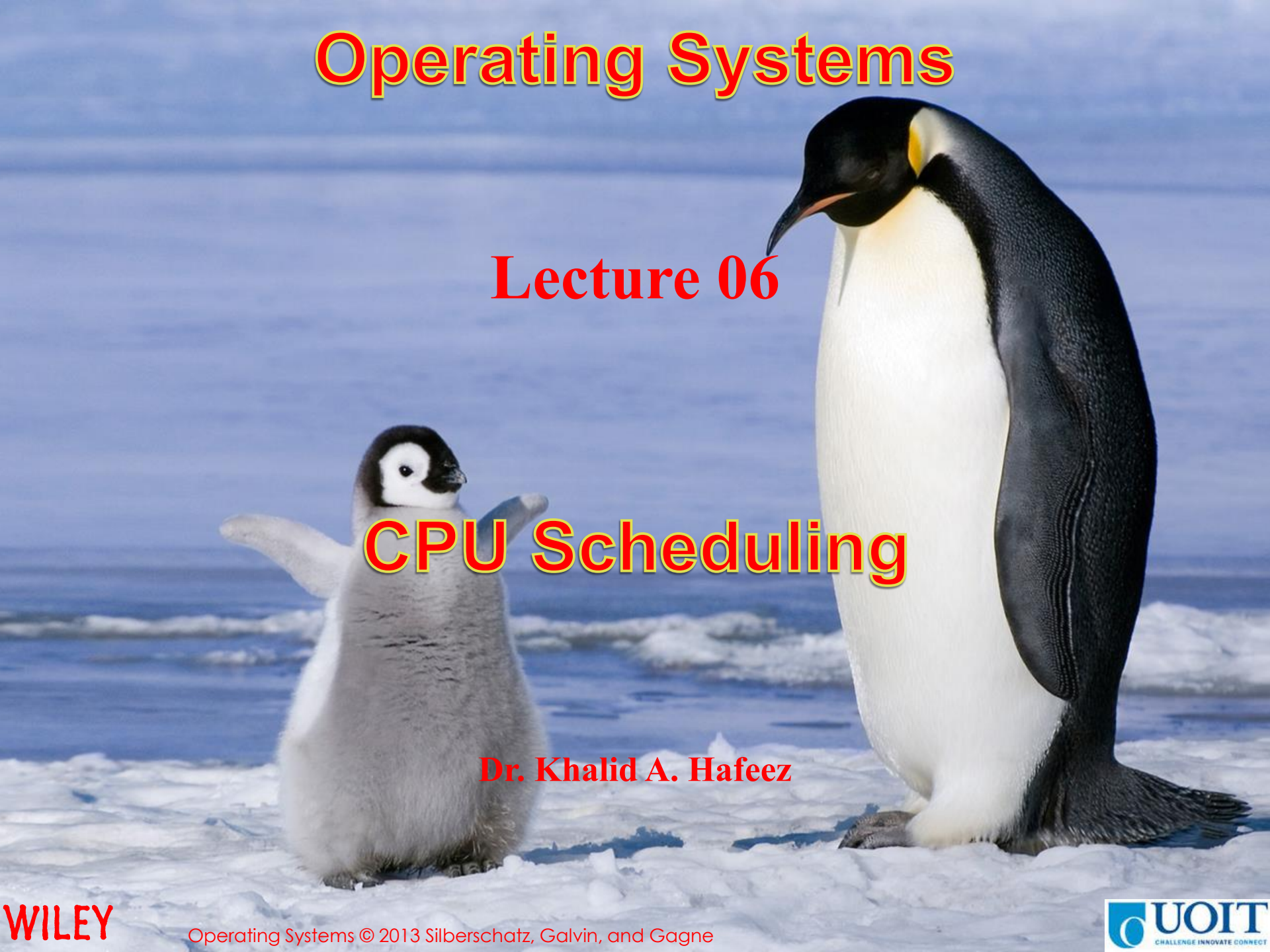
Operating Systems

Lecture 06

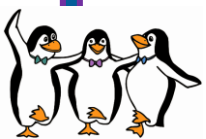
CPU Scheduling

Dr. Khalid A. Hafeez



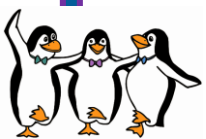
CPU Scheduling

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



Objectives

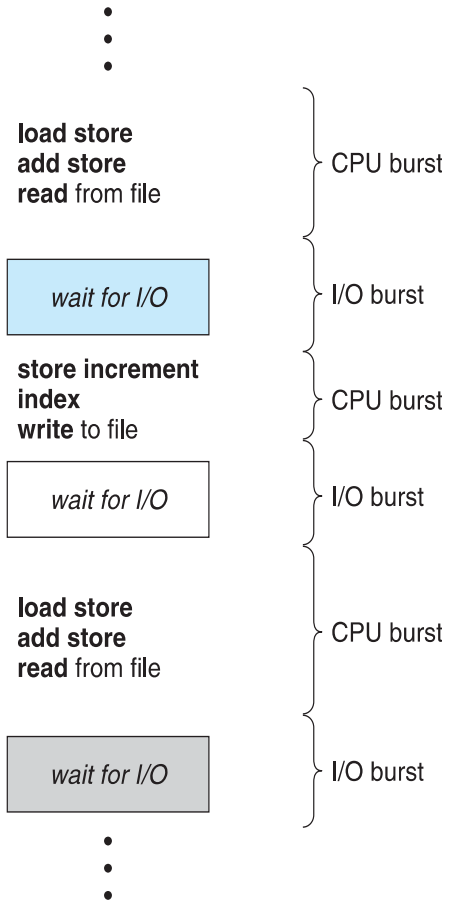
- To introduce CPU scheduling, which is the basis for multiprogrammed operating systems
- To describe various CPU-scheduling algorithms
- To discuss evaluation criteria for selecting a CPU-scheduling algorithm for a particular system
- To examine the scheduling algorithms of several operating systems

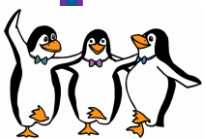


Basic Concepts

- Maximum CPU utilization obtained with multiprogramming
  - Several processes are kept in memory at one time
  - When a process has to wait, another process will use the CPU.

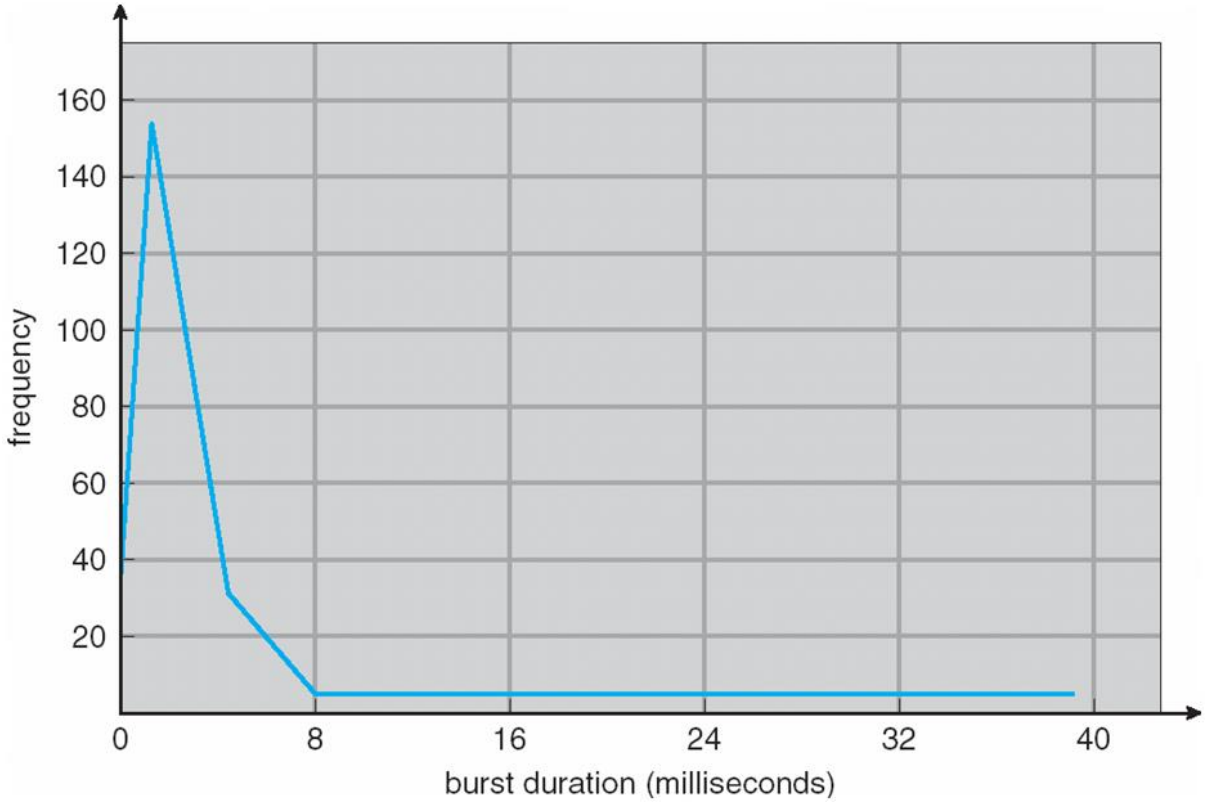
- **CPU–I/O Burst Cycle**
  - Process execution consists of a **cycle** of CPU execution and I/O wait
  - Process execution begins with a **CPU burst** followed by **I/O burst**
- CPU burst distribution is of our main concern





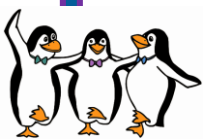
Basic Concepts

- Histogram of CPU-burst Times
  - The curve is generally characterized as exponential or hyperexponential, with a large number of short CPU bursts and 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.



