Operating Systems
[5. CPU Scheduling]

Chung-Wei Lin

cwlin@csie.ntu.edu.tw

CSIE Department

National Taiwan University

From Operating System Concepts (10th Edition)

Objectives

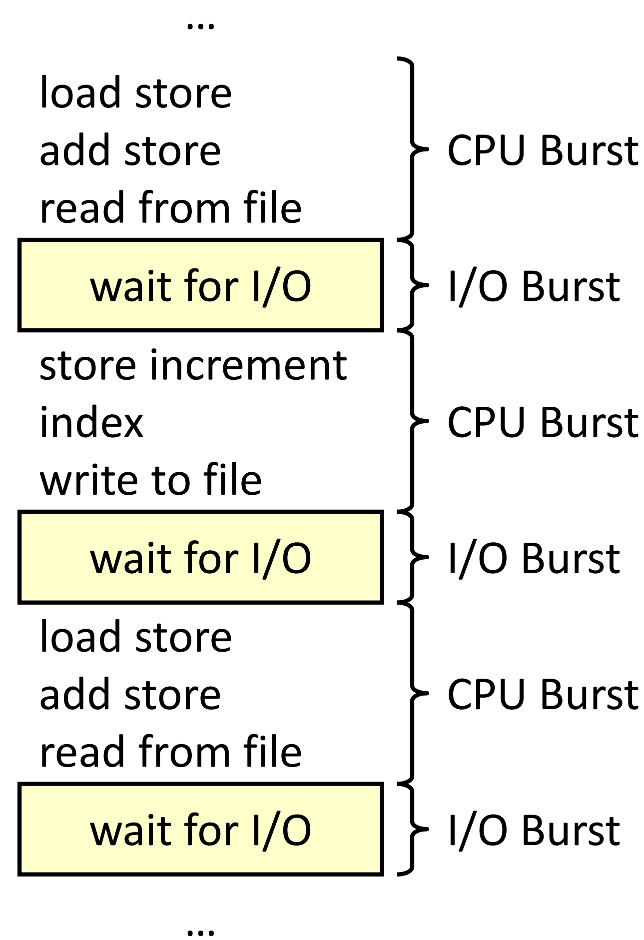
- Describe various CPU scheduling algorithms
- Assess CPU scheduling algorithms based on scheduling criteria
- Explain the issues related to multiprocessor and multicore scheduling
- Describe various real-time scheduling algorithms
- Describe the scheduling algorithms used in the Windows, Linux, and Solaris operating systems
- Apply modeling and simulations to evaluate CPU scheduling algorithms

Outline

- ❑ **Basic Concepts**
  - ➢ **CPU-I/O Burst Cycle**
  - ➢ CPU Scheduler
  - ➢ Preemptive and Nonpreemptive Scheduling
  - ➢ Dispatcher
- ❑ Scheduling Criteria
- ❑ Scheduling Algorithms
- ❑ Thread Scheduling
- ❑ Multi-Processor Scheduling
- ❑ Real-Time CPU Scheduling
- ❑ Operating-System Examples
- ❑ Algorithm Evaluation

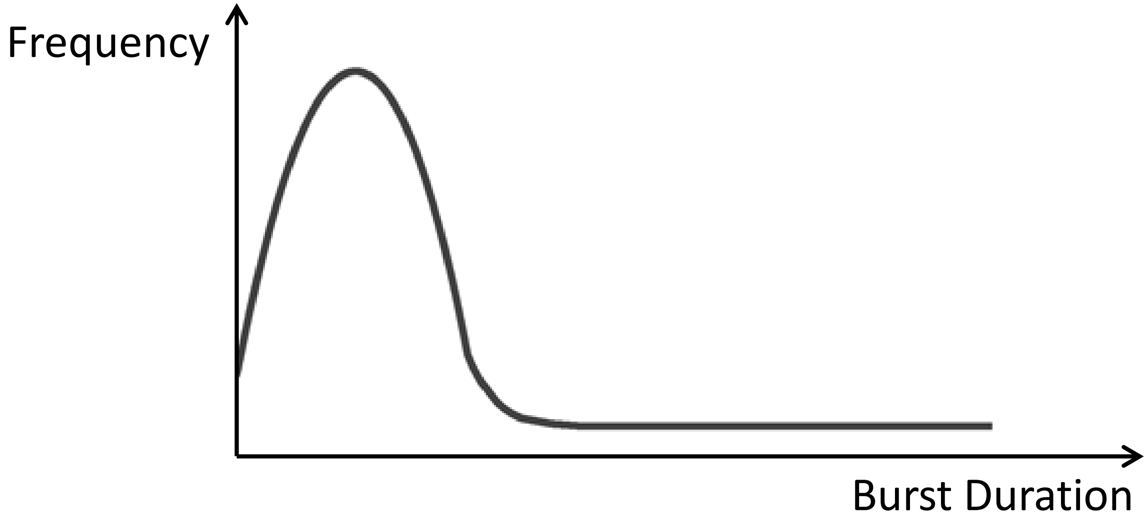
Basic Concepts

- **Objective**
  - Maximize CPU utilization with multiprogramming
- **CPU-I/O burst cycle**
  - Process execution consists of a **cycle** of CPU execution and I/O wait
  - Interleaving **CPU bursts** and **I/O bursts**



Histogram of CPU-Burst Times

- CPU-burst distribution is of main concern
  - A large number of short CPU bursts
  - A small number of long CPU bursts
  - An I/O-bound program typically has many short CPU bursts
  - A CPU-bound program might have a few long CPU bursts

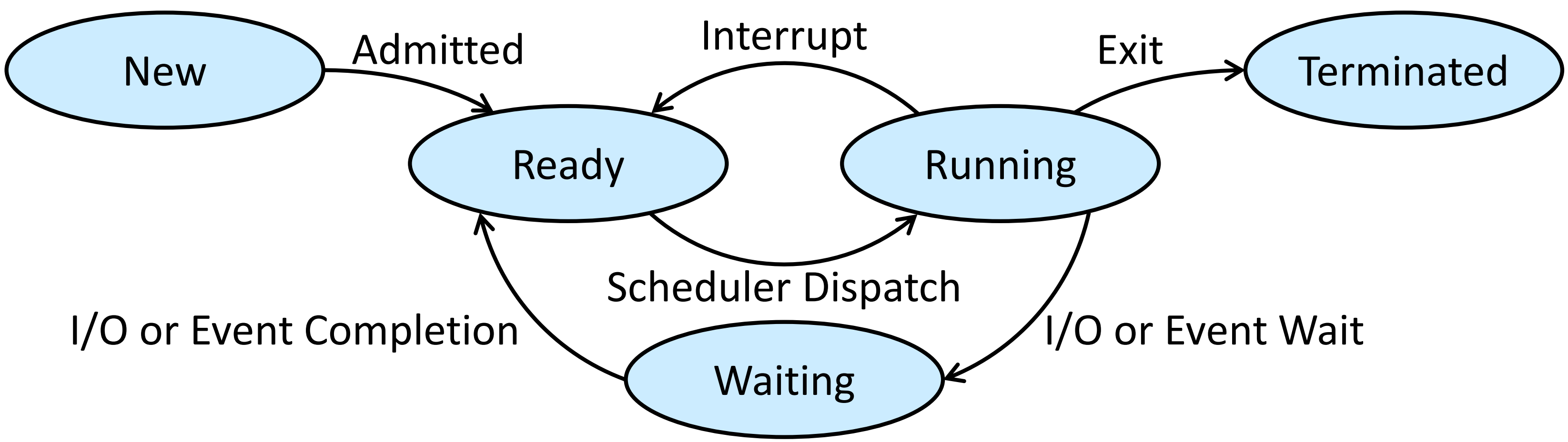


Outline

- **Basic Concepts**
  - CPU-I/O Burst Cycle
  - **CPU Scheduler**
  - Preemptive and Nonpreemptive Scheduling
  - Dispatcher
- Scheduling Criteria
- Scheduling Algorithms
- Thread Scheduling
- Multi-Processor Scheduling
- Real-Time CPU Scheduling
- Operating-System Examples
- Algorithm Evaluation

Recap: Process State

- As a process executes, it changes **state**
  - **New**: the process is being created
  - **Ready**: the process is waiting to be assigned to a processor
  - **Running**: instructions are being executed
    - Only one process can be running on any processor core at any instant
  - **Waiting**: the process is waiting for some event to occur
    - Examples: I/O completion, reception of a signal
  - **Terminated**: the process has finished execution



CPU Scheduler

- The **CPU scheduler** selects a process from the processes in memory that are ready to execute and allocates the CPU to it
  - The ready queue may be ordered in various ways
- CPU scheduling decisions may take place when a process
  - Switches from the running state to the waiting state
  - Switches from the running state to the ready state
  - Switches from the waiting state to the ready state
  - Terminates
- Situations 1 and 4
  - There is no choice in terms of scheduling
  - A new process (if one exists in the ready queue) must be selected
- Situations 2 and 3
  - There is a choice

Outline

- **Basic Concepts**
  - CPU-I/O Burst Cycle
  - CPU Scheduler
  - **Preemptive and Nonpreemptive Scheduling**
  - Dispatcher
- Scheduling Criteria
- Scheduling Algorithms
- Thread Scheduling
- Multi-Processor Scheduling
- Real-Time CPU Scheduling
- Operating-System Examples
- Algorithm Evaluation

Preemptive and Nonpreemptive Scheduling

- When scheduling takes place only under situations 1 and 4, the scheduling scheme is **nonpreemptive**
  - Once the CPU has been allocated to a process, the process keeps the CPU until it releases it either by terminating or by switching to the waiting state
- Otherwise, it is **preemptive**
  - Virtually all modern operating systems including Windows, MacOS, Linux, and UNIX use preemptive scheduling algorithms

Preemptive Scheduling and Race Conditions

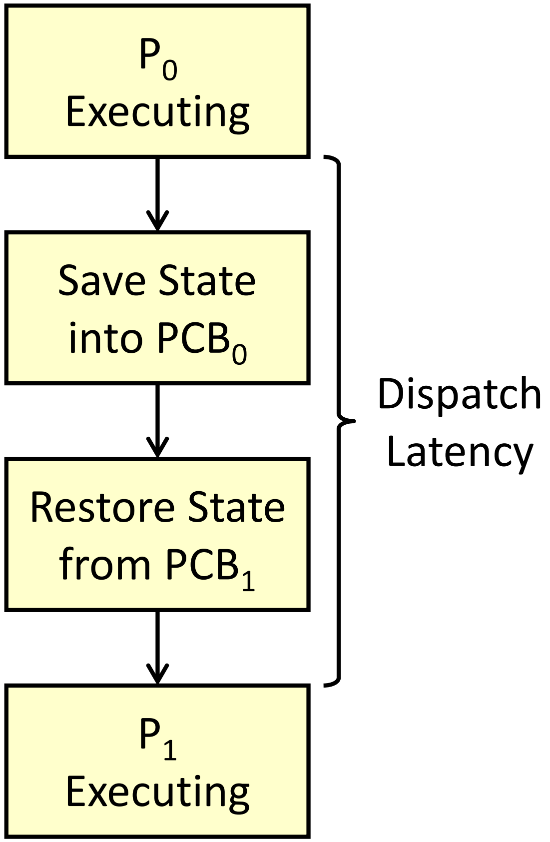
- Preemptive scheduling can result in race conditions when data are shared among several processes
  - Example
    - Two processes share data
    - While one process is updating the data, it is preempted so that the second process can run
    - The second process then tries to read the data, which are in an inconsistent state
- This issue will be explored in detail in
  - Chapter 6: Synchronization Tools
  - Chapter 7: Synchronization Examples

