# CENG 2034 - Operating Systems

## Week 7 - Week 8: CPU Scheduling

Burak Ekici

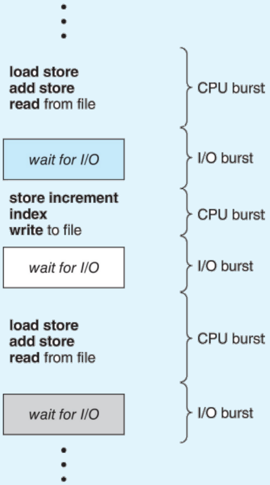
May 5 and May 12, 2023

# Outline

1. Basics
2. Scheduling Criteria & Algorithms
3. Thread Scheduling
4. Multiprocessor Scheduling
5. Real-Time CPU Scheduling
6. OS Examples
7. Algorithm Evaluation

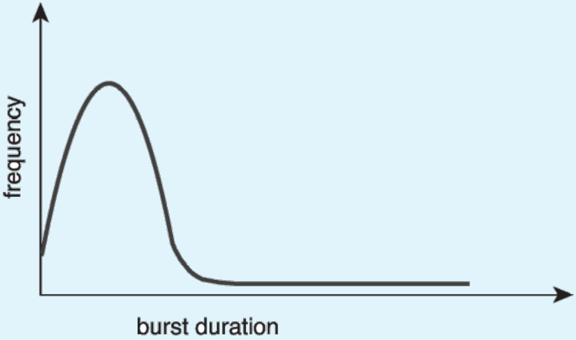
## Basic Concepts

- Maximum CPU utilization obtained with multiprogramming
- CPU–I/O Burst Cycle – Process execution consists of a cycle of CPU execution and I/O wait
- CPU burst followed by I/O burst
- CPU burst distribution is of main concern



## Histogram of CPU-burst Times

- Large number of short bursts
- Small number of longer bursts



## CPU Scheduler

- The CPU scheduler selects among the processes in ready queue, and allocates a CPU core to one of them

## CPU Scheduler

- The CPU scheduler selects among the processes in ready queue, and allocates a CPU core to one of them
    - Queue may be ordered in various ways

## CPU Scheduler

- The CPU scheduler selects among the processes in ready queue, and allocates a CPU core to one of them
  - Queue may be ordered in various ways
- CPU scheduling decisions may take place when a process:

## CPU Scheduler

- The CPU scheduler selects among the processes in ready queue, and allocates a CPU core to one of them
  - Queue may be ordered in various ways
- CPU scheduling decisions may take place when a process:
  1. Switches from running to waiting state

## CPU Scheduler

- The CPU scheduler selects among the processes in ready queue, and allocates a CPU core to one of them
  - Queue may be ordered in various ways
- CPU scheduling decisions may take place when a process:
  1. Switches from running to waiting state
  2. Switches from running to ready state

## CPU Scheduler

- The CPU scheduler selects among the processes in ready queue, and allocates a CPU core to one of them
  - Queue may be ordered in various ways
- CPU scheduling decisions may take place when a process:
  1. Switches from running to waiting state
  2. Switches from running to ready state
  3. Switches from waiting to ready

## CPU Scheduler

- The CPU scheduler selects among the processes in ready queue, and allocates a CPU core to one of them
  - Queue may be ordered in various ways
- CPU scheduling decisions may take place when a process:
  1. Switches from running to waiting state
  2. Switches from running to ready state
  3. Switches from waiting to ready
  4. Terminates

## CPU Scheduler

- The CPU scheduler selects among the processes in ready queue, and allocates a CPU core to one of them
  - Queue may be ordered in various ways
- CPU scheduling decisions may take place when a process:
  1. Switches from running to waiting state
  2. Switches from running to ready state
  3. Switches from waiting to ready
  4. Terminates
- For 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 for execution.