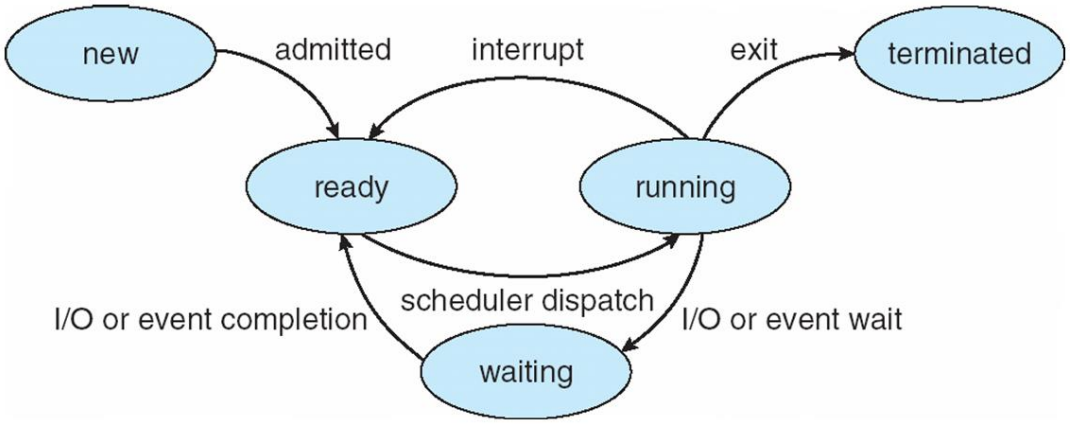
Basic Concepts

- CPU Scheduler
  - When CPU is idle, the operating system must select one of the processes in the ready queue to be executed
  - The **Short-term scheduler (CPU scheduler)** selects from among the processes in ready queue, and allocates the CPU to one of them
    - **Queue may be ordered in various ways:**
      - FIFO queue,
      - Priority queue,
      - Tree, or an unordered linked list.



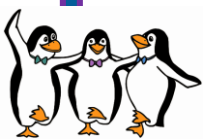
Basic Concepts

- Preemptive Scheduling
  - **CPU-scheduling decisions may take place under the following four circumstances:**
    1. When a process switches from running to waiting state
       - For an I/O request or an invocation of wait() for the termination of a child process
    2. When a process switches from running to ready state
       - Because of an interrupt
    3. When a process switches from waiting to ready
       - When it finishes an I/O operation
    4. When a process terminates



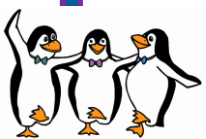
Basic Concepts

- Preemptive Scheduling
  - Scheduling under situations 1 and 4 is **nonpreemptive (cooperative):**
    - There is no choice in terms of scheduling
      - Once the CPU has been allocated to a process, it will keep it until it terminates or switches to the waiting state
  - Situations 3 and 4 are **preemptive scheduling** (there is a choice)
    - Can result in race conditions:
      - Consider the case of two processes that share data.
        - While one process is updating the data is been pre-empted, then the data is inconsistent state
      - Consider preemption while in kernel mode
        - The kernel may be executing a system call that need to modify some important data structure (queue), what will happen if the process been served is preempted during this system call.
      - Consider interrupts occurring during crucial OS activities



Basic Concepts

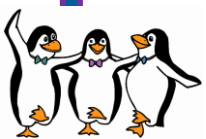
- Dispatcher
  - Dispatcher module gives control of the CPU to the process selected by the short-term scheduler; this involves:
    - Switching context
    - Switching to user mode
    - Jumping to the proper location in the user program to restart that program
  - The dispatcher should be as fast as possible, since it is invoked during every process switch.
    - **Dispatch latency:** time it takes for the dispatcher to stop one process and start another running



Scheduling Criteria

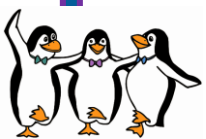
- Different CPU-scheduling algorithms have different properties,
  - The choice of an algorithm may favor one process over another.

- **Many criteria have been suggested for comparing CPU-scheduling algorithms:**
  - CPU utilization: keep the CPU as busy as possible
    - In real system from 40% to 90%
  - Throughput: number of processes that complete their execution per time unit
  - Turnaround time: amount of time taken to execute a particular process
    - It is the sum of the periods spent waiting to get into memory, waiting in the ready queue, executing on the CPU, and doing I/O.
  - 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 Criteria

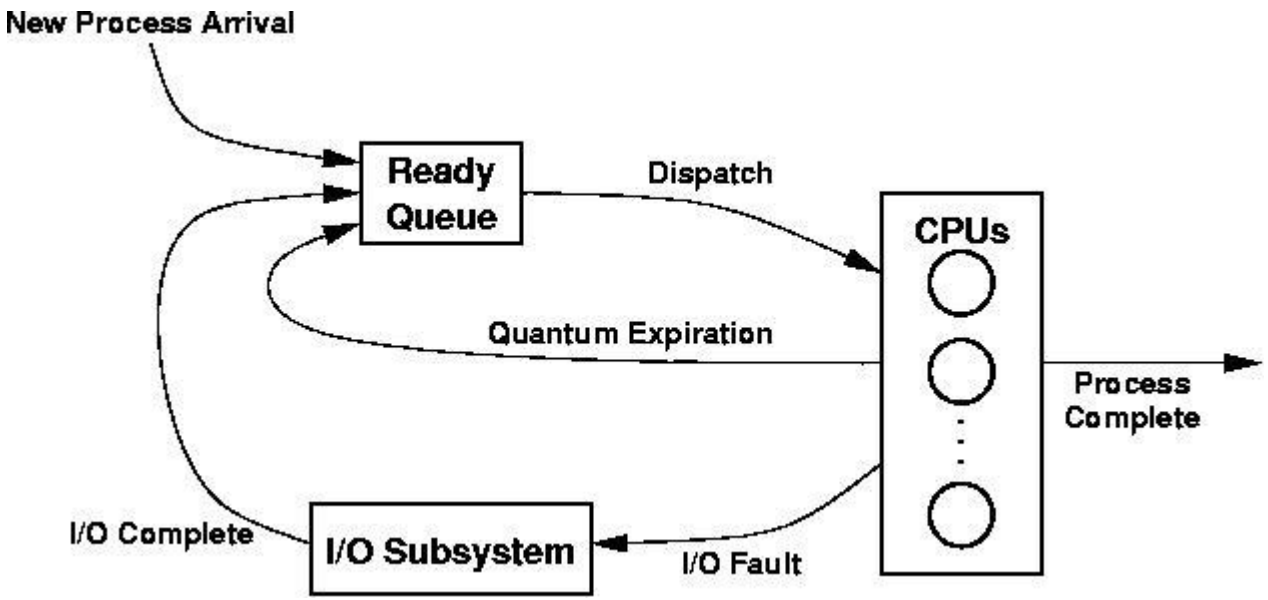
- Scheduling Algorithm Optimization Criteria
  - Max CPU utilization
  - Max throughput
  - Min turnaround time
  - Min waiting time
  - Min response time
- Usually, we optimize the average measure.
- Sometimes, we need to optimize the minimum or maximum values rather than the average
  - Example, to guarantee that all users get good service, we may want to minimize the maximum response time.



Scheduling Algorithms

Scheduling Algorithms

- Deciding on which of the processes in the ready queue is to be allocated the CPU.



- Gantt Chart
  - It is a bar chart that illustrates a particular schedule, including the start and finish times of each of the participating processes

Scheduling Algorithms

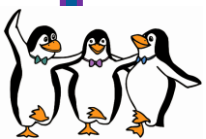
First- Come, First-Served (FCFS) Scheduling

- The process that requests the CPU first is allocated the CPU first.
- The implementation of the FCFS policy is easily managed with a FIFO queue.
- The average waiting time under the FCFS policy is often quite long
- Consider the following set of processes that arrive at time 0, with the length of the CPU burst given in milliseconds:

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



Scheduling Algorithms

- First- Come, First-Served (FCFS) Scheduling
  - Suppose that the processes arrive in the order:

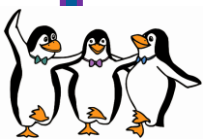
$$P_2 , P_3 , P_1$$

    - 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
    - **Convoy effect:** when short processes are waiting behind long process
      - Consider one CPU-bound and many I/O-bound processes
        - » All processes have to wait for one big process to get off the CPU.
        - » This results in lower CPU and device utilization

Scheduling Algorithms

- Shortest-Job-First (SJF) Scheduling
  - This algorithm associates with each process the length of its **next** CPU burst
    - Use these lengths to schedule the process with the shortest time
    - If two processes have the same length then use FCFS
  - SJF is optimal, it 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
    - SJF scheduling is used frequently in long-term scheduling.
  - The SJF algorithm is a special case of the general priority-scheduling algorithm



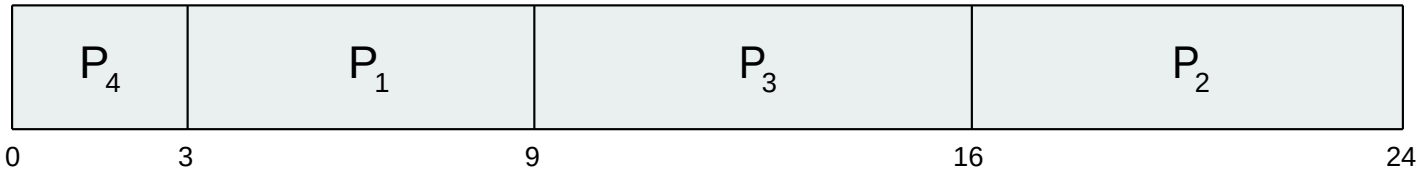
Scheduling Algorithms

Shortest-Job-First (SJF) Scheduling

- Example: consider the following set of processes, with the length of the CPU burst given in milliseconds:

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

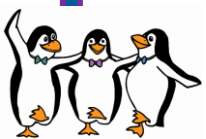
Scheduling Algorithms

- Shortest-Job-First (SJF) Scheduling
  - Determining Length of Next CPU Burst
    - We can only estimate the length: should be similar to the previous one
      - Then pick process with the shortest predicted next CPU burst
    - Can also be done by using the length of previous CPU bursts, using exponential averaging as:

      1. t_n = actual length of n^{th} CPU burst
      2. τ_{n+1} = predicted value for the next CPU burst
      3. $\alpha, 0 \le \alpha \le 1$
      4. Define:

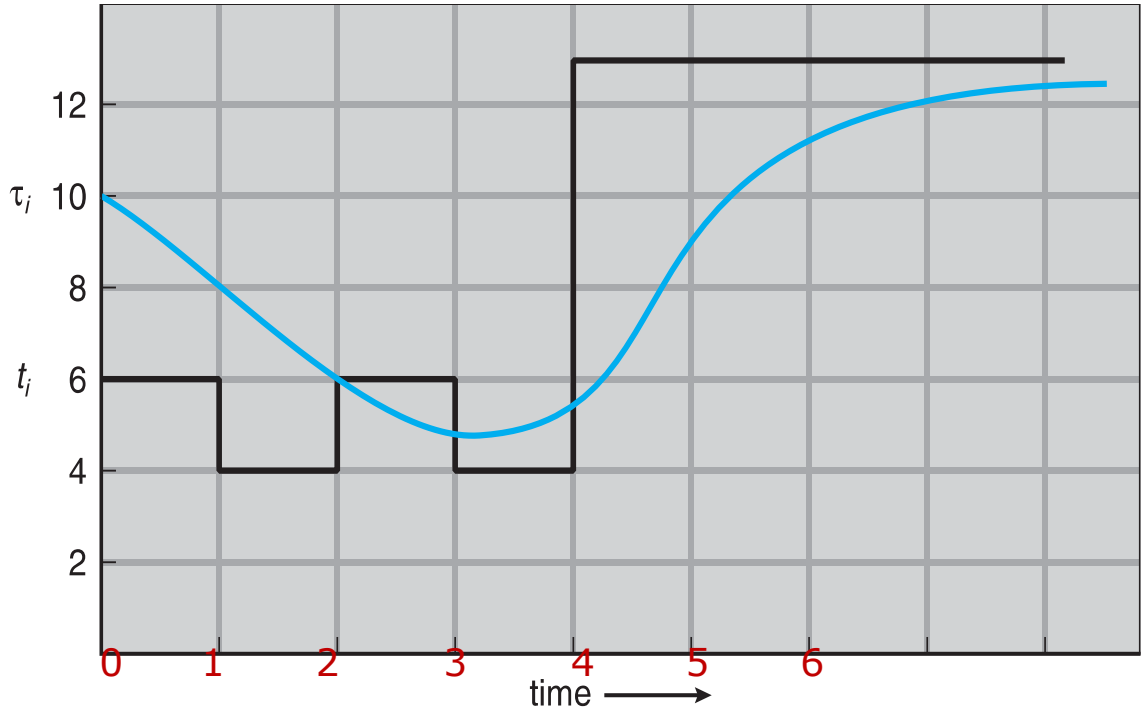
      $$\tau_{n+1} = \alpha\, t_n + (1-\alpha)\tau_n.$$

    - Commonly, α is set to ½ (relative weight of recent burst time and past history in the prediction formula)



Scheduling Algorithms

- Shortest-Job-First (SJF) Scheduling
  - Prediction of the Length of the Next CPU Burst



CPU burst (t_i)	6	4	6	4	13	13	13	...
"guess" (τ_i)	10	8	6	6	5	9	11	12	...

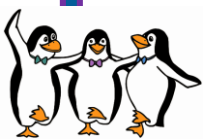
The prediction at t=1→10*0.5 + 6*0.5 =8 but the CPU burst is 4

Scheduling Algorithms

- Shortest-Job-First (SJF) Scheduling
  - 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:

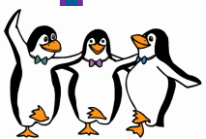
$$\tau_{n+1} = \alpha\, t_n + (1 - \alpha)\alpha\, t_{n-1} + \ldots$$
$$+ (1 - \alpha)^j \alpha\, t_{n-j} + \ldots$$
$$+ (1 - \alpha)^{n+1} \tau_0$$

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



Scheduling Algorithms

- Shortest-Job-First (SJF) Scheduling
  - **The SJF algorithm can be either preemptive or nonpreemptive:**
    - **Preemptive** version called **shortest-remaining-time-first**
      - If a new process arrives with a CPU burst length less than the remaining time of the current executing process, then pre-empt it.
    - **Non-preemptive**:
      - Once the CPU is given to a process it cannot be preempted until it completes its CPU burst

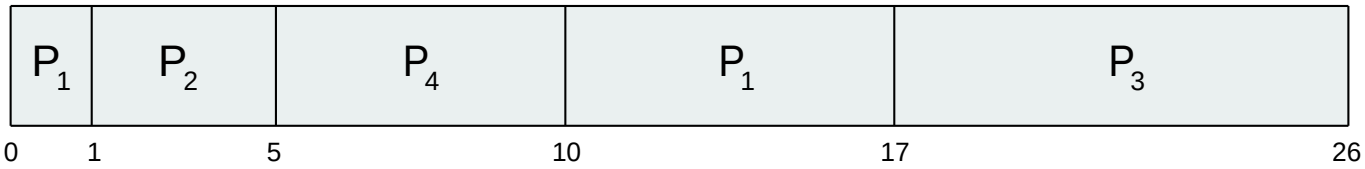


Scheduling Algorithms

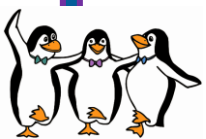
- Shortest-Job-First (SJF) Scheduling
  - Example of Shortest-remaining-time-first (preemptive)
    - Now we add the concepts of varying arrival times and preemption to the analysis

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

    - *Preemptive* SJF Gantt Chart



    - Average waiting time = [(10-1)+(1-1)+(17-2)+(5-3)]/4 = 26/4 = 6.5 msec

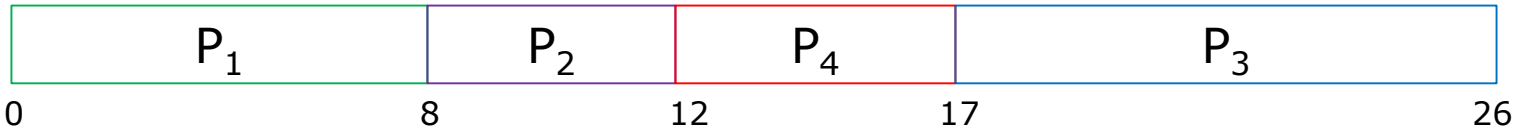


Scheduling Algorithms

- Shortest-Job-First (SJF) Scheduling
  - Example of Shortest-remaining-time-first (non-preemptive)
    - Now we add the concepts of varying arrival times and non-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

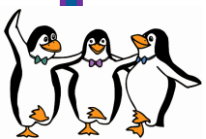
    - *Non-Preemptive* SJF Gantt Chart



    - Average waiting time = [(0-0)+(8-1)+(17-2)+(12-3)]/4 = 26/4 = 7.75 msec

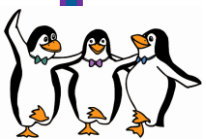
Scheduling Algorithms

- 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)
  - **Priorities can be defined either internally or externally.**
    - Internally → time limits, memory requirements, the number of open files, …
    - Externally → set by criteria outside the OS, such as the importance of the process, the department sponsoring the work, or political factors.
  - SJF is a priority scheduling where priority is the *inverse* of predicted next CPU burst time



Scheduling Algorithms

- Priority Scheduling
  - It can be either preemptive or nonpreemptive:
    - A preemptive priority scheduling algorithm will preempt the CPU if the priority of the newly arrived process is higher than the priority of the currently running process.
    - A nonpreemptive priority scheduling algorithm will simply put the new process at the head of the ready queue.
  - **Problem** ≡ **Starvation** → low priority processes may never execute
    - **Solution** ≡ **Aging** → gradually increase the priority of processes that wait in the system for a long time.
      - Example: we could increase the priority of a waiting process by 1 every 15 minutes.



Scheduling Algorithms

- Priority Scheduling
  - Example of Priority Scheduling
    - Consider the following set of processes, assumed to have arrived at time 0 in the order P1, P2, · · ·, P5

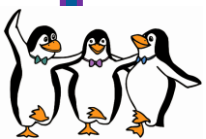
Process	Burst Time	Priority
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

    - Priority scheduling Gantt Chart

    - Average waiting time = ((0-0)+(1-0)+(6-0)+(16-0)+(18-0))/5=8.2 msec

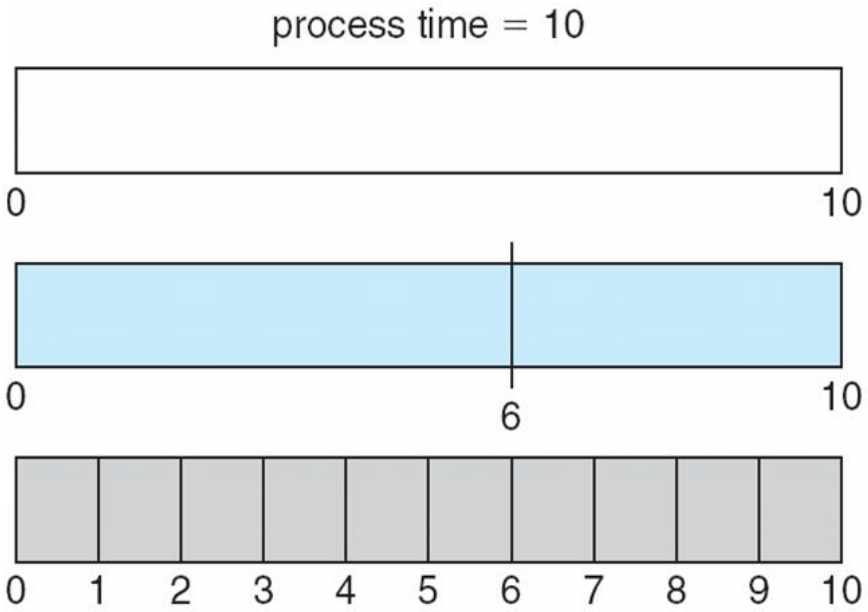
Scheduling Algorithms

- Round Robin (RR)
  - It is designed especially for timesharing Systems
  - 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 at every quantum to schedule next process
  - The RR performance depends on the time quantum q:
    - If *q* is large $\Rightarrow$ FCFS
    - If *q* *is* small $\Rightarrow$ *q* must be large with respect to context switch, otherwise the overhead is too high



Scheduling Algorithms

- Round Robin (RR)
  - Time Quantum and Context Switch Time



quantum	context switches
12	0
6	1
1	9

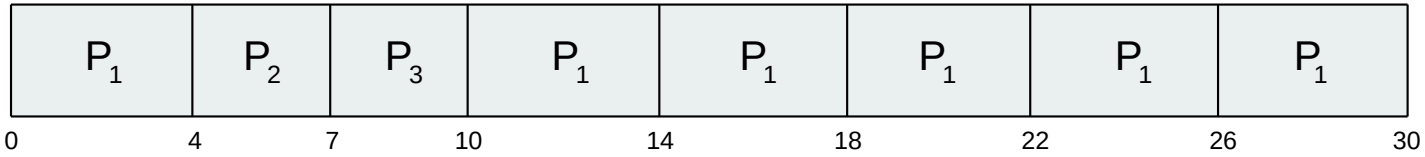
  - In the RR scheduling algorithm, no process is allocated the CPU for more than 1 time quantum in a row (unless it is the only runnable process).