Outline

- **Basic Concepts**
  - CPU-I/O Burst Cycle
  - CPU Scheduler
  - Preemptive and Nonpreemptive Scheduling
  - **Dispatcher**
- Scheduling Criteria
- Scheduling Algorithms
- Thread Scheduling
- Multi-Processor Scheduling
- Real-Time CPU Scheduling
- Operating-System Examples
- Algorithm Evaluation

Dispatcher

- Dispatcher module gives control of the CPU to the process selected by the CPU scheduler, involving
  - Switching context
  - Switching to user mode
  - Jumping to the proper location in the user program to restart that program
- **<u>Dispatch latency</u>**
  - Time it takes for the dispatcher to stop one process and start another running



PCB: Process Control Block

Outline

- Basic Concepts
- <u>**Scheduling Criteria**</u>
- Scheduling Algorithms
- Thread Scheduling
- Multi-Processor Scheduling
- Real-Time CPU Scheduling
- Operating-System Examples
- Algorithm Evaluation

Scheduling Criteria

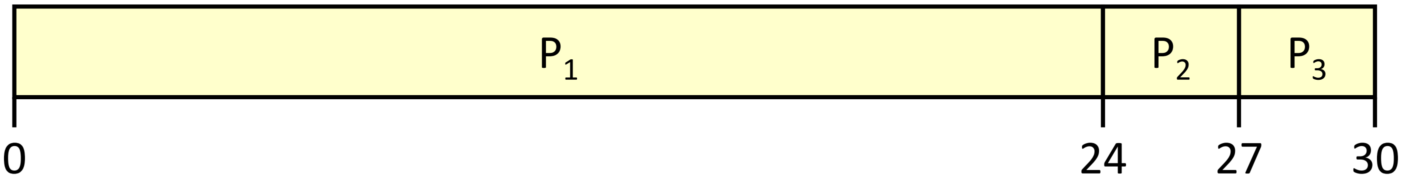
- Maximize **CPU utilization**
  - Keep the CPU as busy as possible
- Maximize **throughput**
  - Number of processes that complete their execution per time unit
- Minimize **turnaround time**
  - Amount of time to execute a particular process
- Minimize **waiting time**
  - Amount of time a process has been waiting in the ready queue
- Minimize **response time**
  - Amount of time it takes from when a request was submitted until the first response is produced

Outline

- Basic Concepts
- Scheduling Criteria
- **Scheduling Algorithms**
  - **First-Come, First-Served Scheduling**
  - Shortest-Job-First Scheduling
  - Round-Robin Scheduling
  - Priority Scheduling
  - Multilevel Queue Scheduling
  - Multilevel Feedback Queue Scheduling
- Thread Scheduling, Multi-Processor Scheduling
- Real-Time CPU Scheduling, Operating-System Examples
- Algorithm Evaluation

First-Come, First-Served (FCFS) Scheduling

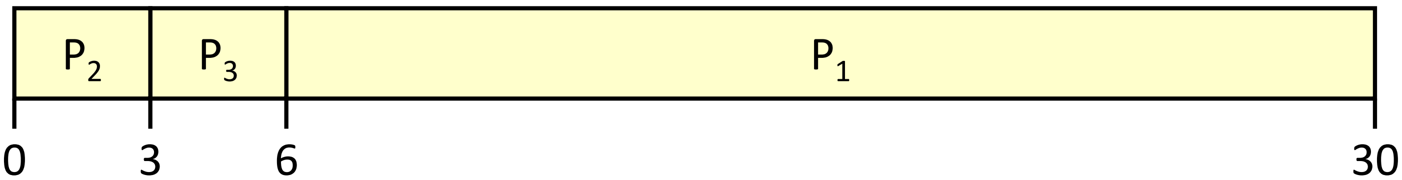
- Example
  - Process: P_1 P_2 P_3
  - Burst time: 24 3 3
- Suppose that the processes arrive in the order: P_1, P_2, P_3
- Gantt Chart for the schedule



- Waiting time
  - $P_1 = 0$, $P_2 = 24$, $P_3 = 27$
- Average waiting time
  - $(0 + 24 + 27) / 3 = 17$

FCFS Scheduling

- Suppose that the processes arrive in the order: P_2, P_3, P_1
- Gantt Chart for the schedule



- Waiting time
  - $P_1 = 6$, $P_2 = 0$, $P_3 = 3$
- Average waiting time
  - $(6 + 0 + 3) / 3 = 3$
- **Convoy effect**
  - All other processes wait for 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

Outline

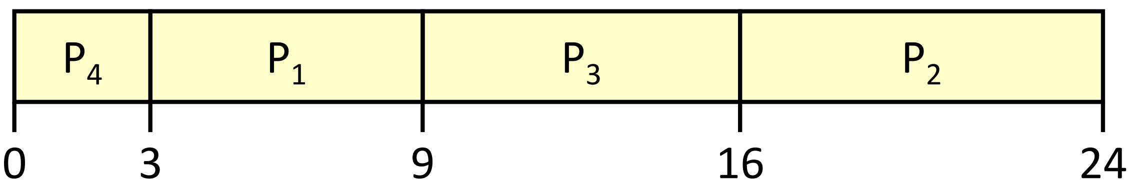
- Basic Concepts
- Scheduling Criteria
- **Scheduling Algorithms**
  - First-Come, First-Served Scheduling
  - **Shortest-Job-First Scheduling**
  - Round-Robin Scheduling
  - Priority Scheduling
  - Multilevel Queue Scheduling
  - Multilevel Feedback Queue Scheduling
- Thread Scheduling, Multi-Processor Scheduling
- Real-Time CPU Scheduling, Operating-System Examples
- Algorithm Evaluation

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 for minimizing average waiting time of a given set of processes
  - Preemptive version called **shortest-remaining-time-first**
- The difficulty is knowing the length of the next CPU request
  - How do we determine the length of the next CPU burst?
    - Estimate
    - Ask the user

SJF Scheduling

- Example
  - Process: P_1 P_2 P_3 P_4
  - Burst time: 6 8 7 3
- SJF scheduling chart



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

Prediction of Length of Next CPU Burst

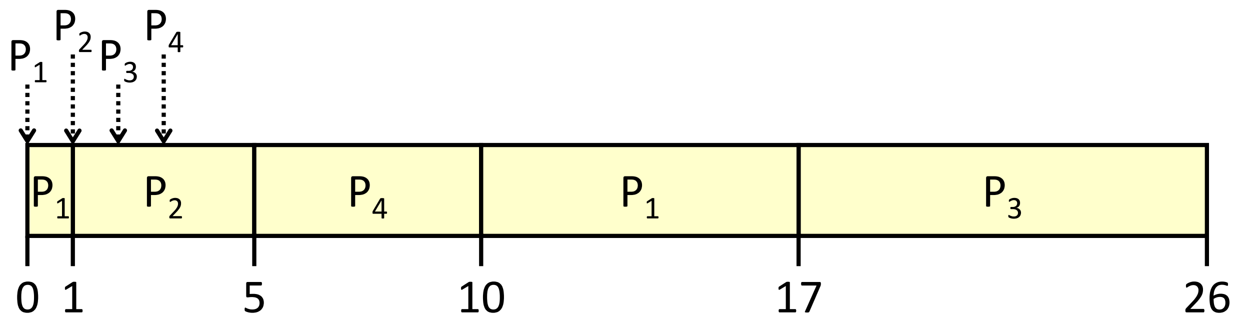
- Should it be similar to the previous one?
  - Pick process with shortest predicted next CPU burst
- Can be done by using the length of previous CPU bursts, using exponential averaging
  - t_n = actual length of n-th CPU burst
  - τ_n = predicted value for n-th CPU burst
  - α, $0 \leq \alpha \leq 1$
  - Define $\tau_{n+1} = \alpha\, t_n + (1 - \alpha)\, \tau_n$
- Example with $\alpha = 0.5$
  - τ_i = 10 8 6 6 5 9 11 12 …
  - t_i = 6 4 6 4 13 13 13 …

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

- Add the concepts of varying arrival times and preemption
  - Process: P_1 P_2 P_3 P_4
  - Arrival time: 0 1 2 3
  - Burst time: 8 4 9 5
- Preemptive "SJF" scheduling chart



- Average waiting time
  - [(10 – 1 – 0) + (1 – 1) + (17 – 2) + (5 – 3)] / 4 = 26 / 4 = 6.5
  - Please figure out how to compute it by yourself

Outline

- Basic Concepts
- Scheduling Criteria
- **<u>Scheduling Algorithms</u>**
  - First-Come, First-Served Scheduling
  - Shortest-Job-First Scheduling
  - **<u>Round-Robin Scheduling</u>**
  - Priority Scheduling
  - Multilevel Queue Scheduling
  - Multilevel Feedback Queue Scheduling
- Thread Scheduling, Multi-Processor Scheduling
- Real-Time CPU Scheduling, Operating-System Examples
- Algorithm Evaluation

Round-Robin (RR) Scheduling

- Each process gets a small unit of CPU time (**time quantum** q)
  - Usually 10-100 milliseconds
  - After the time has elapsed, the process is preempted and added to the end of the ready queue
    - Timer interrupts every quantum to schedule next process
- Assume n processes in the ready queue
  - Each one gets 1/n of 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: FCFS
  - q small: q must be large, considering context switches
    - Q: If there is only one process of 10 time units, how many context switches are there with q = 1, 6, and 12?
    - A: 9, 1, and 0

RR Scheduling with Time Quantum = 4

- Example
  - Process: P_1 P_2 P_3
  - Burst time: 24 3 3
- RR scheduling chart

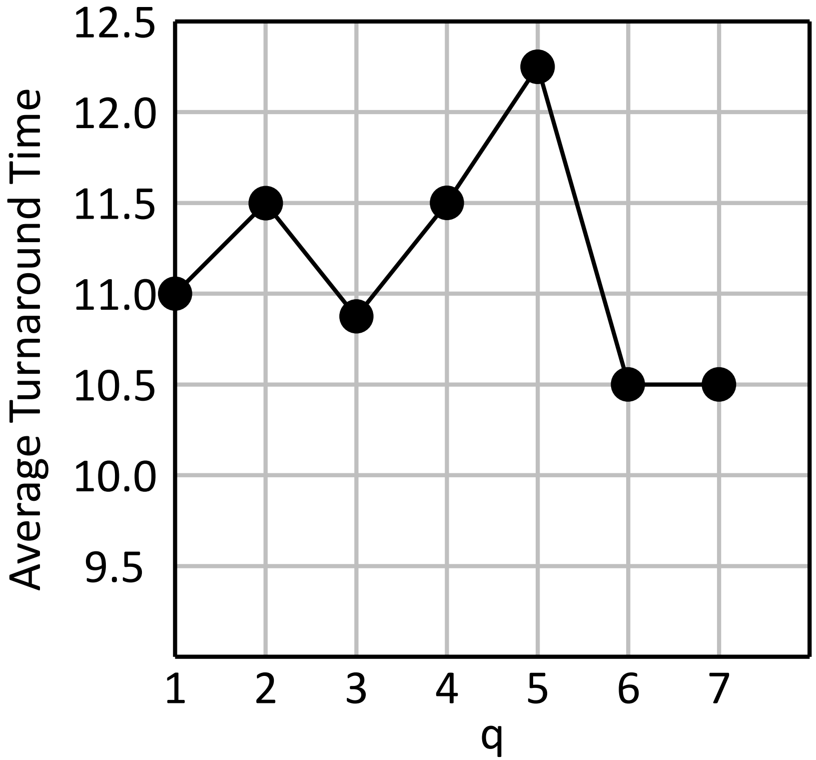
P_1	P_2	P_3	P_1	P_1	P_1	P_1	P_1

0 4 7 10 14 18 22 26 30

- Typically, higher average turnaround than SJF, but better <u>**response**</u>
- q should be large compared to context switch time
  - q is usually 10 milliseconds to 100 milliseconds
  - Context switch < 10 microseconds