## CPU Scheduler

- The CPU scheduler selects among the processes in ready queue, and allocates a CPU core to one of them
  - Queue may be ordered in various ways
- CPU scheduling decisions may take place when a process:
  1. Switches from running to waiting state
  2. Switches from running to ready state
  3. Switches from waiting to ready
  4. Terminates
- For 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 for execution.
- For situations 2 and 3, however, there is a choice

## Preemptive and Nonpreemptive Scheduling

- When scheduling takes place only under circumstances 1 and 4, the scheduling scheme is nonpreemptive

## Preemptive and Nonpreemptive Scheduling

- When scheduling takes place only under circumstances 1 and 4, the scheduling scheme is nonpreemptive
- Otherwise, it is preemptive

## Preemptive and Nonpreemptive Scheduling

- When scheduling takes place only under circumstances 1 and 4, the scheduling scheme is nonpreemptive
- Otherwise, it is preemptive
- Under Nonpreemptive scheduling, 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

## Preemptive and Nonpreemptive Scheduling

- When scheduling takes place only under circumstances 1 and 4, the scheduling scheme is nonpreemptive
- Otherwise, it is preemptive
- Under Nonpreemptive scheduling, 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
- 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

## Preemptive Scheduling and Race Conditions

- Preemptive scheduling can result in race conditions when data are shared among several processes
- Consider the case of two processes that 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

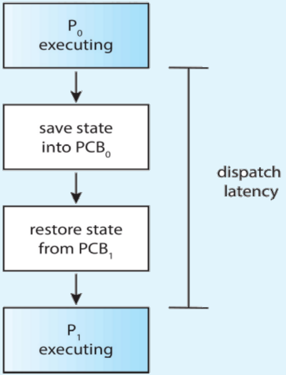
## Preemptive Scheduling and Race Conditions

- Preemptive scheduling can result in race conditions when data are shared among several processes
- Consider the case of two processes that 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

## Dispatcher

- Dispatcher module gives control of the CPU to the process selected by the CPU scheduler; this involves:
  - Switching context
  - Switching to user mode
  - Jumping to the proper location in the user program to restart that program

Dispatch latency – time it takes for the dispatcher to stop one process and start another running



# Outline

## Scheduling Criteria

- CPU utilization – keep the CPU as busy as possible

## Scheduling Criteria

- CPU utilization – keep the CPU as busy as possible
- Throughput – # of processes that complete their execution per time unit

## Scheduling Criteria

- CPU utilization – keep the CPU as busy as possible
- Throughput – # of processes that complete their execution per time unit
- Turnaround time – amount of time to execute a particular process

## Scheduling Criteria

- CPU utilization – keep the CPU as busy as possible
- Throughput – # of processes that complete their execution per time unit
- Turnaround time – amount of time to execute a particular process
- Waiting time – amount of time a process has been waiting in the ready queue

## Scheduling Criteria

- CPU utilization – keep the CPU as busy as possible
- Throughput – # of processes that complete their execution per time unit
- Turnaround time – amount of time to execute a particular process
- 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.

## Scheduling Algorithm Optimization Criteria

- Max CPU utilization

## Scheduling Algorithm Optimization Criteria

- Max CPU utilization
- Max throughput

## Scheduling Algorithm Optimization Criteria

- Max CPU utilization
- Max throughput
- Min turnaround time

## Scheduling Algorithm Optimization Criteria

- Max CPU utilization
- Max throughput
- Min turnaround time
- Min waiting time

## Scheduling Algorithm Optimization Criteria

- Max CPU utilization
- Max throughput
- Min turnaround time
- Min waiting time
- Min response time

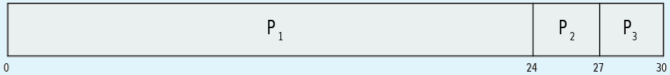
## First-Come, First-Served (FCFS) Scheduling

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

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

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

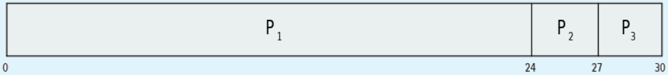
- Suppose that the processes arrive in the order: $P_1$, $P_2$ , $P_3$ The Gantt Chart for the schedule is:



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

| 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:

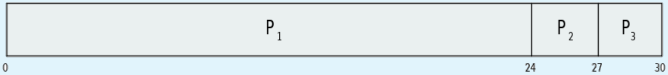



- Waiting time for $P_1 = 0$ $P_2 = 24$ $P_3 = 27$

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

| 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 $P_1 = 0 \quad P_2 = 24 \quad P_3 = 27$
- Average waiting time = $(0 + 24 + 27)/3 = 17$

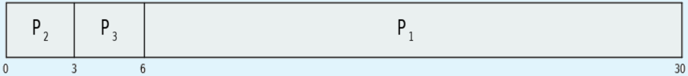
## First-Come, First-Served (FCFS) Scheduling (cont'd)

