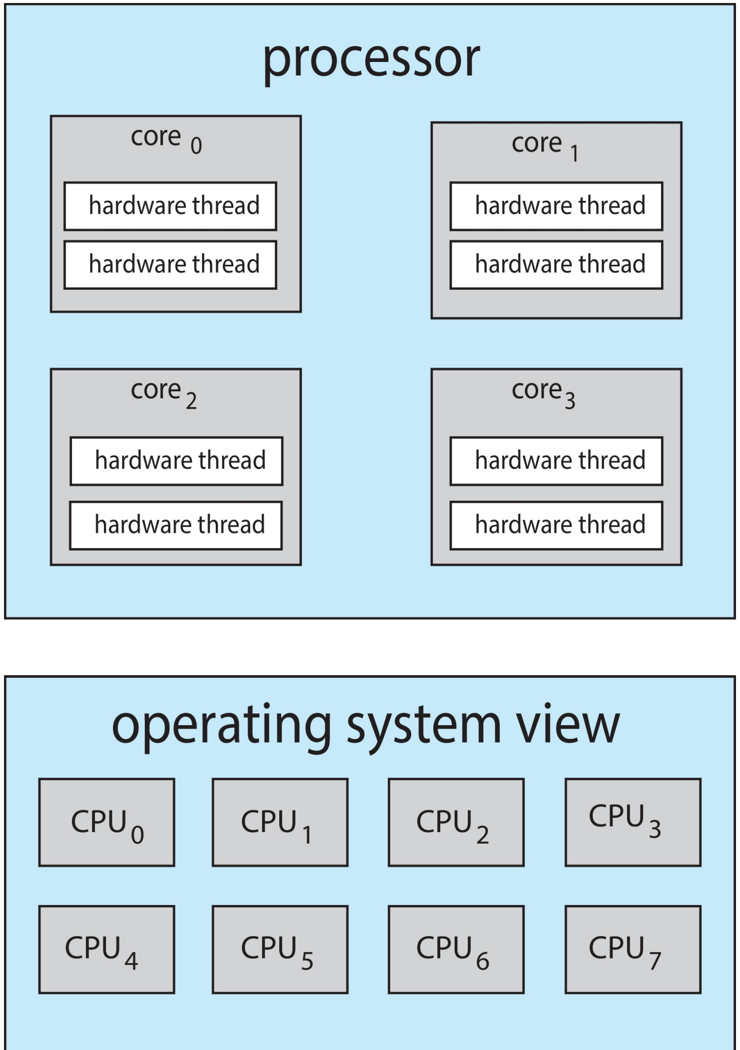


Multithreaded Multicore System

- Each core has > 1 hardware threads.
- If one thread has a memory stall, switch to another thread!



Multithreaded Multicore System



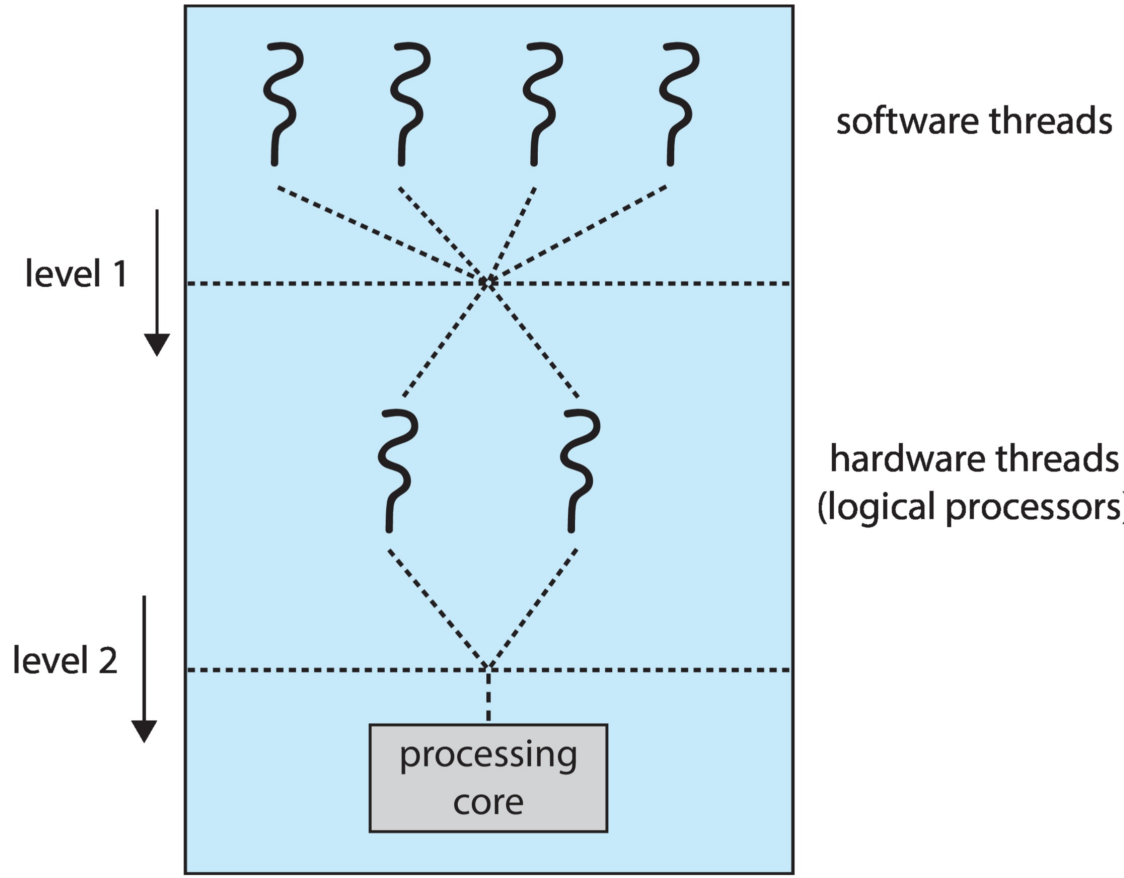
- *Chip-multithreading* (**CMT**) assigns each core multiple hardware threads (Intel refers to this as *hyperthreading*)
- Each hardware thread maintains its architectural state, such as *instruction pointer* and *register set*
- On a quad-core system with 2 hardware threads per core (e.g., Intel i7), the operating system sees 8 logical processors