Turnaround Time Varies With q

- 80% of CPU bursts should be shorter than q
- Example
  - Process: P_1 P_2 P_3 P_4
  - Burst time: 6 3 1 7



Outline

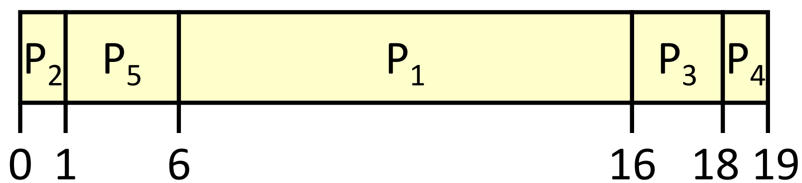
- Basic Concepts
- Scheduling Criteria
- **Scheduling Algorithms**
  - First-Come, First-Served Scheduling
  - Shortest-Job-First Scheduling
  - Round-Robin Scheduling
  - **Priority Scheduling**
  - Multilevel Queue Scheduling
  - Multilevel Feedback Queue Scheduling
- Thread Scheduling, Multi-Processor Scheduling
- Real-Time CPU Scheduling, Operating-System Examples
- Algorithm Evaluation

Priority Scheduling

- A priority number (integer) is associated with each process
  - The CPU is allocated to the process with the highest priority
    - Smallest integer = highest priority
  - Preemptive
  - Nonpreemptive
- SJF is priority scheduling where priority is the inverse of predicted next CPU burst time
- **Starvation**
  - Low priority processes may never execute
- **Aging**
  - As time progresses, increase the priority of the process

Priority Scheduling

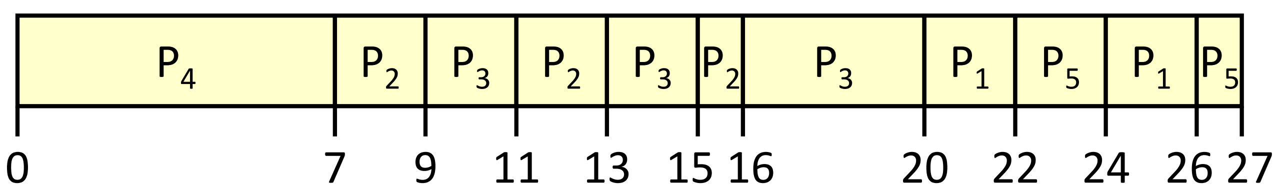
- Example
  - Process: P_1 P_2 P_3 P_4 P_5
  - Burst time: 10 1 2 1 5
  - Priority: 3 1 4 5 2
- Priority scheduling chart



- Average waiting time
  - [6 + 0 + 16 + 18 + 1] / 5 = 8.2

Priority Scheduling w/ Round-Robin

- Example
  - Process: P_1 P_2 P_3 P_4 P_5
  - Burst time: 4 5 8 7 3
  - Priority: 3 2 2 1 3
- Run the process with the highest priority, and processes with the same priority run round-robin
- Scheduling chart with time quantum = 2

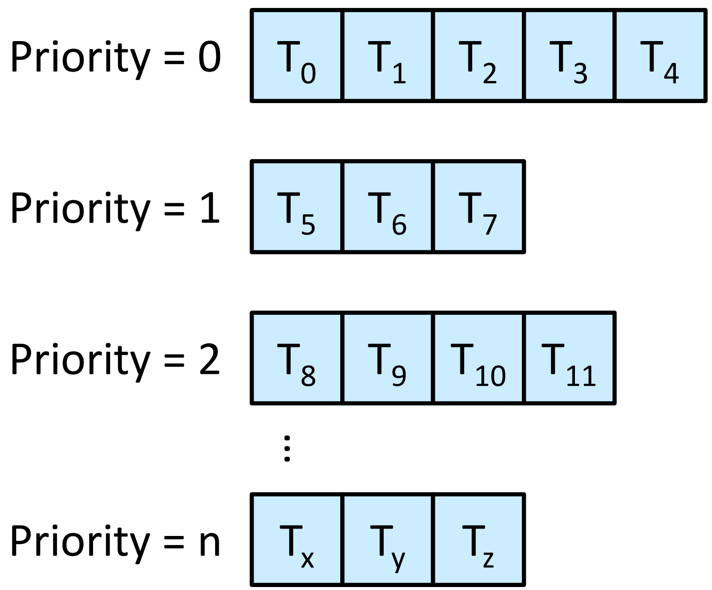


Outline

- Basic Concepts
- Scheduling Criteria
- **Scheduling Algorithms**
  - First-Come, First-Served Scheduling
  - Shortest-Job-First Scheduling
  - Round-Robin Scheduling
  - Priority Scheduling
  - **Multilevel Queue Scheduling**
  - Multilevel Feedback Queue Scheduling
- Thread Scheduling, Multi-Processor Scheduling
- Real-Time CPU Scheduling, Operating-System Examples
- Algorithm Evaluation

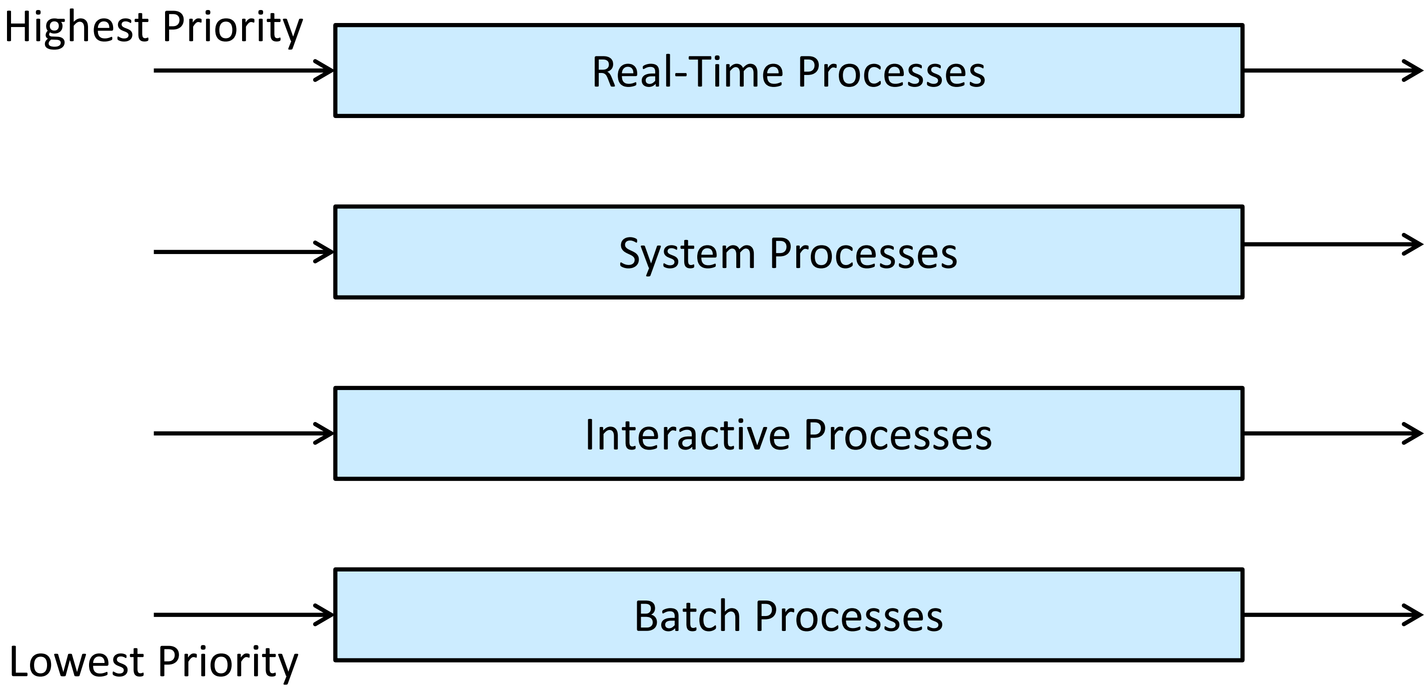
Multilevel Queue Scheduling

- Have separate queues for each priority
- Schedule the process in the highest-priority queue



Multilevel Queue Scheduling

- Prioritization based upon process type
  - Each queue might have its own scheduling algorithm



Outline

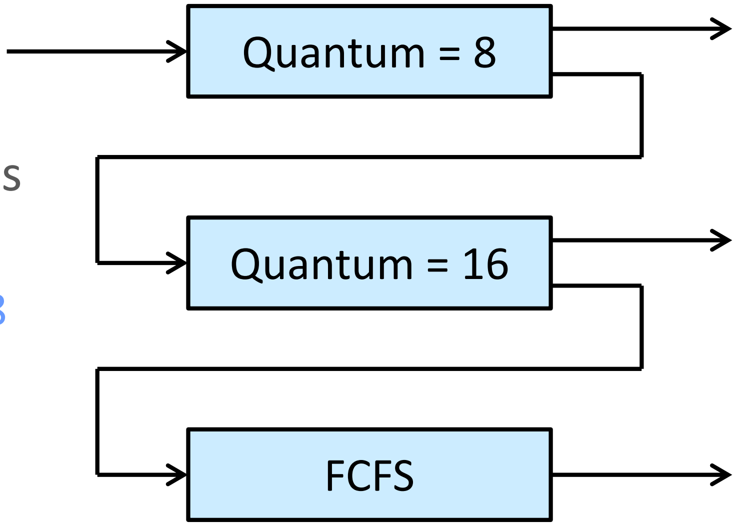
- Basic Concepts
- Scheduling Criteria
- **Scheduling Algorithms**
  - First-Come, First-Served Scheduling
  - Shortest-Job-First Scheduling
  - Round-Robin Scheduling
  - Priority Scheduling
  - Multilevel Queue Scheduling
  - **Multilevel Feedback Queue Scheduling**
- Thread Scheduling, Multi-Processor Scheduling
- Real-Time CPU Scheduling, Operating-System Examples
- Algorithm Evaluation

Multilevel Feedback Queue Scheduling

- A process can move between the various queues
- A multilevel-feedback-queue scheduler is defined by
  - 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
- Aging can be implemented using multilevel feedback queue

Multilevel Feedback Queue Scheduling

- Three queues
  - Q_0: RR with time quantum 8 milliseconds
  - Q_1: RR with time quantum 16 milliseconds
  - Q_2: FCFS
- Scheduling
  - A new process enters queue Q_0 which is served in RR
    - When it gains CPU, the process receives 8 milliseconds
    - If it does not finish in 8 milliseconds, the process is moved to queue Q_1
  - At queue Q_1, it is again served in RR and receives 16 additional milliseconds
    - If it still does not complete, it is preempted and moved to queue Q_2