Suppose that the processes arrive in the order: $P_2$, $P_3$, $P_1$

## First-Come, First-Served (FCFS) Scheduling (cont'd)

Suppose that the processes arrive in the order: $P_2$, $P_3$, $P_1$

- The Gantt chart for the schedule is:




- Waiting time for $P_1 = 6 \quad P_2 = 0 \quad P_3 = 3$
- Average waiting time = $(6 + 0 + 3)/3 = 3$
- Much better than previous case
- Convoy effect - short process behind long process
    - Consider one CPU-bound and many I/O-bound processes

## Shortest-Job-First (SJF) Scheduling

- Associate with each process the length of its next CPU burst

## 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

## Shortest-Job-First (SJF) Scheduling

- Associate with each process the length of its next CPU burst
  - Use these lengths to schedule the process with the shortest time
- SJF is optimal – gives minimum average waiting time for a given set of processes

## Shortest-Job-First (SJF) Scheduling

- Associate with each process the length of its next CPU burst
  - Use these lengths to schedule the process with the shortest time
- SJF is optimal – gives minimum average waiting time for a given set of processes
- Preemptive version called shortest-remaining-time-first

## Shortest-Job-First (SJF) Scheduling

- Associate with each process the length of its next CPU burst
  - Use these lengths to schedule the process with the shortest time
- SJF is optimal – gives minimum average waiting time for a given set of processes
- Preemptive version called shortest-remaining-time-first
- How do we determine the length of the next CPU burst?

## Shortest-Job-First (SJF) Scheduling

- Associate with each process the length of its next CPU burst
  - Use these lengths to schedule the process with the shortest time
- SJF is optimal – gives minimum average waiting time for a given set of processes
- Preemptive version called shortest-remaining-time-first
- How do we determine the length of the next CPU burst?
  - Could ask the user

## Shortest-Job-First (SJF) Scheduling

- Associate with each process the length of its next CPU burst
  - Use these lengths to schedule the process with the shortest time
- SJF is optimal – gives minimum average waiting time for a given set of processes
- Preemptive version called shortest-remaining-time-first
- How do we determine the length of the next CPU burst?
  - Could ask the user
  - Estimate

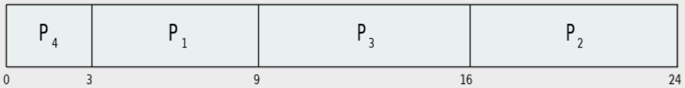
## Example (SJF Scheduling)

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

## Example (SJF Scheduling)

| 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

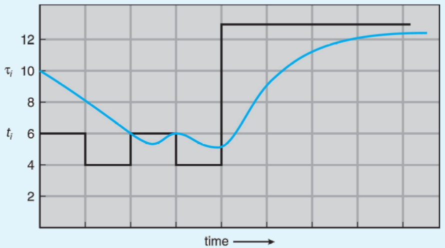
## Determining Length of Next CPU Burst

- Can only estimate the length – should be similar to the previous one
- Then pick process with shortest predicted next CPU burst
- Can be done by using the length of previous CPU bursts, using exponential averaging
  1. $t_n$ = actual length of $n^{\text{th}}$ CPU burst
  2. $\tau_{n+1}$ = predicted value for the next CPU burst
  3. $\alpha$, $0 \leqslant \alpha \leqslant 1$
  4. Define

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

- Commonly, $\alpha$ set to 1/2

## Prediction of the Length of the Next CPU Burst




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

## Example (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 for $\tau_{n+1}$, we get:

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

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

## Shortest Remaining Time First Scheduling

- Preemptive version of SJF
- Whenever a new process arrives in the ready queue, the decision on which process to schedule next is redone using the SJF algorithm.
- Is SRT more “optimal” than SJF in terms of the minimum average waiting time for a given set of processes?