Scheduling Algorithms

- Round Robin (RR)
  - Example of RR with Time Quantum = 4
    - Consider the following set of processes that arrive at time 0

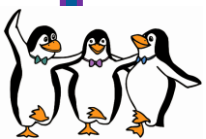
Process	Burst Time
P_1	24
P_2	3
P_3	3

    - The Gantt chart is:



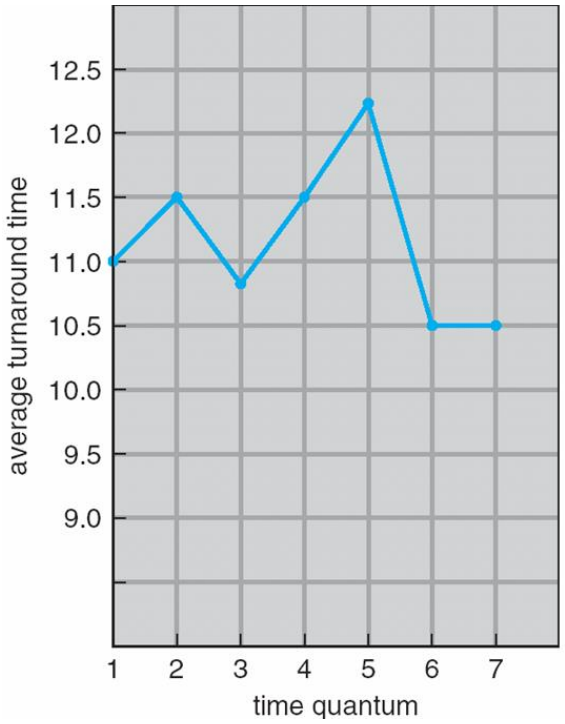
    - The average waiting time: P1 waits for 6 ms (10 - 4), P2 waits for 4 ms, and P3 waits for 7 ms. So, the average waiting time is 17/3 = 5.66 ms

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



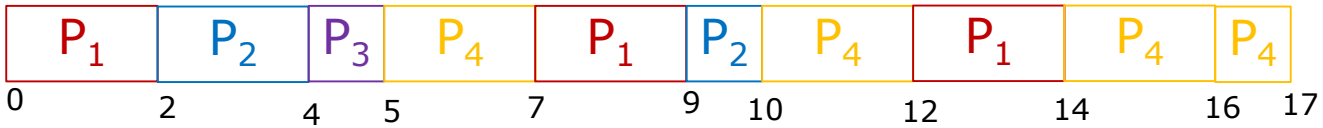
Scheduling Algorithms

- Round Robin (RR)
  - Turnaround time varies with the time quantum
  - A rule of thumb is that 80 percent of the CPU bursts should be shorter than the time quantum.



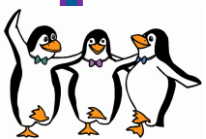
process	time
P_1	6
P_2	3
P_3	1
P_4	7

  - If q=2, turnaround time = (14 + 10 +5 +17)/4 =11.5



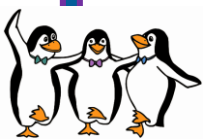
Scheduling Algorithms

- Multilevel Queue Scheduling
  - It is used in situations where processes are easily classified into different groups.
  - Ready queue is partitioned into separate queues,
    - Example :
      - **foreground** (interactive processes)
      - **background** (batch processes)
  - The processes are permanently assigned to one queue,
    - Based on some property of the process, such as memory size, process priority, or process type.
  - Each queue has its own scheduling algorithm, such as:
    - For foreground → RR
    - For background → FCFS



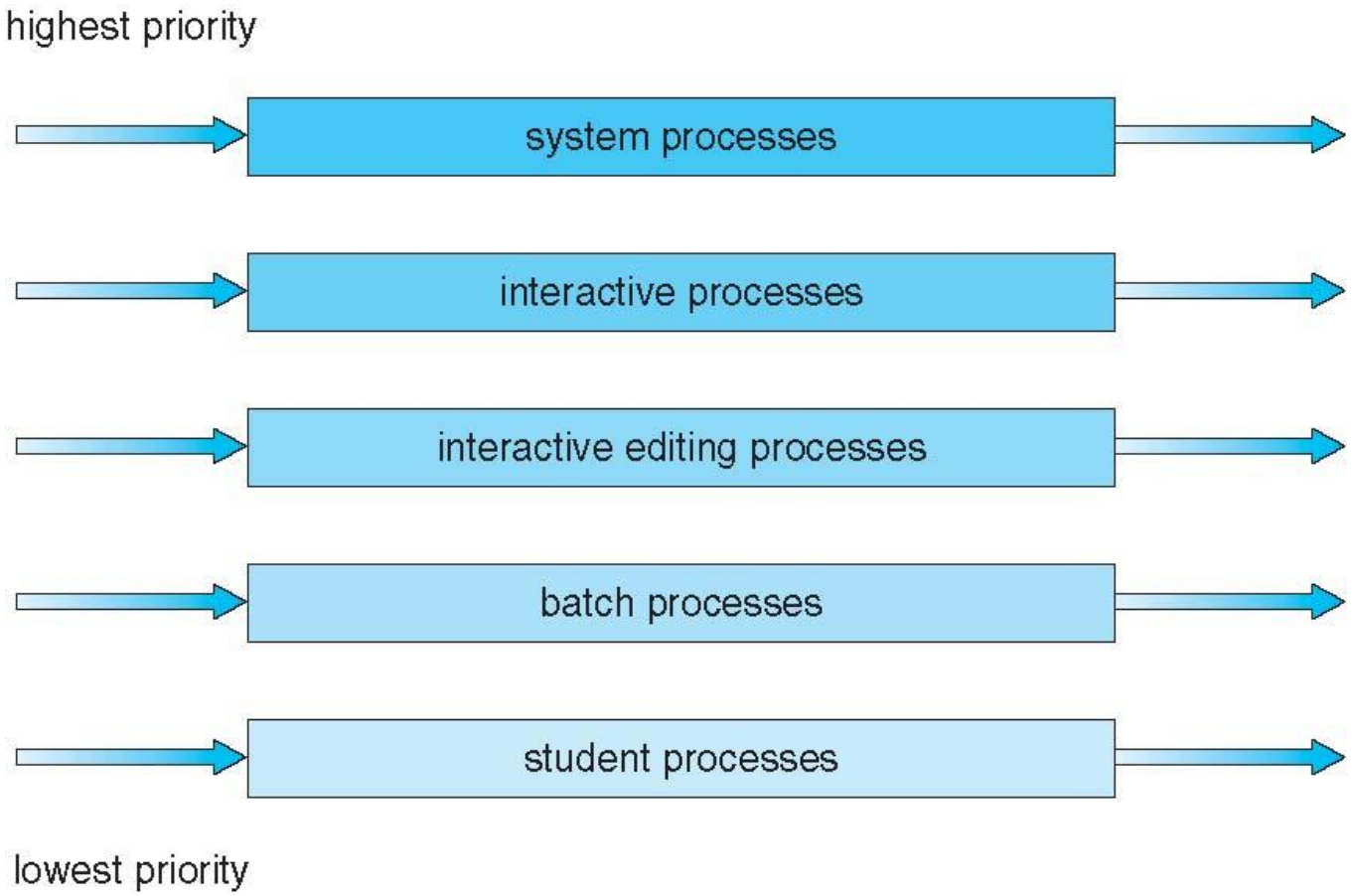
Scheduling Algorithms

- Multilevel Queue Scheduling
  - **Scheduling must be done between the queues, two options:**
    - Fixed priority scheduling → (i.e., serve all from foreground then from background). Possibility of starvation.
      - No low priority queue will be served unless the higher priority queues are empty
    - Time slice → each queue gets a certain amount of CPU time which it can schedule amongst its processes; i.e.,
      - 80% to foreground in RR
      - 20% to background in FCFS



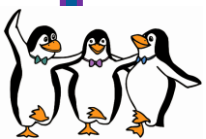
Scheduling Algorithms

- Multilevel Queue Scheduling
  - An example of a multilevel queue scheduling algorithm with five queues, listed below in order of priority:



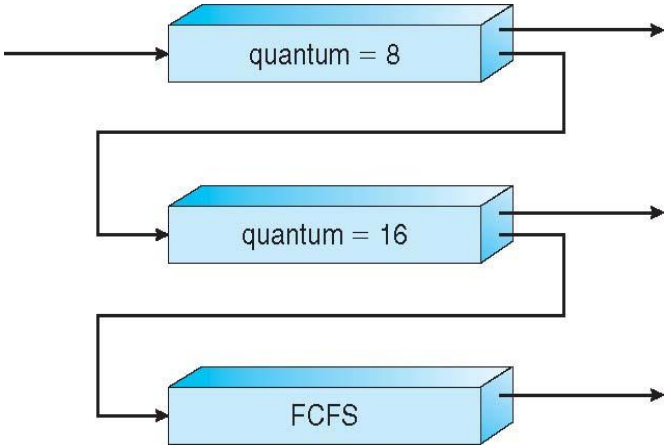
Scheduling Algorithms

- Multilevel Feedback Queue Scheduling
  - It allows a process to move between various queues; aging can be implemented this way
    - A process that waits too long in a lower-priority queue may be moved to a higher-priority queue.
  - Multilevel-feedback-queue scheduler defined by the following parameters:
    - The number of queues
    - The scheduling algorithm for each queue
    - The method used to determine when to upgrade a process to a higher queue
    - The method used to determine when to demote a process to a lower priority queue
    - The method used to determine which queue a process will enter when that process needs service