Outline

- Basic Concepts
- Scheduling Criteria
- Scheduling Algorithms
- **Thread Scheduling**
  - Contention Scope
  - Pthread Scheduling
- Multi-Processor Scheduling
- Real-Time CPU Scheduling
- Operating-System Examples
- Algorithm Evaluation

Thread Scheduling

- On most modern operating systems, it is kernel-level threads, not processes, that are being scheduled
  - Distinction between user-level and kernel-level threads
    - User-level threads are managed by a thread library, and the kernel is unaware of them
    - To run on a CPU, user-level threads must ultimately be mapped to an associated kernel-level thread
    - The mapping may be indirect and may use a lightweight process (LWP)

Outline

- Basic Concepts
- Scheduling Criteria
- Scheduling Algorithms
- **Thread Scheduling**
  - **Contention Scope**
  - Pthread Scheduling
- Multi-Processor Scheduling
- Real-Time CPU Scheduling
- Operating-System Examples
- Algorithm Evaluation

Contention Scope

- **Process-contention scope (PCS)**
  - With the many-to-one and many-to-many models, the thread library "schedules" user-level threads to run on an available LWP
    - Scheduling competition is within the process, typically done via priority set by programmers
- **System-contention scope (SCS)**
  - The kernel uses SCS to decide which kernel-level thread to schedule onto a CPU
    - Scheduling competition is among all threads in the system
    - Systems (such as Windows and Linux) with the one-to-one model schedule threads using SCS only

Outline

- Basic Concepts
- Scheduling Criteria
- Scheduling Algorithms
- **Thread Scheduling**
  - Contention Scope
  - **Pthread Scheduling**
- Multi-Processor Scheduling
- Real-Time CPU Scheduling
- Operating-System Examples
- Algorithm Evaluation

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
- It can be limited by OS
  - Linux and macOS only allow **`PTHREAD_SCOPE_SYSTEM`**

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

```
   /* 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);
}
```

Outline

- Basic Concepts, Scheduling Criteria
- Scheduling Algorithms, Thread Scheduling
- **Multi-Processor Scheduling**
  - Approaches to Multiple-Processor Scheduling
  - Multicore Processors
  - Load Balancing
  - Processor Affinity
  - Heterogeneous Multiprocessing
- Real-Time CPU Scheduling
- Operating-System Examples
- Algorithm Evaluation

Multiple-Processor Scheduling

- CPU scheduling is more complex when multiple CPUs are available
- Multiprocess may be any one of the following architectures
  - Multicore CPUs
  - Multithreaded cores
  - NUMA systems
    - NUMA: Non-Uniform Memory Access
  - Heterogeneous multiprocessing

Outline

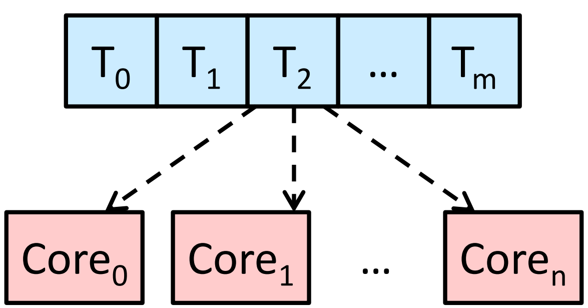
- Basic Concepts, Scheduling Criteria
- Scheduling Algorithms, Thread Scheduling
- <u>**Multi-Processor Scheduling**</u>
  - <u>**Approaches to Multiple-Processor Scheduling**</u>
  - Multicore Processors
  - Load Balancing
  - Processor Affinity
  - Heterogeneous Multiprocessing
- Real-Time CPU Scheduling
- Operating-System Examples
- Algorithm Evaluation

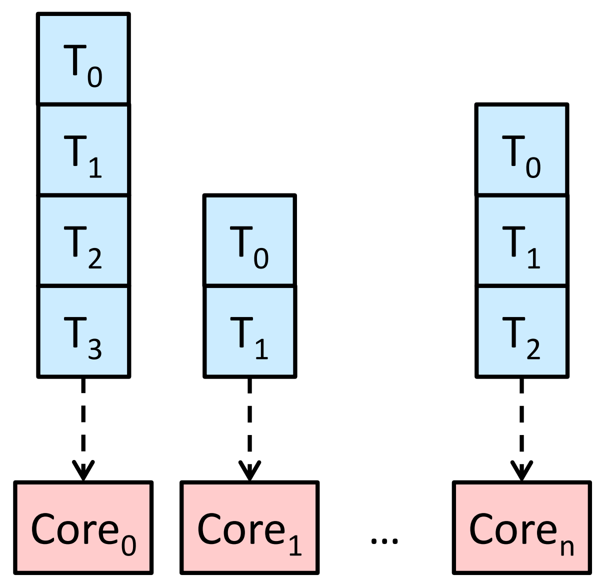
Asymmetric Multiprocessing

- All scheduling decisions, I/O processing, and other system activities are handled by a single processor (master server)
  - The other processors execute only user code
- Advantage
  - Simple: only one core accesses the system data structures, reducing the need for data sharing
- Disadvantage
  - The master server becomes a potential bottleneck where overall system performance may be reduced

Symmetric Multiprocessing (SMP)

- Each processor is self scheduling
  - Scheduling proceeds by having the scheduler for each processor examine the ready queue and select a thread to run
- Two possible strategies
  - All threads may be in a common ready queue
    - Race condition → locking
  - Each processor may have its own private queue of threads
    - Workloads of varying sizes → balancing algorithms



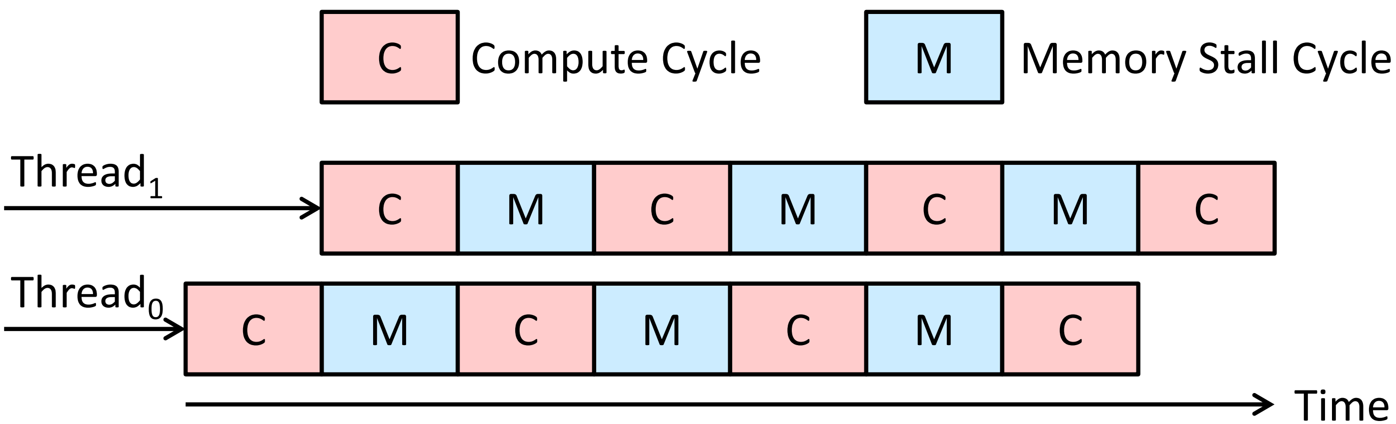


Outline

- Basic Concepts, Scheduling Criteria
- Scheduling Algorithms, Thread Scheduling
- **Multi-Processor Scheduling**
  - Approaches to Multiple-Processor Scheduling
  - **Multicore Processors**
  - Load Balancing
  - Processor Affinity
  - Heterogeneous Multiprocessing
- Real-Time CPU Scheduling
- Operating-System Examples
- Algorithm Evaluation

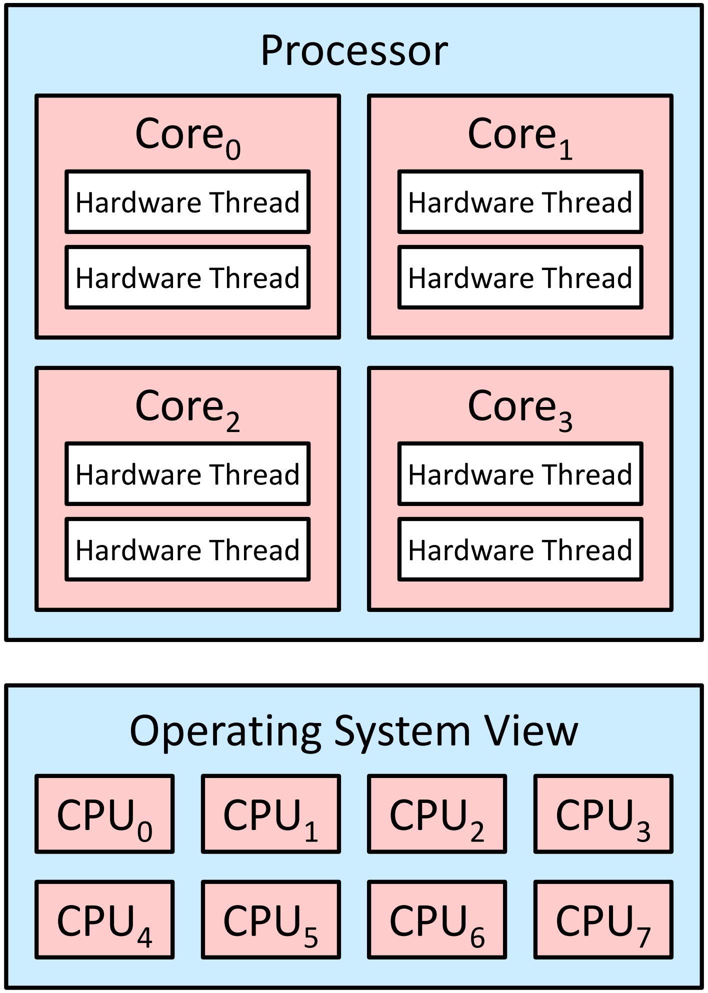
Multithreaded Multicore System (1/3)

- Multicore processers place multiple processor cores on same physical chip
  - Are faster and consumes less power
- Multithreaded multicore system
  - Each core has > 1 hardware threads
  - If one thread has a **memory stall**, switch to another thread
    - Make progress on another thread while memory retrieve happens