Multithreaded Multicore 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
- *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

Multiple-Processor Scheduling – Processor Affinity

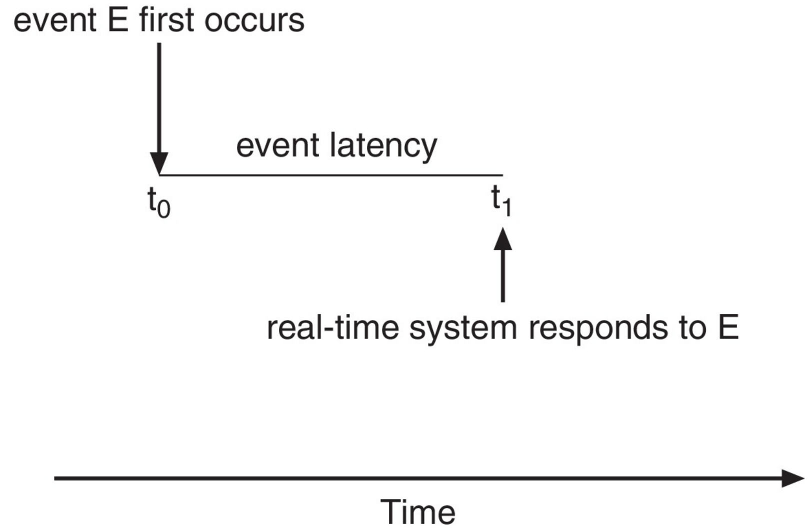
- When a thread has been running on one processor, the cache contents of that processor stores the memory accesses by that thread.
- We refer to this as a thread having *affinity* for a processor (i.e. "processor affinity")
- *Load balancing* may affect processor affinity as a thread may be moved from 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 process to specify a set of processors it may run on.

Real-Time CPU Scheduling

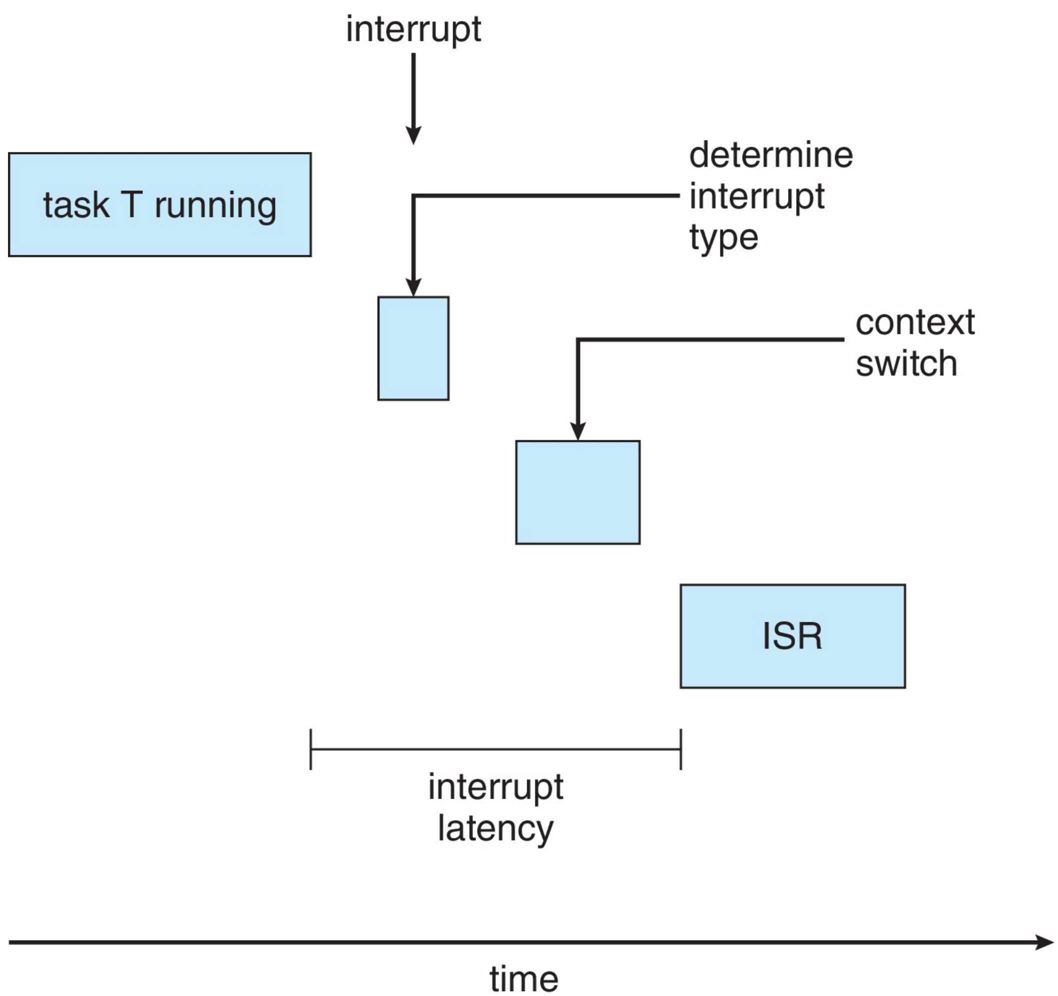
- Can present obvious challenges
- *Soft real-time systems* – Critical real-time tasks have the highest priority, but no guarantee as to when tasks will be scheduled
- *Hard real-time systems* – task must be serviced by its deadline