Scheduling Algorithms

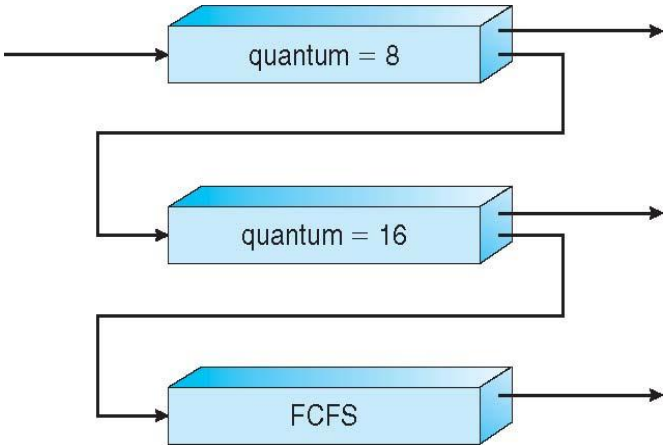
- Multilevel Feedback Queue Scheduling
  - Example of Multilevel Feedback Queue that has three queues:
    - Q_0 – RR with time quantum 8 milliseconds
    - Q_1 – RR time quantum 16 milliseconds
    - Q_2 – FCFS

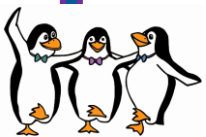


  - **Scheduling**
    - A new job first enters queue Q_0 which is served FCFS
      - When it gains CPU, job receives 8 milliseconds
      - If it does not finish in 8 milliseconds, it is moved to queue Q_1
    - At Q_1 (and Q_0 is empty) the job is again served FCFS and receives 16 additional milliseconds
      - If it still does not complete, it is preempted and moved to queue Q_2
    - Processes in Q_2 are run on an FCFS basis but are run only when Q_0 and Q_1 are empty

Scheduling Algorithms

- Multilevel Feedback Queue Scheduling
  - This scheme separates processes according to their CPU bursts:
    - If a process uses too much CPU time, it will be moved to a lower-priority queue.
    - This scheme leaves I/O-bound and interactive processes in the higher-priority queues.

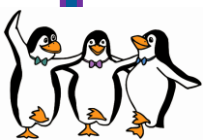




Thread Scheduling

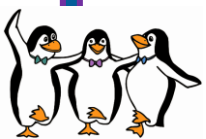
- Threads can be either user-level or kernel-level threads
- On operating systems that support threads, it is the kernel-level threads (not processes) that are being scheduled by the operating system.
  - The thread library maps user-level threads to kernel-level threads

- **Contention Scope**
  - In many-to-one and many-to-many models, thread library schedules user-level threads to run on an available LWP (lightweight process)
    - This scheme is known as **process-contention scope (PCS)** since competition for the CPU takes place among threads belonging to the same process
    - Typically done via priority set by programmer
  - To decide which kernel-level thread to schedule onto a CPU, the kernel uses **system-contention scope (SCS).**
    - The competition in SCS is among all threads in system
    - Systems using the one-to-one model, such as Windows, Linux, and Solaris, schedule threads using only SCS



Thread Scheduling

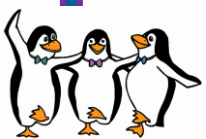
- Pthread Scheduling
  - API allows specifying either PCS or SCS during thread creation
    - PTHREAD_SCOPE_PROCESS schedules threads using PCS scheduling
    - PTHREAD_SCOPE_SYSTEM schedules threads using SCS scheduling
  - Can be limited by OS – Linux and Mac OS X only allow PTHREAD_SCOPE_SYSTEM
  - Example:
    - Pthread scheduling API.
      - The program first determines the existing contention scope and sets it to PTHREAD_SCOPE_SYSTEM.
      - It then creates five separate threads that will run using the SCS scheduling policy.



Thread Scheduling

- Pthread scheduling API.

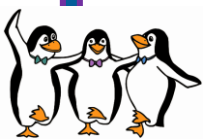
```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5
int main(int argc, char *argv[]) {
   int i, scope;
   pthread_t tid[NUM_THREADS];
   pthread_attr_t attr;
   /* get the default attributes */
   pthread_attr_init(&attr);
   /* first inquire on the current scope */
   if (pthread_attr_getscope(&attr, &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");
   }
```



Thread Scheduling

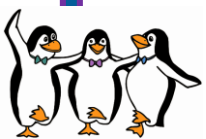
- Pthread scheduling API.

```
/* 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);
}
```



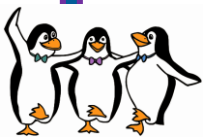
Multiple-Processor Scheduling

- CPU scheduling more complex when multiple CPUs are available
- Assume that we have **homogeneous processors:** the processors are identical in terms of their functionality, still we have problems:
  - Consider a system with an I/O device attached to a private bus of one processor.
    - Processes that wish to use that device must be scheduled to run on that processor.



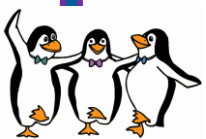
Multiple-Processor Scheduling

- **Approaches to Multiple-Processor Scheduling:**
  - **Asymmetric multiprocessing**
    - All scheduling decisions, I/O processing, and other system activities handled by a single processor→ the master server.
  - **Symmetric multiprocessing (SMP)**
    - Each processor is self-scheduling, all processes are in common ready queue, or each has its own private queue of ready processes
    - It is the most common, used in Windows, Linux, and Mac OS X.



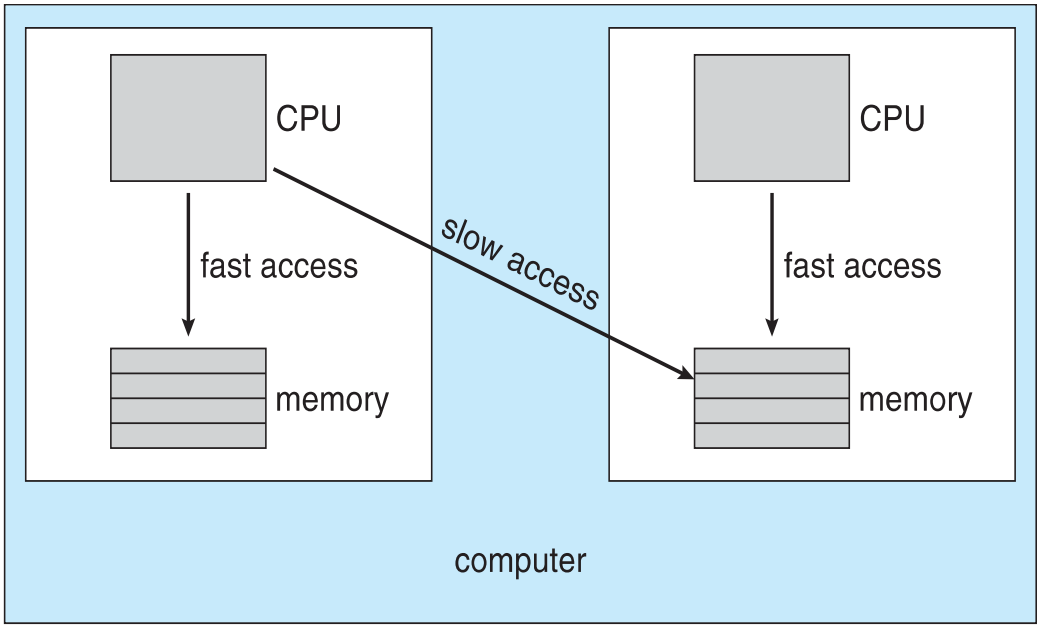
Multiple-Processor Scheduling

- **Approaches to Multiple-Processor Scheduling:**
  - **Processor affinity** – keep a process running on the same processor that it was before,
    - A process has affinity for a processor on which it is currently running
    - If a process is allowed to be migrated to another processor then the old cache has to be invalidated and a new cache has to be repopulated
    - **Affinity takes two forms:**
      - **Soft affinity:** OS tries to keep a process on the same processor but has no guarantee to keep it this way
      - **Hard affinity:** OS allows a process to specify a subset of processors on which it may run by using a system call (e.g. sched_setaffinity())
      - Many systems provide both soft and hard affinity



Multiple-Processor Scheduling

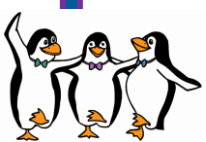
- **NUMA and CPU Scheduling:**
  - The main-memory architecture of a system can affect processor affinity issues.
  - The Figure illustrates an architecture featuring non-uniform memory access (NUMA), in which a CPU has faster access to some parts of the main memory than to other parts.
    - This happens when the system has multiple CPUs and memory boards.
    - The CPUs on a board can access the memory on that board faster than they can access memory on other boards in the system