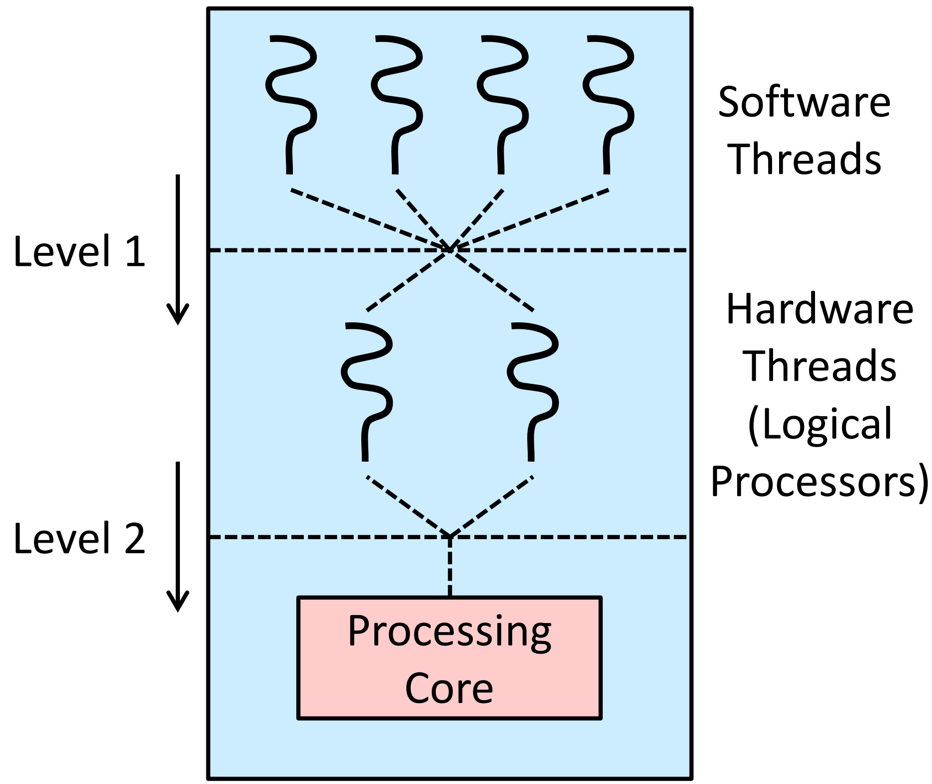
Multithreaded Multicore System (2/3)

- **Chip multithreading (CMT)** assigns each core multiple hardware threads
  - Intel refers to this as **hyper-threading**
- On a quad-core system with 2 hardware threads per core, the operating system sees 8 logical processors



Multithreaded Multicore System (3/3)

- Two levels of scheduling
  - The first level (an operating system) chooses which software thread to run on each hardware thread (logical CPU)
  - The second level specifies how each core decides which hardware thread to run
- They are not necessarily mutually exclusive



Outline

- Basic Concepts, Scheduling Criteria
- Scheduling Algorithms, Thread Scheduling
- **Multi-Processor Scheduling**
  - Approaches to Multiple-Processor Scheduling
  - Multicore Processors
  - **Load Balancing**
  - Processor Affinity
  - Heterogeneous Multiprocessing
- Real-Time CPU Scheduling
- Operating-System Examples
- Algorithm Evaluation

Load Balancing

- On SMP systems, it is important to keep the workload balanced among all processors to fully utilize the benefits of having more than one processor
- **Push migration**
  - A specific task periodically checks the load on each processor
  - If it finds an imbalance, evenly distributes the load by moving (or pushing) threads from overloaded to idle or less-busy processors
- **Pull migration**
  - An idle processor pulls a waiting task from a busy processor

Outline

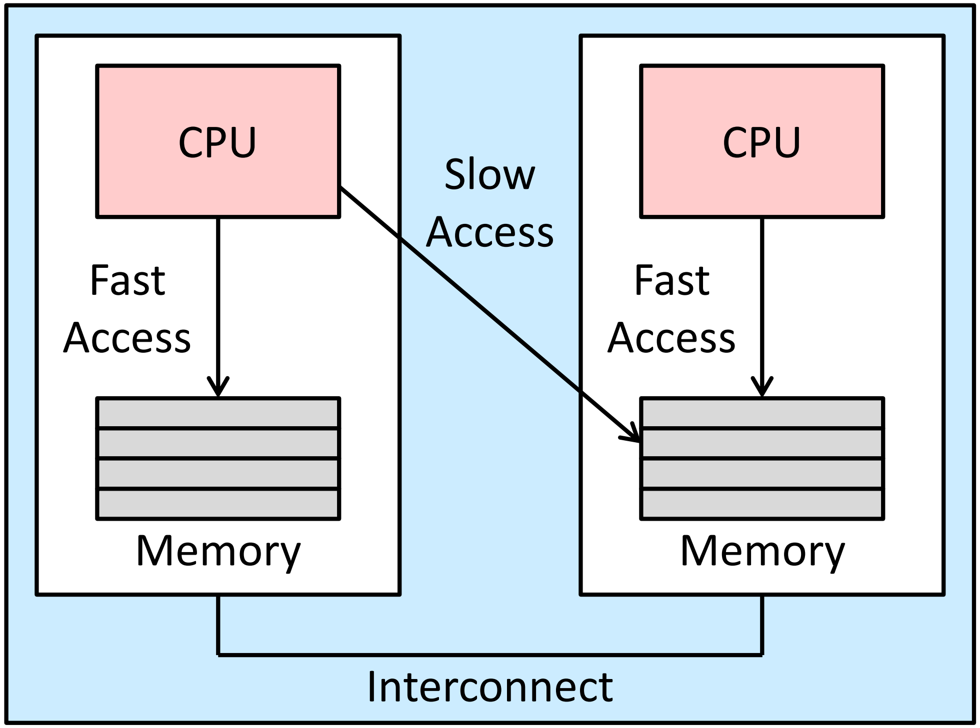
- Basic Concepts, Scheduling Criteria
- Scheduling Algorithms, Thread Scheduling
- **Multi-Processor Scheduling**
  - Approaches to Multiple-Processor Scheduling
  - Multicore Processors
  - Load Balancing
  - **Processor Affinity**
  - Heterogeneous Multiprocessing
- Real-Time CPU Scheduling
- Operating-System Examples
- Algorithm Evaluation

Processor Affinity

- A process has an affinity for the processor on which it is currently running
  - When a thread has been running on a processor, the data most recently accessed by the thread populate the cache for the processor
- Load balancing may affect processor affinity
  - A thread may be moved from one processor to another to balance loads, but it loses the contents in the cache of the processor
- **Soft affinity**
  - Attempt to keep a thread running on the same processor
    - No guarantee
- **Hard affinity**
  - Allow a process 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 close to the CPU that the thread is running on
  - NUMA: Non-Uniform Memory Access



Outline

- Basic Concepts, Scheduling Criteria
- Scheduling Algorithms, Thread Scheduling
- **Multi-Processor Scheduling**
  - Approaches to Multiple-Processor Scheduling
  - Multicore Processors
  - Load Balancing
  - Processor Affinity
  - **Heterogeneous Multiprocessing**
- Real-Time CPU Scheduling
- Operating-System Examples
- Algorithm Evaluation

Heterogeneous Multiprocessing (HMP)

- Some mobile systems are designed using cores that vary in terms of their clock speed and power management
- ARM **big.LITTLE** architecture
  - **Big** cores consume greater energy and therefore should only be used for short periods of time
  - **Little** cores use less energy and can therefore be used for longer periods

Outline

- Basic Concepts, Scheduling Criteria
- Scheduling Algorithms, Thread Scheduling
- Multi-Processor Scheduling
- **Real-Time CPU Scheduling**
  - Minimizing Latency
  - Priority-Based Scheduling
  - Rate-Monotonic Scheduling
  - Earliest-Deadline-First Scheduling
  - Proportional Share Scheduling
  - POSIX Real-Time Scheduling
- Operating-System Examples
- Algorithm Evaluation

Real-Time CPU Scheduling

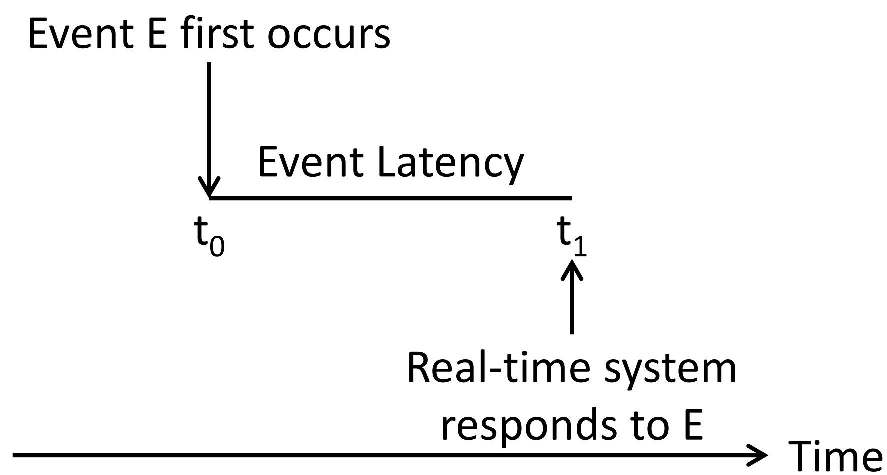
- **Soft real-time systems**
  - Critical real-time tasks have the highest priority, but no guarantee as to when tasks will be scheduled
- **Hard real-time systems**
  - A task must be serviced by its deadline
  - Service after the deadline has expired is the same as no service at all

Outline

- Basic Concepts, Scheduling Criteria
- Scheduling Algorithms, Thread Scheduling
- Multi-Processor Scheduling
- **Real-Time CPU Scheduling**
  - **Minimizing Latency**
  - Priority-Based Scheduling
  - Rate-Monotonic Scheduling
  - Earliest-Deadline-First Scheduling
  - Proportional Share Scheduling
  - POSIX Real-Time Scheduling
- Operating-System Examples
- Algorithm Evaluation

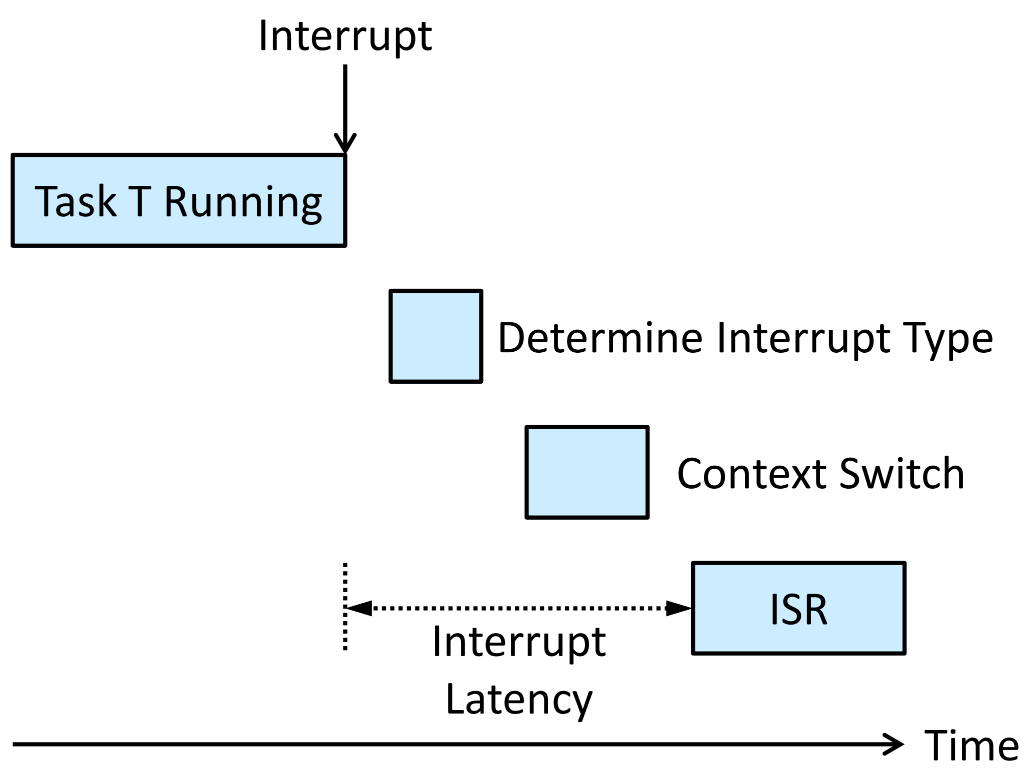
Minimizing Latency

- Event latency
  - The amount of time that elapses
    - From when an event occurs
    - To when it is serviced
- Two types of latencies affect performance
  - **<u>Interrupt latency</u>**
    - Time from arrival of interrupt to start of routine that services interrupt
  - **<u>Dispatch latency</u>**
    - Time for schedule to take current process off CPU and switch to another