Types of Latencies

- *Event latency* – the amount of time that elapses from *when an event occurs* to *when it is serviced*.
- Two types of latencies affect performance
  1. *Interrupt latency* – time from *arrival of interrupt* to *start of routine* that services interrupt
  2. *Dispatch latency* – time for schedule to take current process off CPU and switch to another

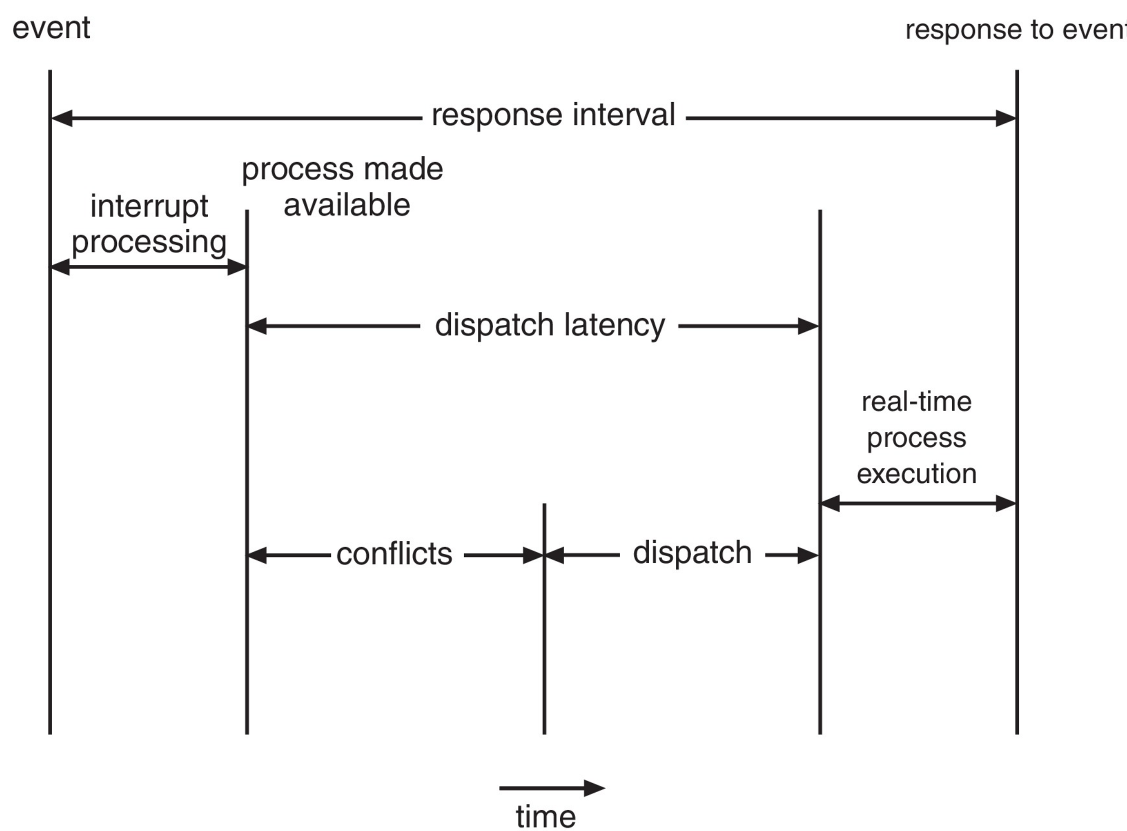


Interrupt Latency



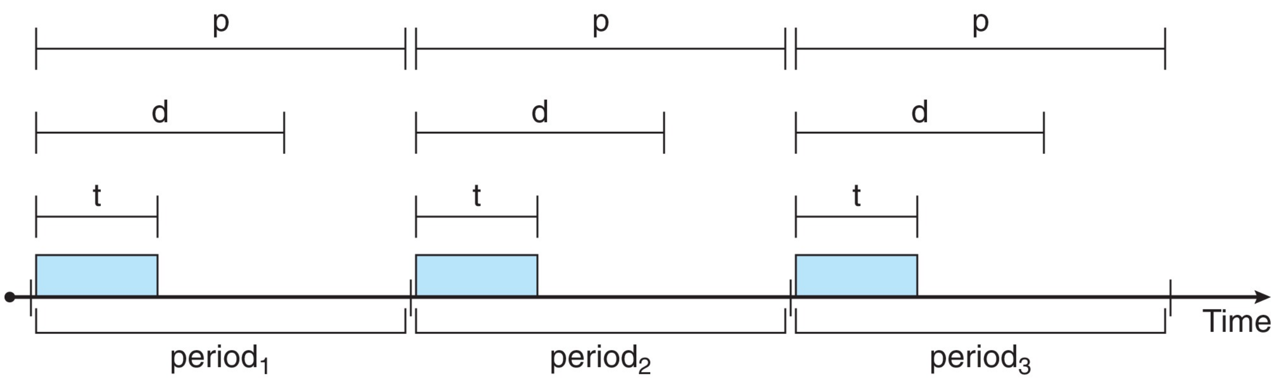
Dispatch Latency

- Conflict phase of dispatch latency:
  1. *Preemption* of any process running in kernel mode
  2. *Release* by low-priority process of resources needed by high-priority processes



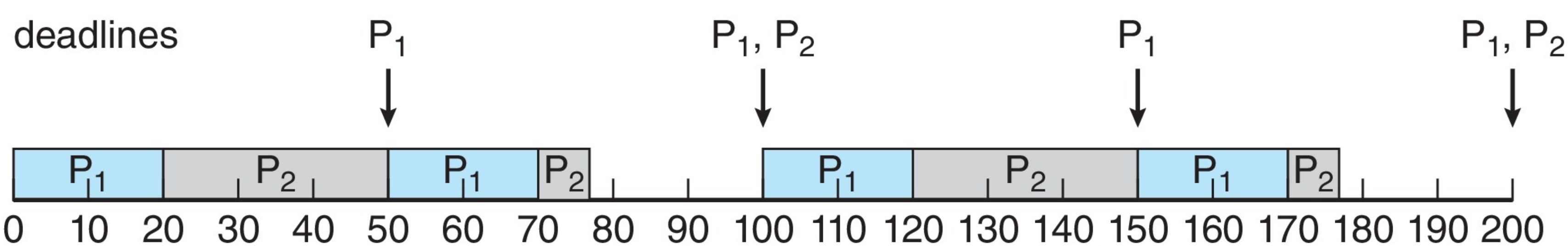