Note that memory-placement algorithms can also consider affinity

Multiple-Processor Scheduling

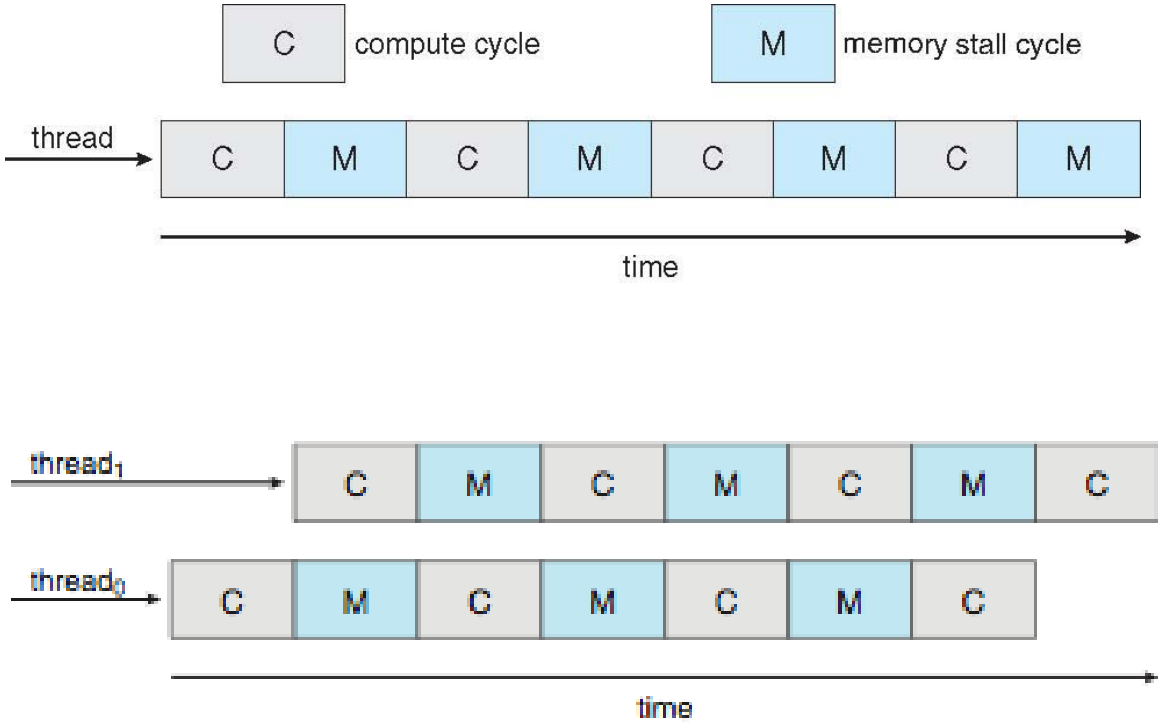
- Load Balancing
  - If SMP is used, we need to keep all CPUs loaded for efficiency
  - **Load balancing** attempts to keep workload evenly distributed across all processors
    - It is necessary in systems where each processor has its own private ready queue
    - It is not needed when the system has one common ready queue,
  - **Two general approaches to load balancing**
    - **Push migration** → a specific task periodically checks the load on each processor, and if it finds an imbalance, evenly distributes the load by moving (or pushing) processes from overloaded to idle or less-busy processors.
    - **Pull migration** → occurs when an idle processors pulls a waiting task from a busy processor
  - Push and pull migration are often implemented in parallel



Multiple-Processor Scheduling

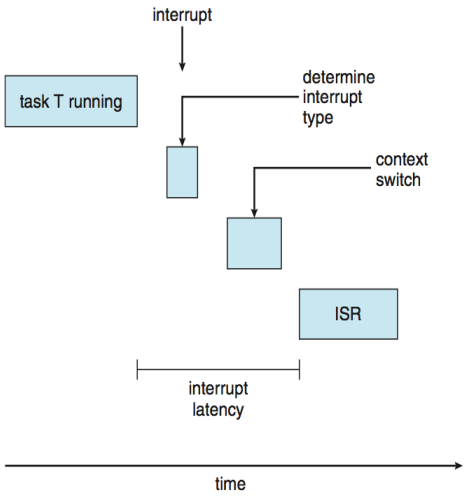
Multicore Processors

- Recent trend to place multiple processor cores on the same physical chip
  - It is faster and consumes less power than multiple processors
- Multiple threads per core also growing
  - It takes advantage of memory stall to make progress on another thread while memory retrieve happens



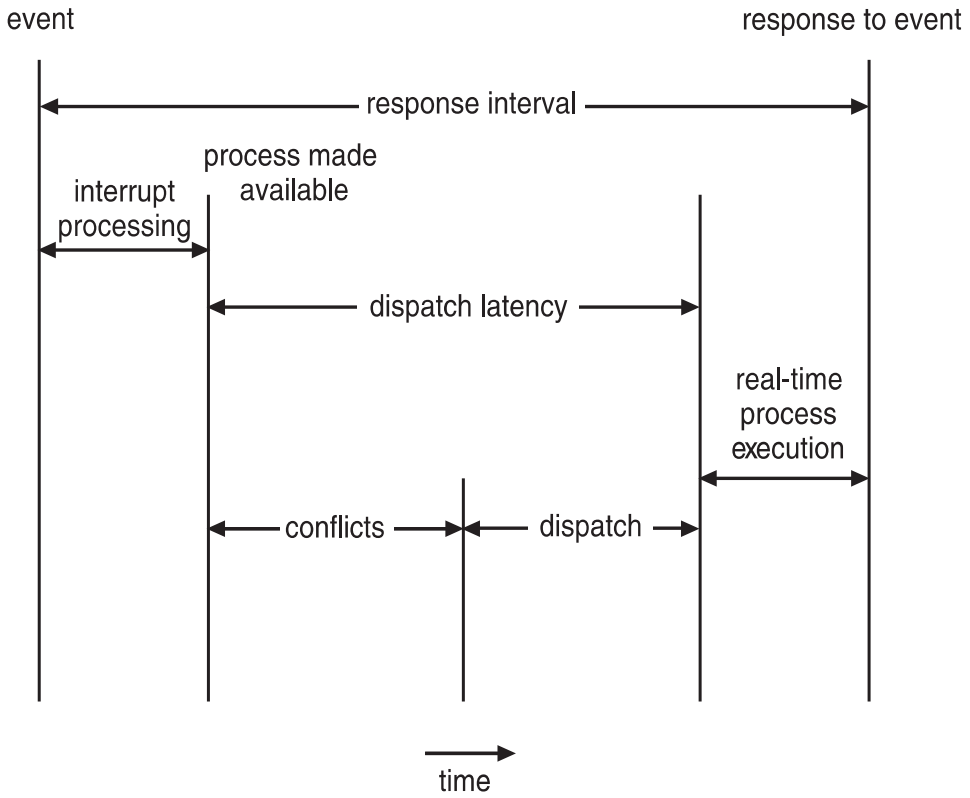
Real-Time CPU Scheduling

- Can present obvious challenges
- **Soft real-time systems** – no guarantee as to when critical real-time process will be scheduled
- **Hard real-time systems** – task must be serviced by its deadline
- When an event occurs, the system must respond to and service it as quickly as possible.
- Event latency is 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



Real-Time CPU Scheduling

- Conflict phase of dispatch latency has two components:
  1. Preemption of any process running in kernel mode
  2. Release by low-priority process of resources needed by high-priority processes
- As an example, in Solaris, the dispatch latency with preemption disabled is over a hundred milliseconds. With preemption enabled, it is reduced to less than a millisecond.



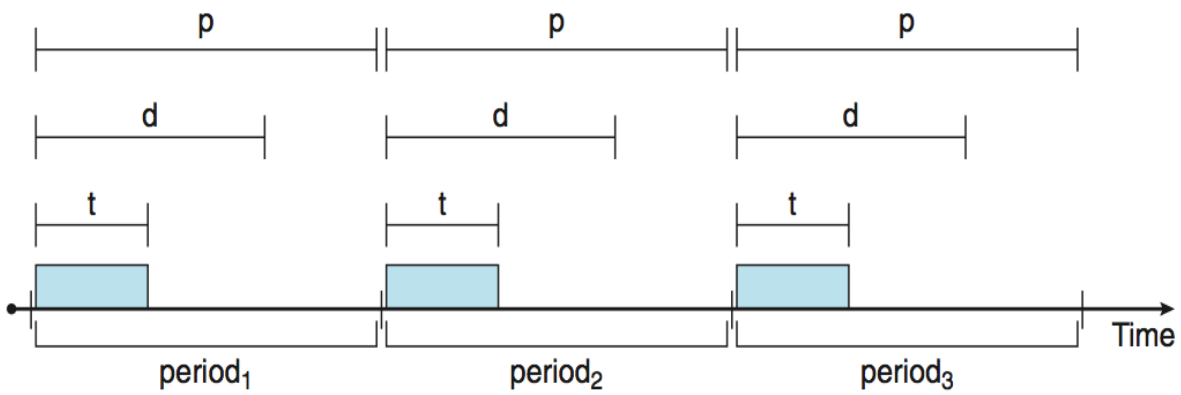
Real-Time CPU Scheduling

Priority-based Scheduling

- For real-time scheduling, scheduler must support preemptive, priority-based scheduling
  - But only guarantees soft real-time
- For hard real-time 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*, such that:

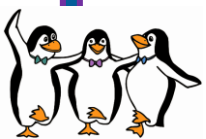
    $$0 \leq t \leq d \leq p$$

  - **Rate** of periodic task is $1/p$



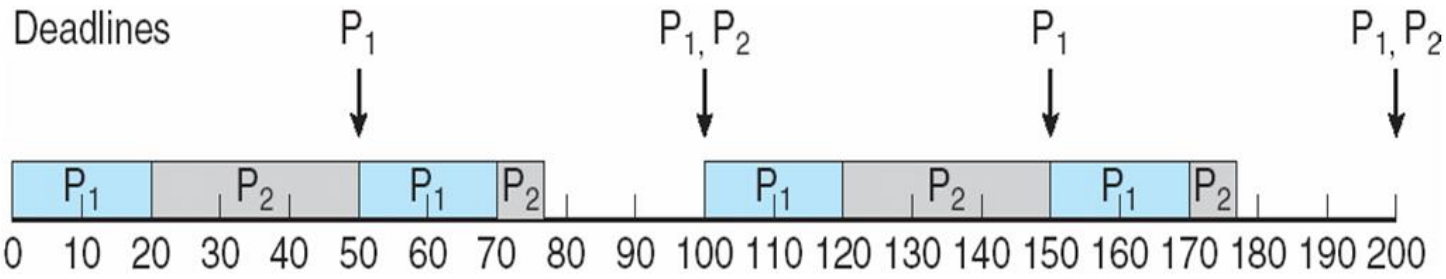
Real-Time CPU Scheduling

- Virtualization and Scheduling
  - Virtualization software schedules multiple guests onto CPU(s)
  - Each guest doing its own scheduling
    - Not knowing it doesn't own the CPUs
    - Can result in poor response time
    - Can effect time-of-day clocks in guests
  - Can undo good scheduling algorithm efforts of guests



Real-Time CPU Scheduling

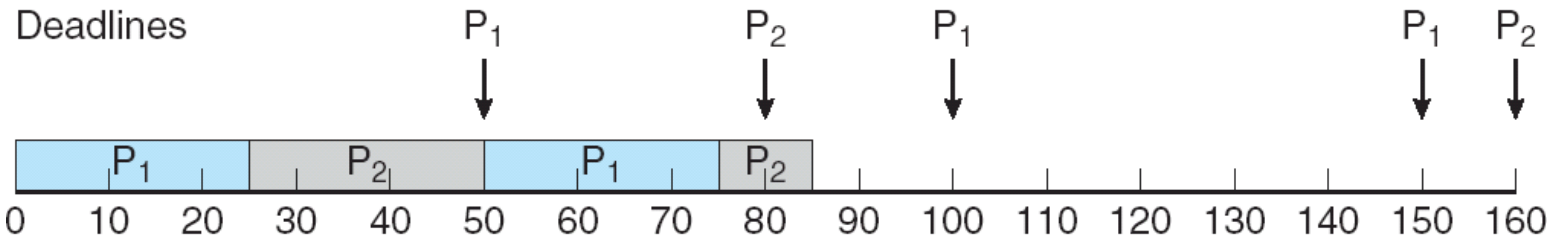
- Rate Montonic Scheduling
  - A priority is assigned based on the inverse of its period
  - Shorter periods = higher priority;
  - Longer periods = lower priority
  - P_1 is assigned a higher priority than P_2.
  - Example:
    - We have two processes, P1 and P2 with periods p1 = 50 and p2 = 100. The processing times are t1 = 20 for P1 and t2 = 35 for P2. The deadline for each process requires that it complete its CPU burst by the start of its next period.