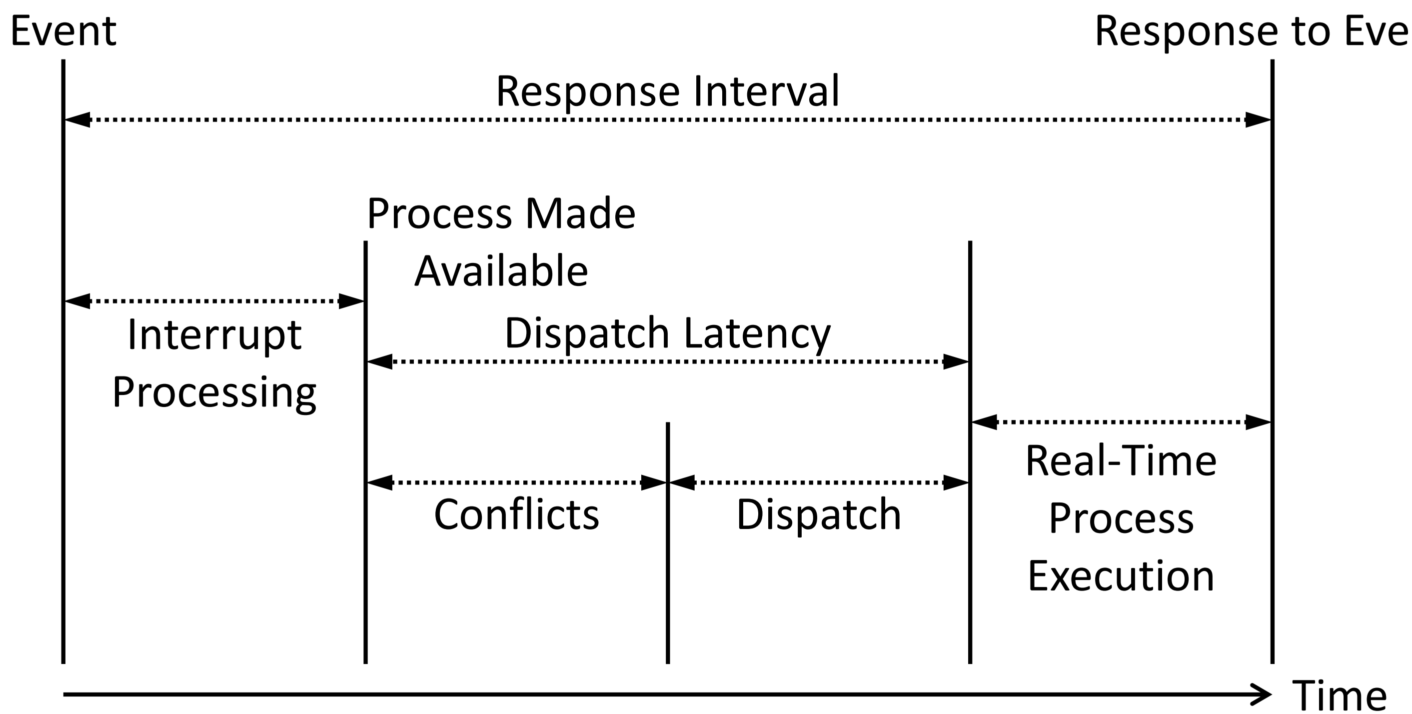
Interrupt Latency

- The amount of time
  - From the arrival of an interrupt at the CPU
  - To the start of the routine that services the interrupt
    - ISR: Interrupt Service Routine



Dispatch Latency

- The amount of time required for the scheduling dispatcher to stop one process and start another
  - Conflict phase
    - Preemption of any process running in kernel mode
    - Release by low-priority process of resources needed by high-priority processes

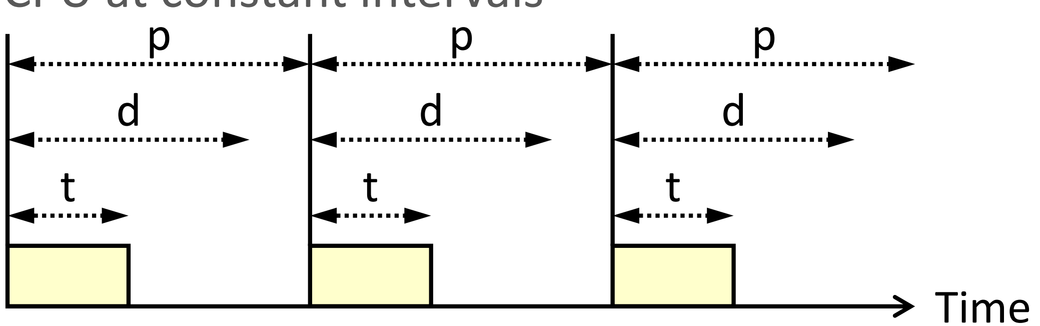


Outline

- Basic Concepts, Scheduling Criteria
- Scheduling Algorithms, Thread Scheduling
- Multi-Processor Scheduling
- **Real-Time CPU Scheduling**
  - Minimizing Latency
  - **Priority-Based Scheduling**
  - Rate-Monotonic Scheduling
  - Earliest-Deadline-First Scheduling
  - Proportional Share Scheduling
  - POSIX Real-Time Scheduling
- Operating-System Examples
- Algorithm Evaluation

Priority-Based Scheduling

- For real-time scheduling, a scheduler must support a priority-based algorithm with preemption
  - Only guarantees soft real-time functionality
- Hard real-time systems must provide ability to meet deadlines
- New process characteristics
  - **Periodic** processes require CPU at constant intervals
    - Processing time = t
    - Deadline = d
    - Period = p
    - $0 \le t \le d \le p$
    - **Rate** of periodic task is 1/p

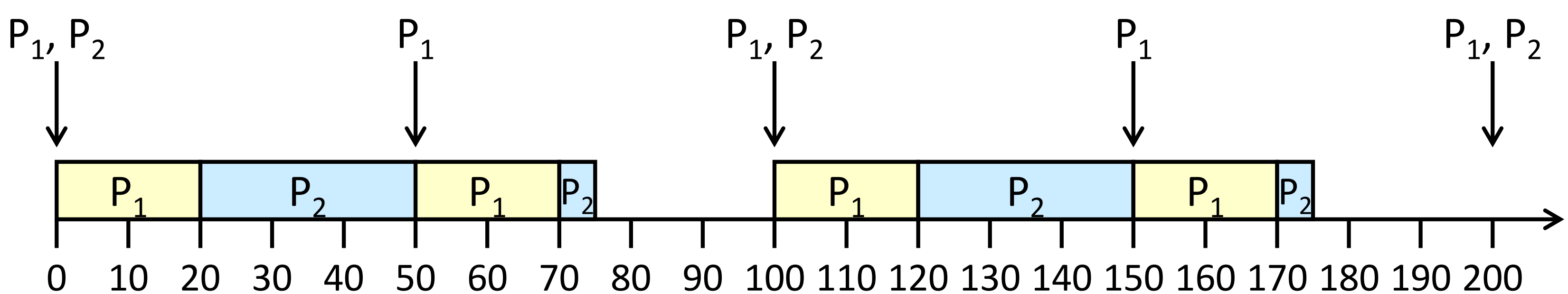


Outline

- Basic Concepts, Scheduling Criteria
- Scheduling Algorithms, Thread Scheduling
- Multi-Processor Scheduling
- **Real-Time CPU Scheduling**
  - Minimizing Latency
  - Priority-Based Scheduling
  - **Rate-Monotonic Scheduling**
  - Earliest-Deadline-First Scheduling
  - Proportional Share Scheduling
  - POSIX Real-Time Scheduling
- Operating-System Examples
- Algorithm Evaluation

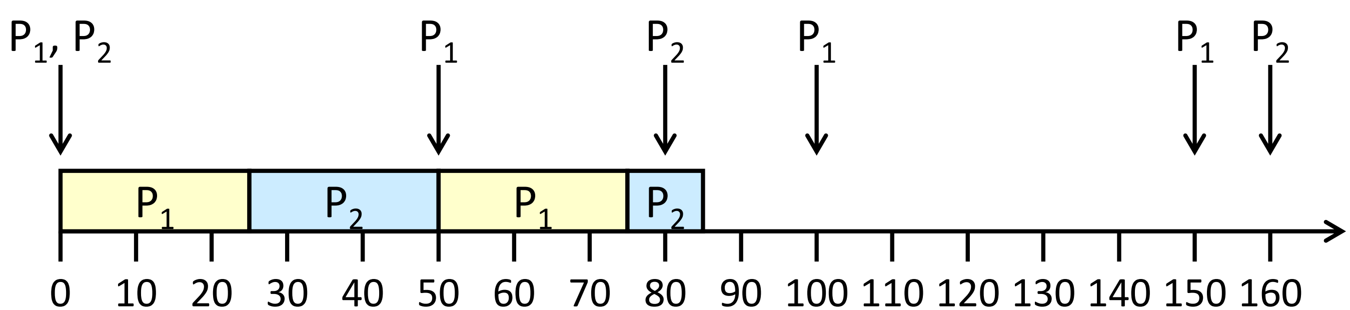
Rate-Monotonic Scheduling (1/2)

- A priority is assigned based on the inverse of its period
  - Shorter periods = higher priority
  - Longer periods = lower priority
- Example
  - Process: P_1 P_2
  - t: 20 35
  - d (= p): 50 100
  - P_1 is assigned a higher priority than P_2



Rate-Monotonic Scheduling (2/2)

- A priority is assigned based on the inverse of its period
  - Shorter periods = higher priority
  - Longer periods = lower priority
- Example
  - Process: P_1 P_2
  - t: 25 35
  - d (= p): 50 80
  - P_1 is assigned a higher priority than P_2
  - Process P_2 misses finishing its deadline at time 80
    - Even if the utilization is smaller than 1