Priority-based Scheduling

- For real-time scheduling, scheduler must support *preemptive*, *priority-based scheduling*
  - But only guarantees soft real-time
- For hard real-time, it must also provide ability to meet deadlines
- Processes have new characteristics: *periodic* ones require CPU at constant intervals
  - Has processing time *t*, deadline *d,* period *p*
  - $0 \leq t \leq d \leq p$
  - *Rate of periodic task* is $1/p$



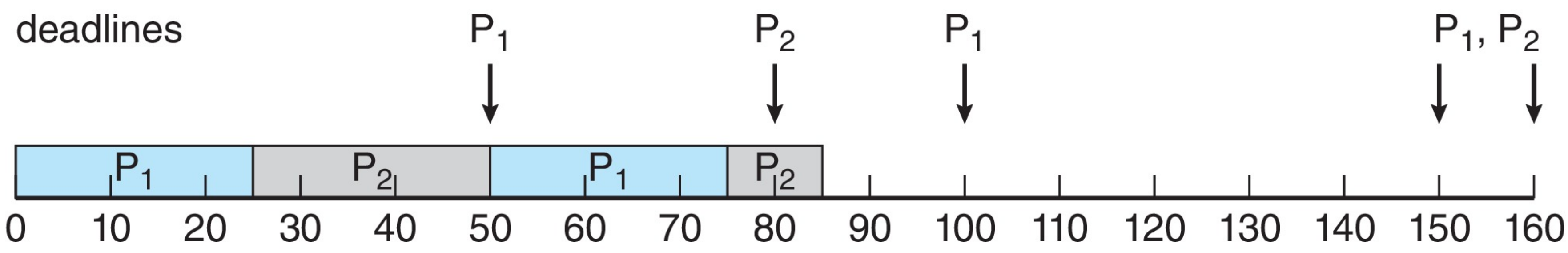
Rate Monotonic Scheduling

- A priority is assigned based on the inverse of its period
- Shorter periods = higher priority;
- Longer periods = lower priority
- P_1 is assigned a higher priority than P_2.