Real-Time CPU Scheduling

Rate Montonic Scheduling

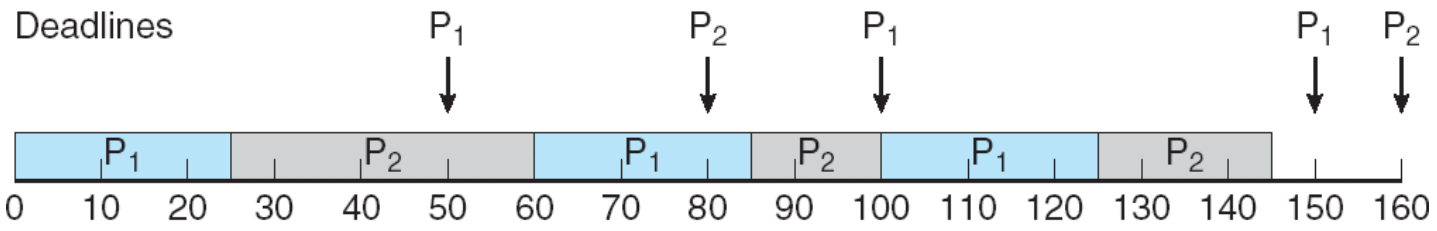
- Example: Missed Deadlines with Rate Monotonic Scheduling
  - Assume that process P1 has a period of p1 = 50 and a CPU burst of t1 = 25. For P2, the corresponding values are p2 = 80 and t2 = 35.
  - Rate-monotonic scheduling would assign process P1 a higher priority, as it has the shorter period.
  - The total CPU utilization of the two processes is (25/50)+(35/80) = 0.94,
- Despite being optimal, then, rate-monotonic scheduling has a limitation:
  - CPU utilization is bounded, and it is not always possible fully to maximize CPU resources.
  - The worst-case CPU utilization for scheduling N processes is $N(2^{1/N} - 1)$.



Real-Time CPU Scheduling

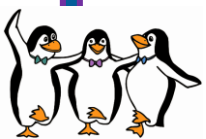
Earliest Deadline First Scheduling (EDF)

- Priorities are assigned according to deadlines:
  - the earlier the deadline, the higher the priority;
  - the later the deadline, the lower the priority
- Example:
  - Recall that P1 has values of p1 = 50 and t1 = 25 and that P2 has values of p2 = 80 and t2 = 35.
  - Process P1 has the earliest deadline, so its initial priority is higher than that of process P2. Process P2 begins running at the end of the CPU burst for P1.
  - However, whereas rate-monotonic scheduling allows P1 to preempt P2 at the beginning of its next period at time 50, EDF scheduling allows process P2 to continue running.
  - P2 now has a higher priority than P1 because its next deadline (at time 80) is earlier than that of P1 (at time 100).



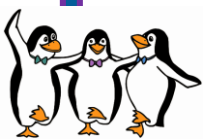
Real-Time CPU Scheduling

- 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



Real-Time CPU Scheduling

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

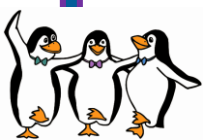


Real-Time CPU Scheduling

POSIX Real-Time Scheduling API

- This program first determines the current scheduling policy and then sets the scheduling algorithm to SCHED FIFO.

```
#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");
   }
```

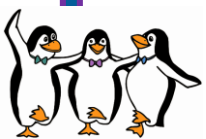


Real-Time CPU Scheduling

- POSIX Real-Time Scheduling API

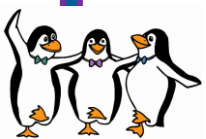
```
   /* set the scheduling policy - FIFO, RR, or OTHER */
   if (pthread_attr_setschedpolicy(&attr, SCHED_FIFO) != 0)
      fprintf(stderr, "Unable to set policy.\n");
   /* create the threads */
   for (i = 0; i < NUM_THREADS; i++)
      pthread_create(&tid[i],&attr,runner,NULL);
   /* now join on 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);
}
```



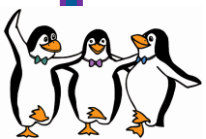
Operating System Examples

- Linux scheduling
- Windows scheduling
- Solaris scheduling



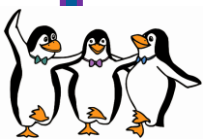
Operating System Examples

- Linux Scheduling Through Version 2.5
  - Prior to kernel version 2.5, it uses a 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



Operating System Examples

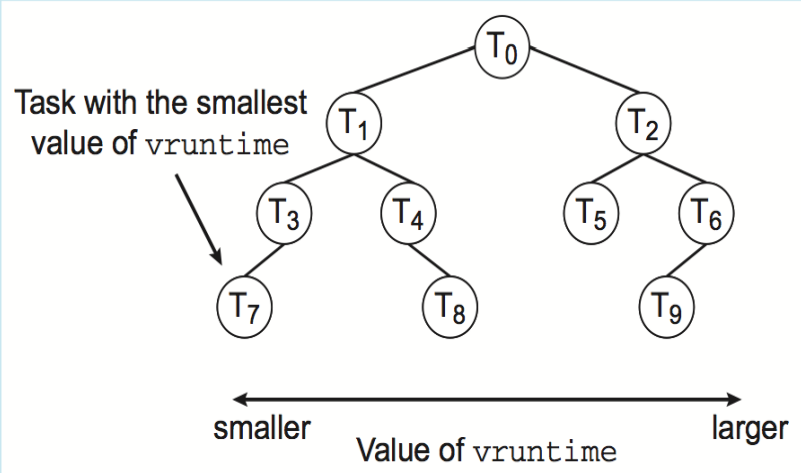
- Linux Scheduling in Version 2.6.23 +
  - It uses the ***Completely Fair Scheduler*** (**CFS**)
  - **Scheduling classes**
    - Each has specific priority
    - Scheduler picks highest priority task in highest scheduling class
    - Rather than quantum based on fixed time allotments, based on proportion 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



Operating System Examples

- **CFS Performance**

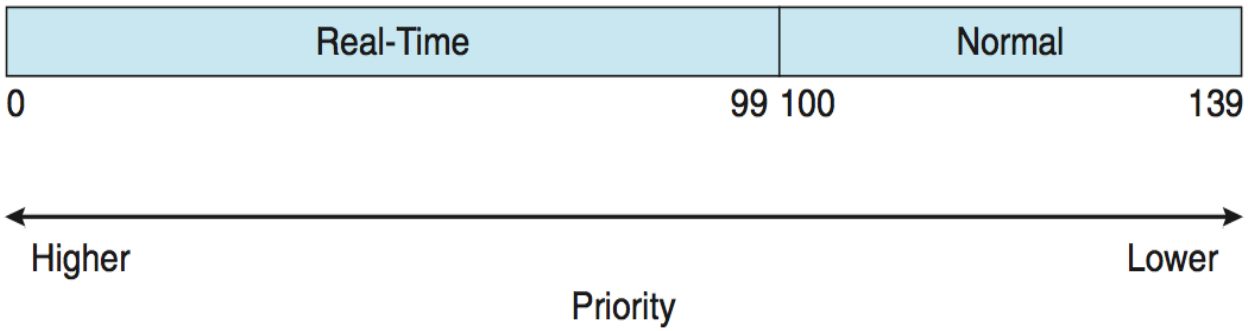
The Linux CFS scheduler provides an efficient algorithm for selecting which task to run next. Each runnable task is placed in a red-black tree—a balanced binary search tree whose key is based on the value of `vruntime`. This tree is shown below:



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.

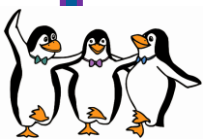
Operating System Examples

- Linux 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



Operating System Examples

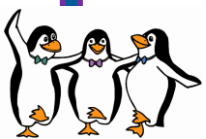
- Windows Scheduling
  - Windows uses priority-based preemptive scheduling
  - Highest-priority thread runs next
  - **Dispatcher** is scheduler
  - Thread runs until (1) blocks, (2) uses time slice, (3) preempted by higher-priority thread
  - Real-time threads can preempt non-real-time
  - 32-level 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**



Operating System Examples

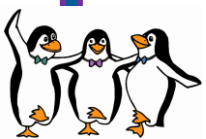
Windows Priority Classes

- 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



Operating System Examples

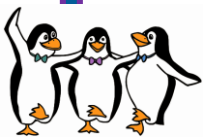
- Windows Priority Classes
  - If wait occurs, priority boosted depending on what was waited for
  - Foreground window given 3x priority boost
  - Windows 7 added **user-mode scheduling** (**UMS**)
    - Applications create and manage threads independent of kernel
    - For large number of threads, much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



Operating System Examples

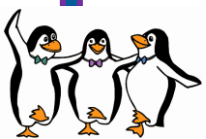
- Windows Priories

	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



Operating System Examples

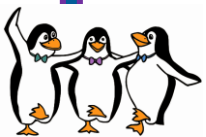
- Solaris
  - Priority-based scheduling
  - Six classes available
    - Time sharing (default) (TS)
    - Interactive (IA)
    - Real time (RT)
    - System (SYS)
    - Fair Share (FSS)
    - Fixed priority (FP)
  - Given thread can be in one class at a time
  - Each class has its own scheduling algorithm
  - Time sharing is multi-level feedback queue
    - Loadable table configurable by sysadmin