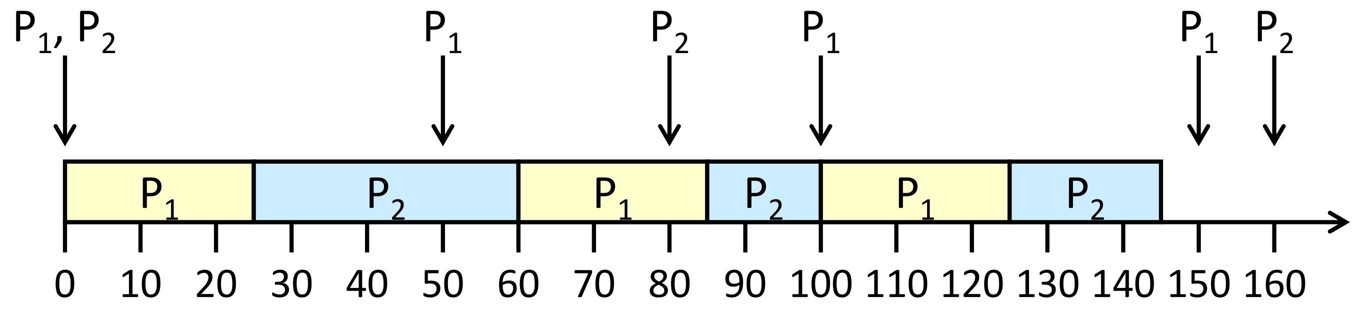


Outline

- Basic Concepts, Scheduling Criteria
- Scheduling Algorithms, Thread Scheduling
- Multi-Processor Scheduling
- **Real-Time CPU Scheduling**
  - Minimizing Latency
  - Priority-Based Scheduling
  - Rate-Monotonic Scheduling
  - **Earliest-Deadline-First Scheduling**
  - Proportional Share Scheduling
  - POSIX Real-Time Scheduling
- Operating-System Examples
- Algorithm Evaluation

Earliest-Deadline-First (EDF) Scheduling

- Priorities are assigned according to deadlines
  - The earlier the deadline, the higher the priority
  - The later the deadline, the lower the priority
- Example
  - Process: P_1 P_2
  - t: 25 35
  - d (= p): 50 80



Outline

- Basic Concepts, Scheduling Criteria
- Scheduling Algorithms, Thread Scheduling
- Multi-Processor Scheduling
- **Real-Time CPU Scheduling**
  - Minimizing Latency
  - Priority-Based Scheduling
  - Rate-Monotonic Scheduling
  - Earliest-Deadline-First Scheduling
  - **Proportional Share Scheduling**
  - POSIX Real-Time Scheduling
- Operating-System Examples
- Algorithm Evaluation

Proportional Share Scheduling

- T shares are allocated among all processes in the system, and a process i receives N_i shares where $N_i < T$
  - Ensure that the process receives N_i / T of the total processor time
- Proportional share schedulers must work in conjunction with an admission-control policy
- Example
  - T = 100
  - Processes A, B, and C are assigned 50, 15, and 20 shares, respectively
  - A new process D requesting 30 shares will be rejected by the admission controller

Outline

- Basic Concepts, Scheduling Criteria
- Scheduling Algorithms, Thread Scheduling
- Multi-Processor Scheduling
- **Real-Time CPU Scheduling**
  - Minimizing Latency
  - Priority-Based Scheduling
  - Rate-Monotonic Scheduling
  - Earliest-Deadline-First Scheduling
  - Proportional Share Scheduling
  - **POSIX Real-Time Scheduling**
- Operating-System Examples
- Algorithm Evaluation

POSIX Real-Time Scheduling

- The POSIX.1b standard
- API provides functions for managing real-time threads
- Two scheduling classes for real-time threads
  - **`SCHED_FIFO`**
    - Threads of equal priority are scheduled with a FCFS strategy and a FIFO queue
  - **`SCHED_RR`**
    - Similar to **`SCHED_FIFO`**, except time-slicing occurs for threads of equal priority
- Two functions for getting and setting scheduling policy
  - **`pthread_attr_getschedpolicy(pthread_attr_t *attr, int *policy)`**
  - **`pthread_attr_setschedpolicy(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

```
    /* 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);
}
```

Outline

- ❑ Basic Concepts
- ❑ Scheduling Criteria
- ❑ Scheduling Algorithms
- ❑ Thread Scheduling
- ❑ Multi-Processor Scheduling
- ❑ Real-Time CPU Scheduling
- ❑ **Operating-System Examples**
  - ➢ **Linux Scheduling**
  - ➢ Windows scheduling
  - ➢ Solaris scheduling
- ❑ Algorithm Evaluation

Linux Scheduling Through Version 2.5

- Prior to Version 2.5, the Linux kernel ran a variation of the traditional UNIX scheduling algorithm
- Version 2.5 moves to constant order O(1) scheduling time
  - Worked well, but poor response times for interactive processes that are common on many desktop computer systems

Linux Scheduling in Version 2.6.23 +

- **Completely Fair Scheduler (CFS)** became the default Linux scheduling algorithm
- Scheduling based on **scheduling classes**
  - Each class is assigned a specific priority
  - The kernel can accommodate different scheduling algorithms based on the needs of the system and its processes
    - The scheduling criteria for a Linux server, for example, may be different from those for a mobile device running Linux
  - The scheduler selects the highest-priority task belonging to the highest-priority scheduling class
  - Standard Linux kernels implement two scheduling classes
    - A default scheduling class using the CFS scheduling algorithm
    - A real-time scheduling class

Linux Default Scheduling

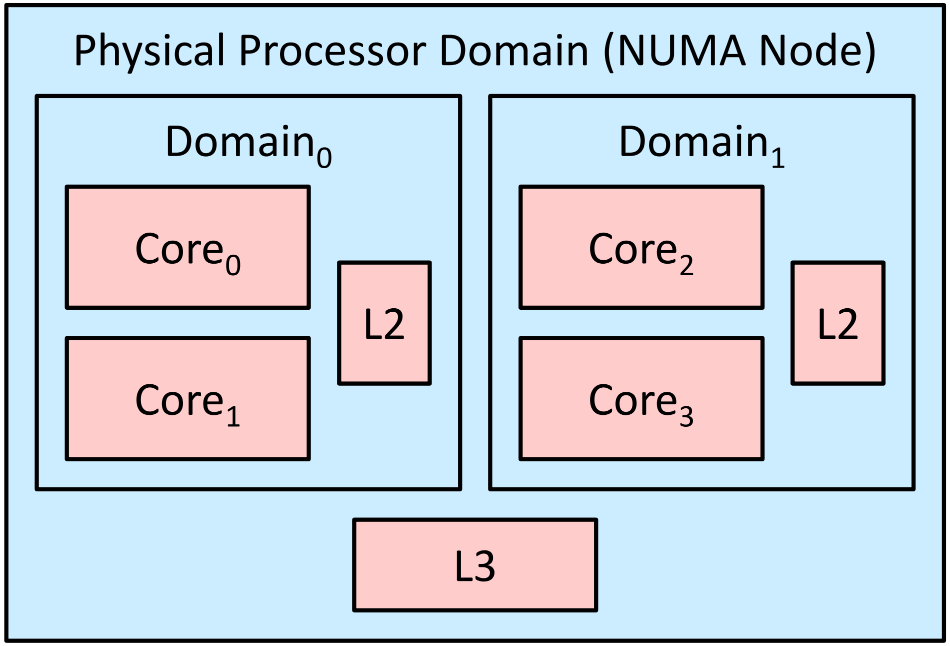
- Quantum is calculated based on **nice value** from -20 to +19
  - Calculate **targeted latency**, an interval of time during which every runnable task should run at least once
    - Allocate proportions of CPU time from the value of targeted latency
    - Increase the value of targeted latency if the number of active tasks in the system grows beyond a threshold
- CFS scheduler maintains the **virtual run time** of each task using the per-task variable **`vruntime`**
  - Place each runnable task in a red-black tree
    - A balanced binary search tree whose key is based on the value of **`vruntime`**
  - Pick the task with the lowest **`vruntime`** to decide the next task
    - If a lower-priority (higher-priority) task runs for 200 milliseconds, its **`vruntime`** is higher (less) than 200 milliseconds

Linux Real-Time Scheduling

- Real-time scheduling according to POSIX.1b
  - Real-time tasks are assigned static priorities within the range of 0 to 99
  - Normal tasks are assigned priorities from 100 to 139
    - Nice value of -20 maps to global priority 100
    - Nice value of +19 maps to global priority 139
  - Smaller integer = higher priority

Linux NUMA-Aware Load Balancing

- **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)
  - The goal is to keep threads from migrating between domains



Outline

- Basic Concepts
- Scheduling Criteria
- Scheduling Algorithms
- Thread Scheduling
- Multi-Processor Scheduling
- Real-Time CPU Scheduling
- **Operating-System Examples**
  - Linux Scheduling
  - **Windows scheduling**
  - Solaris scheduling
- Algorithm Evaluation

Windows Scheduling

- Windows schedules threads using a priority-based, preemptive scheduling algorithm
  - The **dispatcher** handles scheduling
  - A thread selected to run by the dispatcher will run until
    - It is preempted by a higher-priority thread
    - It terminates
    - Its time quantum ends
    - It calls a blocking system call, such as for I/O
- 32-level priority to determine the order of thread execution
  - The **variable class** is from 1 to 15; the **real-time class** is from 16 to 31
  - Priority 0 is memory-management thread
  - Dispatcher
    - Use a queue for each scheduling priority
    - Execute a special thread called the **idle thread**, if no ready thread is found

Windows Priorities (1/2)

- The Windows API identifies the following classes to which a thread 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_PRIORITY_CLASS**
- A thread within a given priority class has a relative priority
  - **TIME_CRITICAL, HIGHEST, ABOVE_NORMAL, NORMAL, BELOW_NORMAL, LOWEST, IDLE**
- The priority of each thread is based on both the priority class it belongs to and its relative priority within that class
  - The base priority is **NORMAL** within the class
  - If quantum expires, the priority is lowered but never below the base

Windows Priorities (2/2)

	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

Windows Priority Classes

- Windows has a special scheduling rule for processes in the **`NORMAL_PRIORITY_CLASS`**
  - When a process moves into the **foreground**, Windows increases the scheduling quantum by some factor, typically by 3
- Windows 7 added **user-mode scheduling (UMS)**
  - Allow applications to create and manage threads independently of the kernel
    - For applications that create a large number of threads, scheduling threads in user mode is much more efficient than that in kernel mode
  - UMS schedulers come from programming language libraries like C++ Concurrent Runtime (ConcRT) framework

Outline

- Basic Concepts
- Scheduling Criteria
- Scheduling Algorithms
- Thread Scheduling
- Multi-Processor Scheduling
- Real-Time CPU Scheduling
- **Operating-System Examples**
  - Linux Scheduling
  - Windows scheduling
  - **Solaris scheduling**
- Algorithm Evaluation

Solaris Scheduling (1/3)

- Priority-based scheduling
  - Six classes available
    - Real time (RT)
    - System (SYS)
    - Fair share (FSS)
    - Fixed priority (FP)
    - Time sharing (TS)
    - Interactive (IA)
  - Within each class, there are different priorities and scheduling algorithms
  - The default class for a process is time sharing
    - It uses a multilevel feedback queue