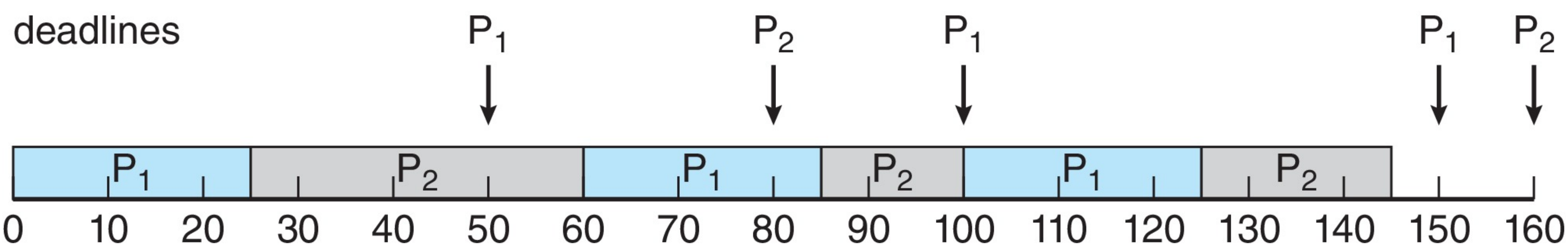
Missed Deadlines with Rate Monotonic Scheduling

- Process P2 misses finishing its deadline at time 80



Earliest Deadline First Scheduling (EDF)

- Priorities are assigned according to deadlines:
  - the earlier the deadline, the higher the priority;
  - the later the deadline, the lower the priority



Proportional Share Scheduling

- *T* shares are allocated among all processes in the system
- An application receives *N* shares where $N < T$
- This ensures each application will receive N / T of the total processor time

POSIX Real-Time Scheduling

- The *POSIX.1b* standard
- API provides functions for managing real-time threads
- Defines two scheduling classes for real-time threads:
  - **SCHED_FIFO** – threads are scheduled using a FCFS strategy with a FIFO queue. There is no time-slicing for threads of equal priority
  - **SCHED_RR** – similar to SCHED_FIFO except time-slicing occurs for threads of equal priority
- Defines two functions for getting and setting scheduling policy:
  - `pthread_attr_getsched_policy(pthread_attr_t *attr, int *policy)`
  - `pthread_attr_setsched_policy(pthread_attr_t *attr, int policy)`

POSIX Real-Time Scheduling API

```
#include <pthread.h>

#include <stdio.h>

#define NUM_THREADS 5
```

```
int main(int argc, char *argv[])

{

   int i, policy;
   pthread_t_tid[NUM_THREADS];

   pthread_attr_t attr;

   /* get the default attributes */

   pthread_attr_init(&attr);

   /* get the current scheduling policy */
   if (pthread_attr_getschedpolicy(&attr, &policy) != 0)

      fprintf(stderr, "Unable to get policy.\n");

   else {

      if (policy == SCHED_OTHER) printf("SCHED_OTHER\n");

      else if (policy == SCHED_RR) printf("SCHED_RR\n");

      else if (policy == SCHED_FIFO) printf("SCHED_FIFO\n");

   }
```

POSIX Real-Time Scheduling API (Cont.)

```
   /* set the scheduling policy - FIFO, RR, or OTHER */
   if (pthread_attr_setschedpolicy(&attr, SCHED_FIFO) != 0)

      fprintf(stderr, "Unable to set policy.\n");

   /* 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);

}
```

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
  - Map into global priority with numerically lower values indicating higher priority
  - Higher priority gets larger *q*
  - Task runnable as long as time left in time slice (*active*)
  - If no time left (*expired*), not runnable until all other tasks use their slices
  - All runnable tasks tracked in per-CPU *run-queue 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 +

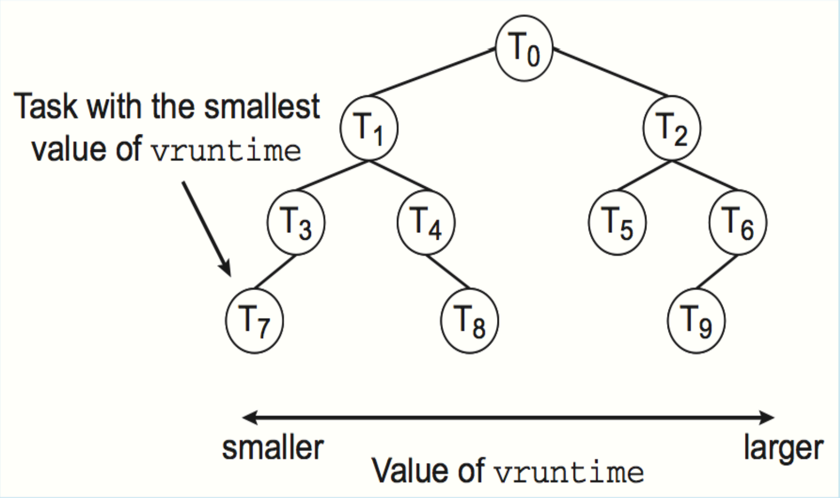
- Completely Fair Scheduler (**CFS**)
- Scheduling classes
  - Each has specific priority
  - Scheduler picks highest priority task in highest scheduling class
  - Rather than quantum based on fixed time allotments, based on proportion of CPU time
  - 2 scheduling classes included, others can be added
    - default
    - real-time