Operating System Examples

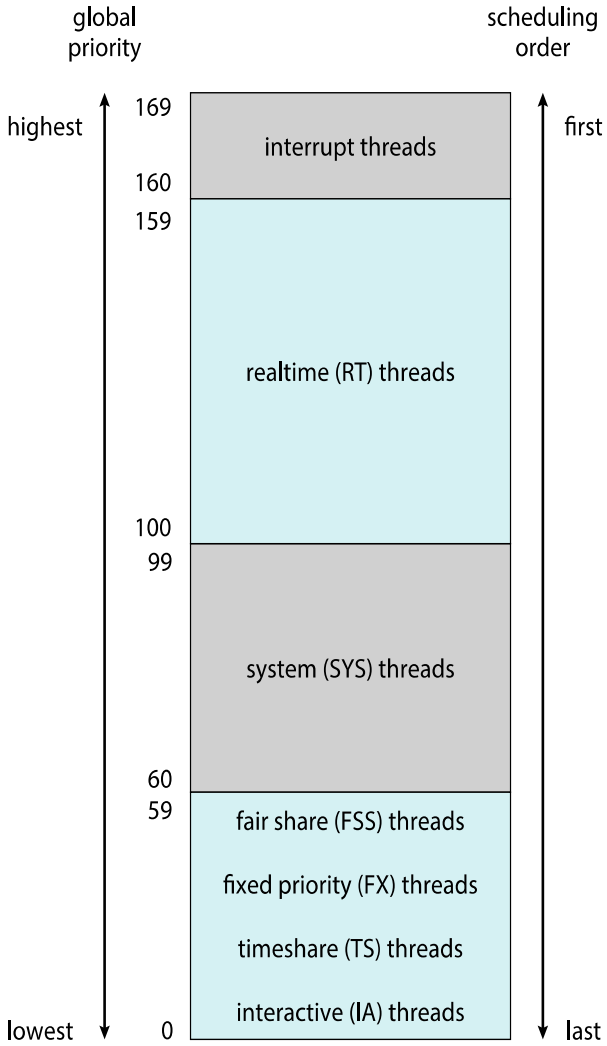
Solaris Dispatch Table

priority	time quantum	time quantum expired	return from sleep
0	200	0	50
5	200	0	50
10	160	0	51
15	160	5	51
20	120	10	52
25	120	15	52
30	80	20	53
35	80	25	54
40	40	30	55
45	40	35	56
50	40	40	58
55	40	45	58
59	20	49	59



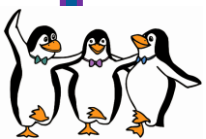
Operating System Examples

Solaris Scheduling



Operating System Examples

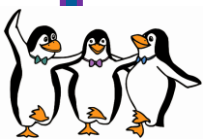
- Solaris Scheduling
  - Scheduler converts class-specific priorities into a per-thread global priority
    - Thread with highest priority runs next
    - Runs until (1) blocks, (2) uses time slice, (3) preempted by higher-priority thread
    - Multiple threads at same priority selected via RR



Algorithm Evaluation

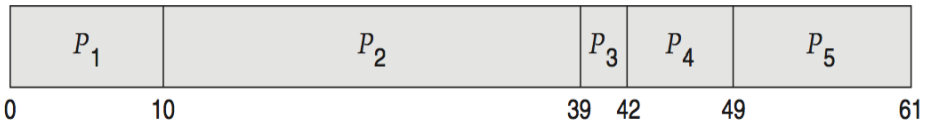
- How to select CPU-scheduling algorithm for an OS?
- Determine criteria, then evaluate algorithms
- **Deterministic modeling**
  - Type of **analytic evaluation**
  - Takes a particular predetermined workload and defines the performance of each algorithm for that workload
- Consider 5 processes arriving at time 0:

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

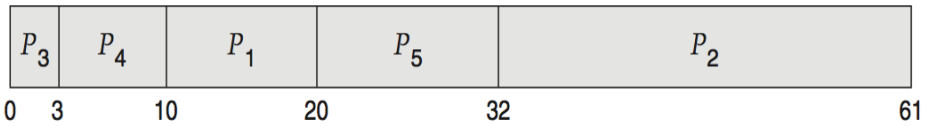


Algorithm Evaluation

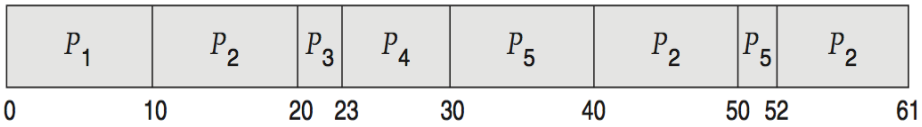
- Deterministic Evaluation
  - For each algorithm, calculate minimum average waiting time
  - Simple and fast, but requires exact numbers for input, applies only to those inputs
    - FCS is 28ms:



    - Non-preemptive SFJ is 13ms:

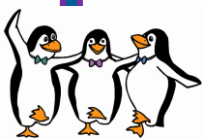


    - RR is 23ms:



Algorithm Evaluation

- 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

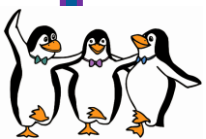


Algorithm Evaluation

- 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:

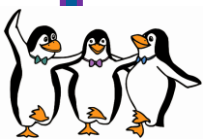
    $n = \lambda \times W$

    - 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



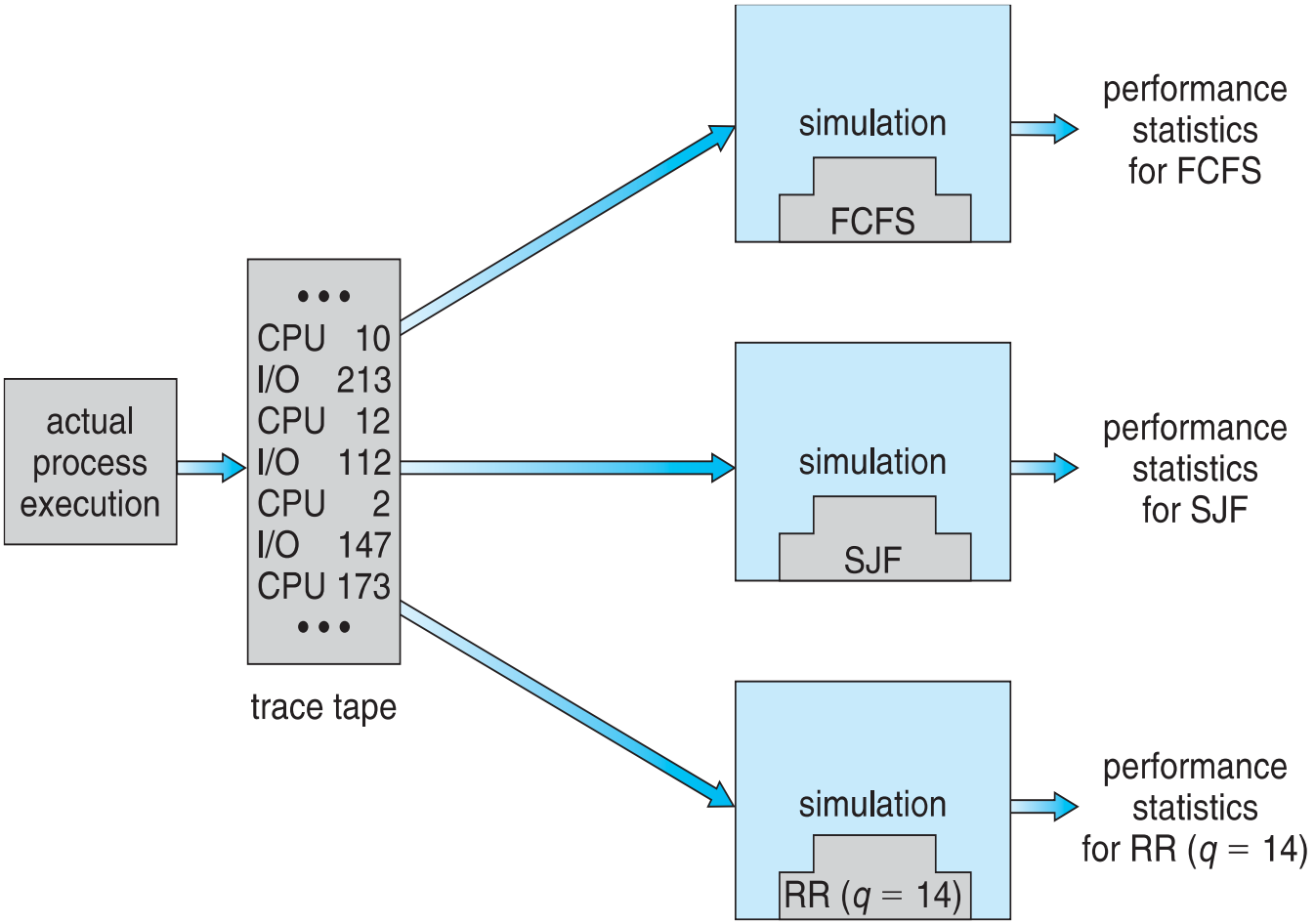
Algorithm Evaluation

- 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



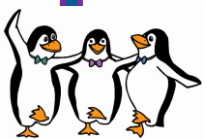
Algorithm Evaluation

- Evaluation of CPU Schedulers by Simulation



Algorithm Evaluation

- 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



Examples

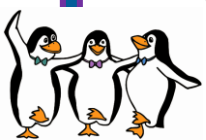
- Q1: Suppose that the following processes arrive for execution at the times indicated. Each process will run for the amount of time listed. In answering the questions, use nonpreemptive scheduling, and base all decisions on the information you have at the time the decision must be made.

Process	Arrival Time	Burst Time
*P*1	0.0	8
*P*2	0.4	4
*P*3	1.0	1

a) What is the average turnaround time for these processes with the FCFS scheduling algorithm?

b) What is the average turnaround time for these processes with the SJF scheduling algorithm?

c) The SJF algorithm is supposed to improve performance, but notice that we chose to run process *P*1 at time 0 because we did not know that two shorter processes would arrive soon. Compute what the average turnaround time will be if the CPU is left idle for the first 1 unit and then SJF scheduling is used. Remember that processes P1 and P2 are waiting during this idle time, so their waiting time may increase. This algorithm could be known as future-knowledge scheduling.



Examples

- Q1: Answer:

a) 10.53

b) 9.53

c) 6.86