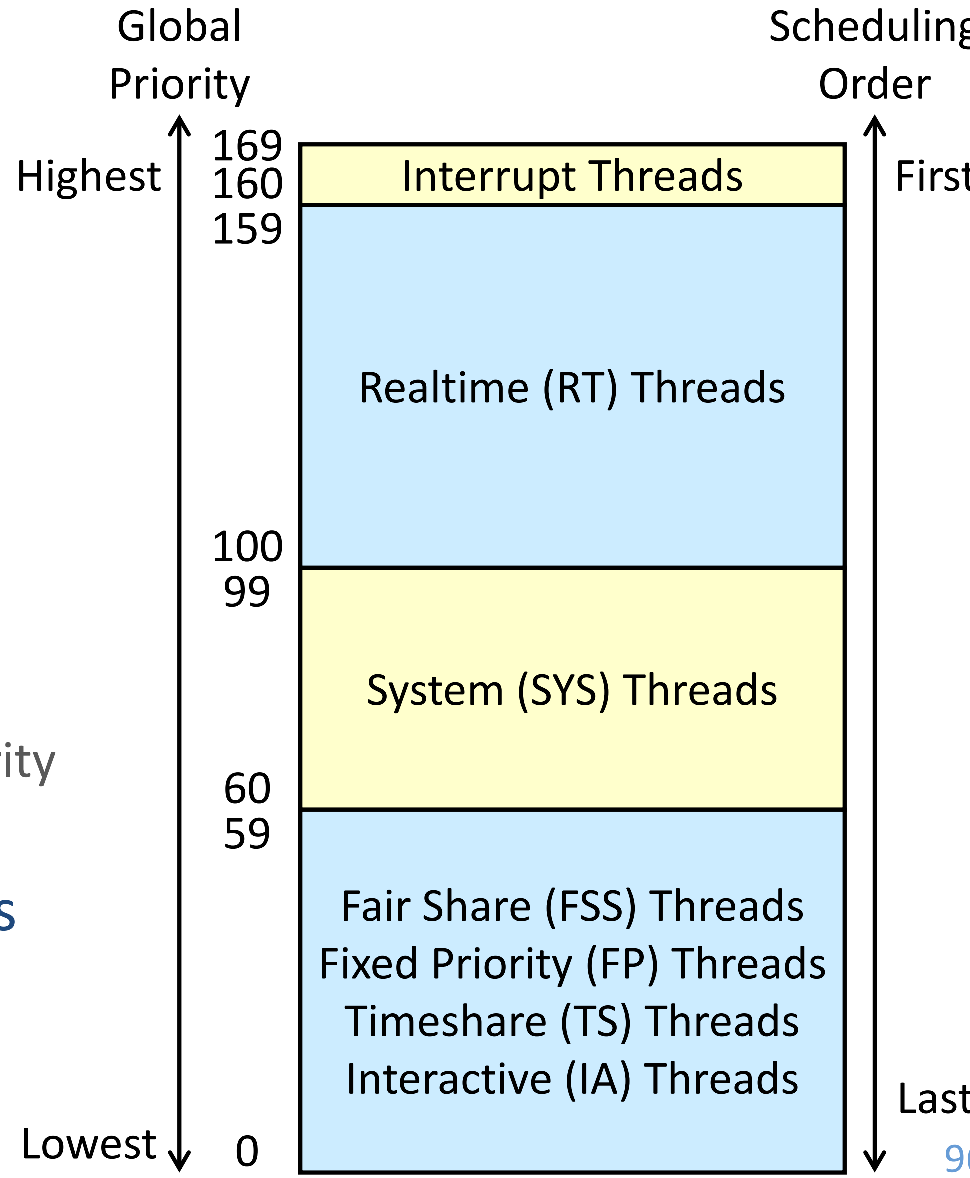
Solaris Scheduling (2/3)

- Solaris dispatch table for time-sharing and interactive threads

Priority	Time Quantum	Priority after Time Quantum Expired	Priority after Returning 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

Solaris Scheduling (3/3)

- The scheduler
  - Convert the class-specific priorities into global priorities
  - Select the thread with the highest global priority to run
- The selected thread runs on the CPU until it
  - Blocks
  - Uses its time slice
  - Is preempted by a higher-priority thread
- RR for same-priority threads



Outline

- Basic Concepts
- Scheduling Criteria
- Scheduling Algorithms
- Thread Scheduling
- Multi-Processor Scheduling
- Real-Time CPU Scheduling
- Operating-System Examples
- **Algorithm Evaluation**
  - Deterministic Modeling
  - Queueing Models
  - Simulations
  - Implementation

Algorithm Evaluation

- How to select CPU-scheduling algorithms for an OS?
  - Determine criteria
  - Then evaluate algorithms

Outline

- Basic Concepts
- Scheduling Criteria
- Scheduling Algorithms
- Thread Scheduling
- Multi-Processor Scheduling
- Real-Time CPU Scheduling
- Operating-System Examples
- **Algorithm Evaluation**
  - **Deterministic Modeling**
  - Queueing Models
  - Simulations
  - Implementation

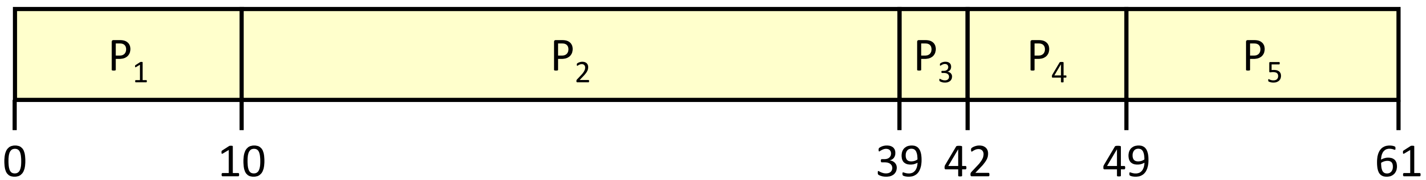
Deterministic Modeling (1/2)

- A type of **analytic evaluation**
  - Take a particular predetermined workload
  - Define the performance of each algorithm for that workload
- Example

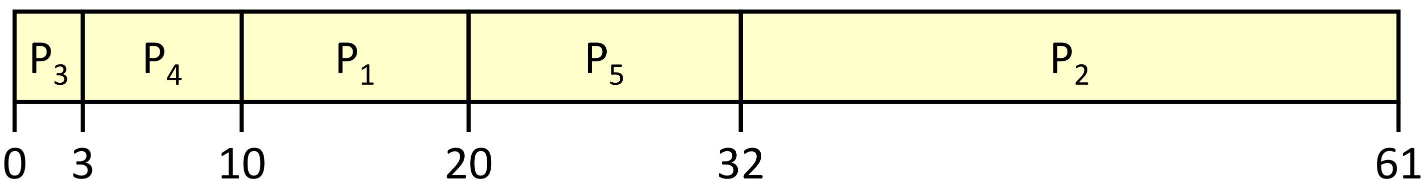
Process:	P_1	P_2	P_3	P_4	P_5
Arrival time:	0	0	0	0	0
Burst time:	10	29	3	7	12

Deterministic Modeling (2/2)

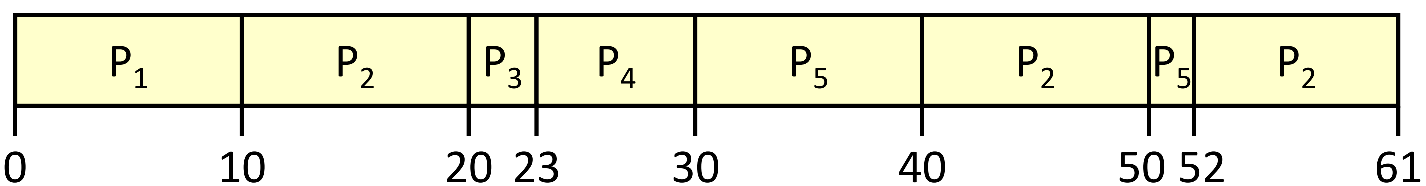
- **Calculate minimum average waiting time for each algorithm**
  - Simple and fast
  - Require exact numbers for input and apply only to those inputs
- **Example**
  - FCFS is 28ms



  - Nonpreemptive SJF is 13ms



  - RR is 23ms



Outline

- Basic Concepts
- Scheduling Criteria
- Scheduling Algorithms
- Thread Scheduling
- Multi-Processor Scheduling
- Real-Time CPU Scheduling
- Operating-System Examples
- **Algorithm Evaluation**
  - Deterministic Modeling
  - **Queueing Models**
  - Simulations
  - Implementation

Queueing Models

- Describe the arrival of processes as well as CPU and I/O bursts probabilistically
  - Commonly exponential, and described by mean
- Compute average throughput, utilization, waiting time, etc.
  - A computer system is described as a network of servers
  - Each server has a queue of waiting processes
    - The CPU is a server with its ready queue
    - The I/O system is a server with its device queues

Little's Formula

- Parameters
  - n = average queue length
  - λ = average arrival rate into queue
  - W = average waiting time in queue
- Little's law: $n = \lambda \times W$
  - In a steady state, processes leaving queue must equal processes arriving
  - Valid for any scheduling algorithm and arrival distribution
- Example
  - If
    - Average 7 process arrivals per second
    - Average 14 processes in queue
  - Then
    - Average wait time per process = 2 seconds

Outline

- Basic Concepts
- Scheduling Criteria
- Scheduling Algorithms
- Thread Scheduling
- Multi-Processor Scheduling
- Real-Time CPU Scheduling
- Operating-System Examples
- **Algorithm Evaluation**
  - Deterministic Modeling
  - Queueing Models
  - **Simulations**
  - Implementation

Simulations

- Queueing models are limited
- Simulations are more accurate
  - Programming a model of the computer system
  - Software data structures representing the major components of the system
  - A variable representing a clock
- Data to drive simulation are gathered via
  - Random number generator according to probabilities
  - Distributions defined mathematically or empirically
  - Trace tapes record sequences of real events in real systems

Simulations

- Actual process execution
  - Trace tape
    - …
    - CPU 10
    - I/O 213
    - CPU 12
    - I/O 112
    - CPU 2
    - I/O 147
    - CPU 173
    - …
- Simulations with FCFS, SJF, RR (q = 14), …
- Performance statistics for FCFS, SJF, RR (q = 14), …

Outline

- Basic Concepts
- Scheduling Criteria
- Scheduling Algorithms
- Thread Scheduling
- Multi-Processor Scheduling
- Real-Time CPU Scheduling
- Operating-System Examples
- **Algorithm Evaluation**
  - Deterministic Modeling
  - Queueing Models
  - Simulations
  - **Implementation**

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)
  - Again, environments vary

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

Q&A

Practice Exercise 1

- Example
  - Process: P_1 P_2 P_3
  - Arrival time: 0.0 0.4 1.0
  - Burst time: 8 4 1
- Use nonpreemptive scheduling
  - What is the average turnaround time for these processes with the FCFS scheduling algorithm?
  - What is the average turnaround time for these processes with the SJF scheduling algorithm?
  - Is SJF scheduling algorithm the optimal one? Why?

Practice Exercise 1

- **Example**
  - Process: P_1 P_2 P_3
  - Arrival time: 0.0 0.4 1.0
  - Burst time: 8 4 1
- **Use nonpreemptive scheduling**
  - What is the average turnaround time for these processes with the FCFS scheduling algorithm?
    - P_1, P_2, P_3 → ((8 - 0) + (12 - 0.4) + (13 - 1)) / 3 = 10.53
  - What is the average turnaround time for these processes with the SJF scheduling algorithm?
    - P_1, P_3, P_2 → ((8 - 0) + (9 - 1) + (13 - 0.4)) / 3 = 9.53
  - Is SJF scheduling algorithm the optimal one? Why?
    - P_2, P_3, P_1 → ((4.4 - 0.4) + (5.4 - 1) + (13.4 - 0)) / 3 = 7.27
    - P_3, P_2, P_1 → ((2 - 1) + (6 - 0.4) + (14 - 0)) / 3 = 6.87

Practice Exercise 2

- We have actually introduced a set of scheduling algorithms
- What is the set relation between the following pairs of algorithms (or algorithm sets)?
  - Priority and FCFS
  - RR and FCFS
  - Multilevel feedback queues and FCFS
  - Priority and SJF
  - RR and SJF