Linux Scheduling in Version 2.6.23 +

- 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

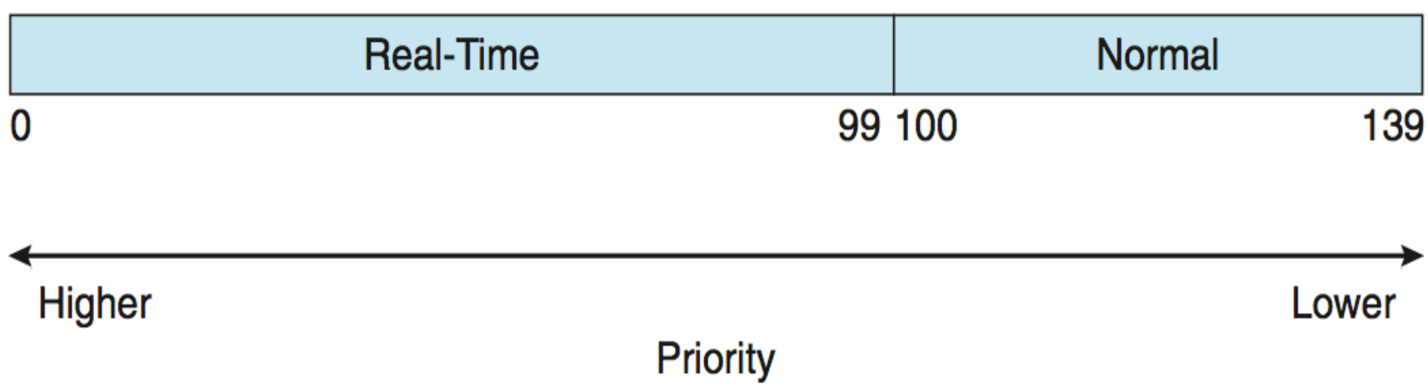
The Linux CFS scheduler provides an efficient algorithm for selecting which task to run next. Each runnable task is placed in a red-black tree—a balanced binary search tree whose key is based on the value of `vruntime`. This tree is shown below:



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.

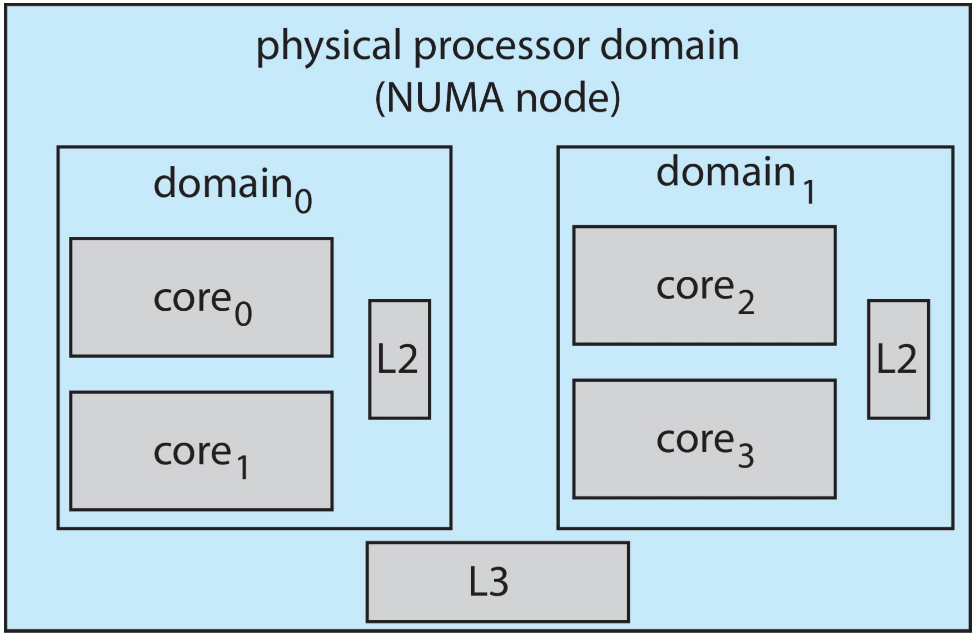
Linux Real-time Scheduling

- 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



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*

Windows Priority Classes

- Win32 API identifies several priority classes to which a process can belong
  - **REALTIME_PRIORITY_CLASS**, **HIGH_PRIORITY_CLASS**, **ABOVE_NORMAL_PRIORITY_CLASS**, **NORMAL_PRIORITY_CLASS**, **BELOW_NORMAL_PRIORITY_CLASS**, **IDLE_PRIORITY_CLASS**
  - All are variable except **REALTIME**
- A thread within a given priority class has a relative priority
  - **TIME_CRITICAL, HIGHEST**, **ABOVE_NORMAL, NORMAL**, **BELOW_NORMAL**, **LOWEST**, **IDLE**
- Priority class and relative priority combine to give numeric priority
- Base priority is **NORMAL** within the class
- If quantum expires, priority lowered, but never below base

Windows Priority Classes (Cont.)

- If wait occurs, priority boosted depending on what was waited for
- Foreground window given 3x priority boost
- Windows 7 added *user-mode scheduling* (**UMS**)
  - Applications create and manage threads independent of kernel
  - For large number of threads, much more efficient
  - UMS schedulers come from programming language libraries like C++ *Concurrent Runtime* (**ConcRT**) framework

Windows 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

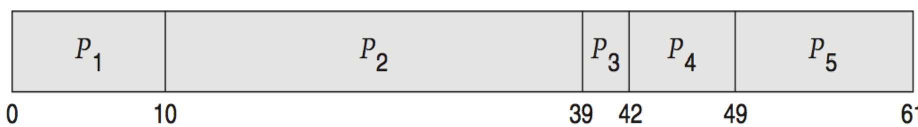
Algorithm Evaluation

- 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:

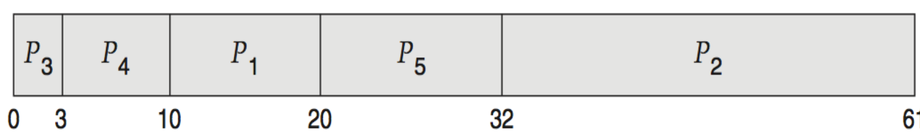
Process	Burst Time
P_1	10
P_2	29
P_3	3
P_4	7
P_5	12

Deterministic Evaluation

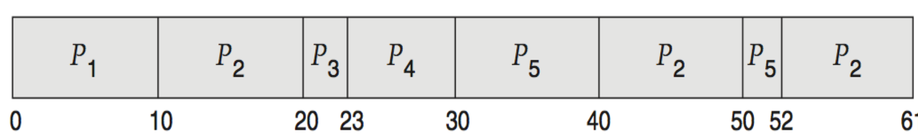
- For each algorithm, calculate minimum average waiting time
- Simple and fast, but requires exact numbers for input, applies only to those inputs
  - FCS is 28ms:



  - Non-preemptive SFJ is 13ms:



  - RR is 23ms:



Queueing Models

- Describes the arrival of processes, and CPU and I/O bursts probabilistically
  - Commonly exponential, and described by mean
  - Computes average throughput, utilization, waiting time, etc
- Computer system described as network of servers, each with queue of waiting processes
  - Knowing arrival rates and service rates
  - Computes utilization, average queue length, average wait time, etc

Little's Formula

- n = average queue length
- W = average waiting time in queue
- λ = average arrival rate into queue
- Little's law – in steady state, processes leaving queue must equal processes arriving, thus:

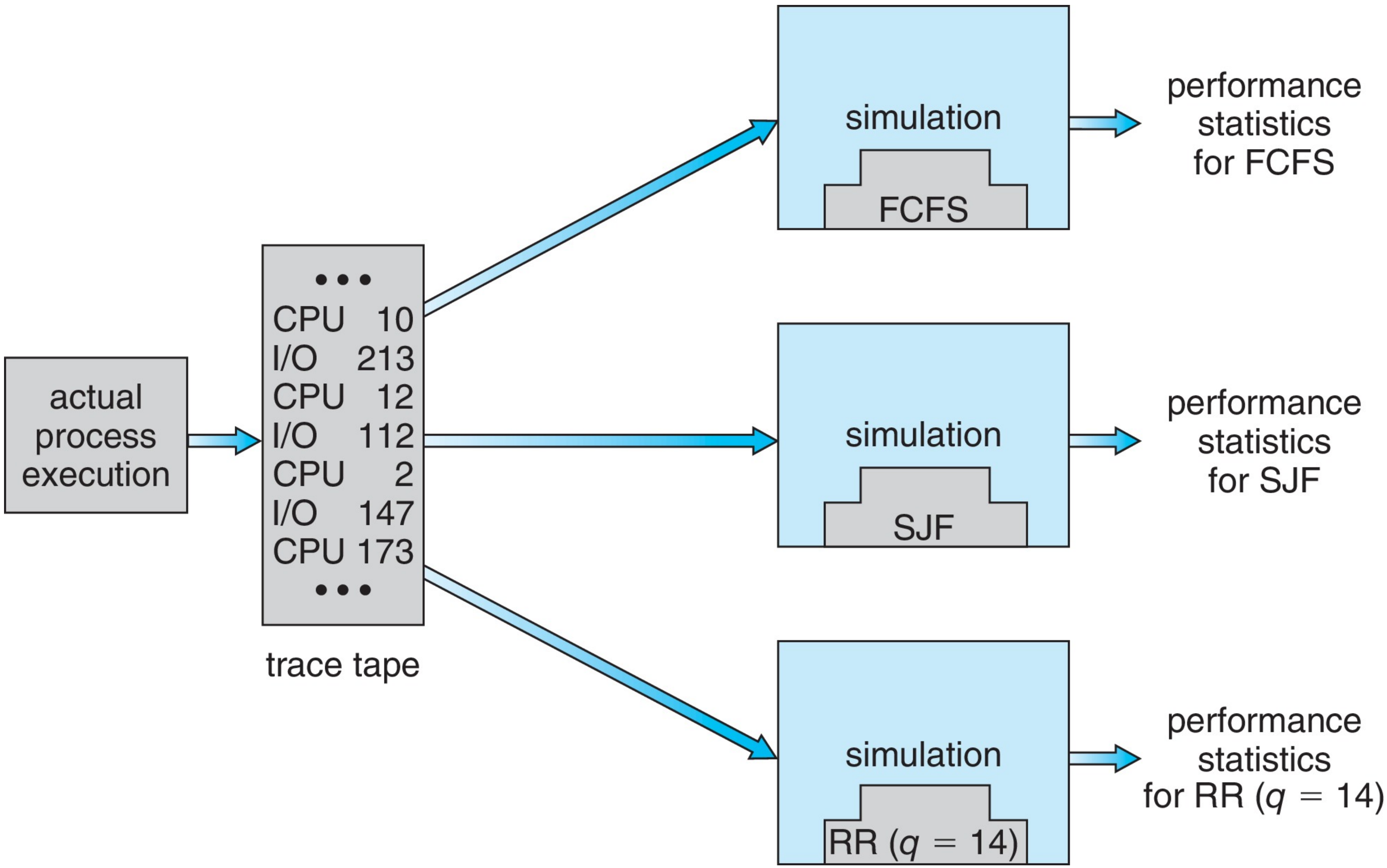
$$n = \lambda \times W$$

  - Valid for any scheduling algorithm and arrival distribution
- For example, if on average 7 processes arrive per second, and normally 14 processes in queue, then average wait time per process = 2 seconds

Simulations

- Queueing models limited
- *Simulations* more accurate
  - Programmed model of computer system
  - Clock is a variable
  - Gather statistics indicating algorithm performance
  - Data to drive simulation gathered via
    - Random number generator according to probabilities
    - Distributions defined mathematically or empirically
    - Trace tapes record sequences of real events in real systems

Evaluation of CPU Schedulers by Simulation



Implementation

- Even simulations have limited accuracy
- Just implement new scheduler and test in real systems
  - High cost, high risk
  - Environments vary
- Most flexible schedulers can be modified per-site or per-system
- Or APIs to modify priorities
- But again environments vary

Summary

- *CPU scheduling* is the task of selecting a waiting process from the ready queue and allocating the CPU to it. The CPU is allocated to the selected process by the dispatcher.

- *Scheduling algorithms* may be either *preemptive* (where the CPU can be taken away from a process) or *nonpreemptive* (where a process must voluntarily relinquish control of the CPU). Almost all modern operating systems are preemptive.

- Scheduling algorithms can be evaluated according to the following five criteria: (1) *CPU utilization*, (2) *throughput*, (3) *turnaround time*, (4) *waiting time*, and (5) *response time*.

- *First-come, first-served* (**FCFS**) scheduling is the simplest scheduling algorithm, but it can cause short processes to wait for very long processes.

Summary (Cont.)

- *Shortest-job-first* (**SJF**) scheduling is provably optimal, providing the shortest average waiting time. Implementing SJF scheduling is difficult, how- ever, because predicting the length of the next CPU burst is difficult.

- *Round-robin* (**RR**) scheduling allocates the CPU to each process for a time quantum. If the process does not relinquish the CPU before its time quantum expires, the process is preempted, and another process is scheduled to run for a time quantum.

- *Priority scheduling* assigns each process a priority, and the CPU is allocated to the process with the highest priority. Processes with the same priority can be scheduled in FCFS order or using RR scheduling.

Summary (Cont.)

- *Multilevel queue scheduling* partitions processes into several separate queues arranged by priority, and the scheduler executes the processes in the highest-priority queue. Different scheduling algorithms may be used in each queue.
- *Multilevel feedback queues* are similar to multilevel queues, except that a process may migrate between different queues.
- *Multicore processors* place one or more CPUs on the same physical chip, and each CPU may have more than one hardware thread. From the perspective of the operating system, each hardware thread appears to be a logical CPU.
- *Load balancing* on multicore systems equalizes loads between CPU cores, although migrating threads between cores to balance loads may invalidate cache contents and therefore may increase memory access times.

Summary (Cont.)

- *Soft real-time scheduling* gives priority to real-time tasks over non-real- time tasks. Hard real-time scheduling provides timing guarantees for real- time tasks,

- *Rate-monotonic real-time scheduling* schedules periodic tasks using a static priority policy with preemption.

- *Earliest-deadline-first* (**EDF**) scheduling assigns priorities according to deadline. The earlier the deadline, the higher the priority; the later the deadline, the lower the priority.

- *Proportional share scheduling* allocates *T* shares among all applications. If an application is allocated *N* shares of time, it is ensured of having N/T of the total processor time.

Summary (Cont.)

- **Linux** uses the *completely fair scheduler* (**CFS**), which assigns a proportion of CPU processing time to each task. The proportion is based on the virtual runtime (**vruntime**) value associated with each task.
- **Windows** scheduling uses a preemptive, 32-level priority scheme to determine the order of thread scheduling.
- *Modeling and simulations* can be used to evaluate a CPU scheduling algorithm.

End of Chapter 5