## Example (Shortest-remaining-time-first)

- Now we add the concepts of varying arrival times and preemption to the analysis

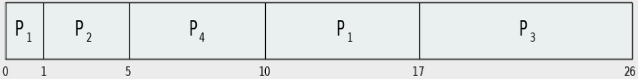
| Process | Arrival Time | Burst Time |
|---|---|---|
| $P_1$ | 0 | 8 |
| $P_2$ | 1 | 4 |
| $P_3$ | 2 | 9 |
| $P_4$ | 3 | 5 |

## Example (Shortest-remaining-time-first)

- 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

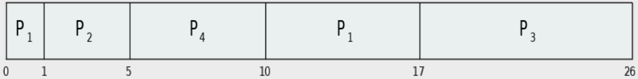


## Example (Shortest-remaining-time-first)

- 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



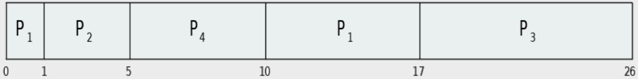

- Waiting time = completion time - execution time - arrival time

## Example (Shortest-remaining-time-first)

- 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




- Waiting time = completion time - execution time - arrival time
- Average waiting time = $[(17-8-0)+(5-4-1)+(26-9-2)+(10-5-3)]/4 = 26/4 = 6.5$

## Round Robin (RR)

- Each process gets a small unit of CPU time (time quantum $q$), usually 10-100 milliseconds
- After this time has elapsed, the process is preempted and added to the end of the ready queue
- 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 $(n-1) \times q$ time units
- Timer interrupts every quantum to schedule next process
- Performance $\begin{array}{lll} q \text{ large} & \Rightarrow & \text{FIFO (FCFS)} \\ q \text{ small} & \Rightarrow & \text{RR} \end{array}$
- Note that $q$ must be large with respect to context switch, otherwise overhead is too high

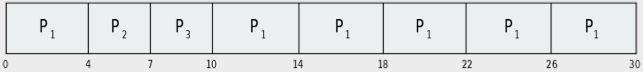
## Example (RR with Time Quantum = 4)

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

## Example (RR with Time Quantum = 4)

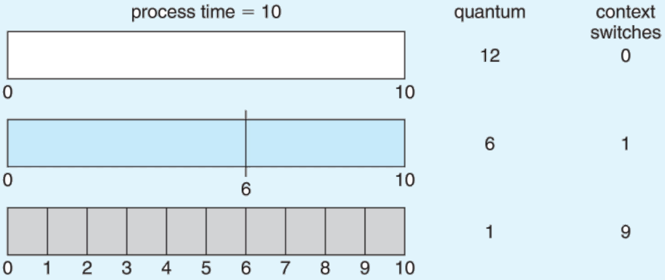
| Process | Burst Time |
|---|---|
| $P_1$ | 24 |
| $P_2$ | 3 |
| $P_3$ | 3 |

- The Gantt chart is:

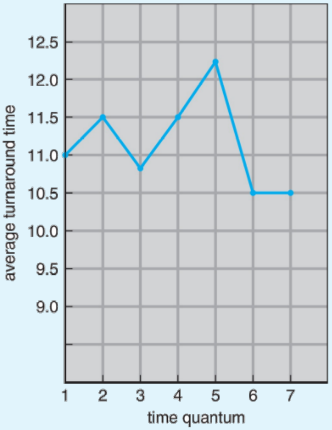



- Typically, higher average turnaround than SJF, but better response
- $q$ should be large compared to context switch time
  - $q$ usually 10 milliseconds to 100 milliseconds,
  - Context switch < 10 microseconds

## Time Quantum and Context Switch Time

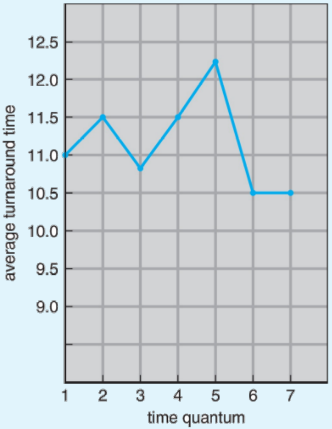


# Turnaround Time Varies With The Time Quantum




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

## Turnaround Time Varies With The Time Quantum




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

NB: 80% of CPU bursts should be shorter than $q$

## 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
- Problem ≡ Starvation – low priority processes may never execute
- Solution ≡ Aging – as time progresses increase the priority of the process

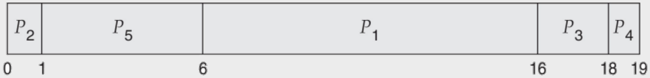
## Example

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

## Example

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

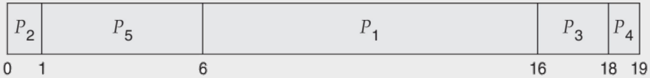
- Priority scheduling Gantt Chart



## Example

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

- Priority scheduling Gantt Chart




- Average waiting time = 8.2

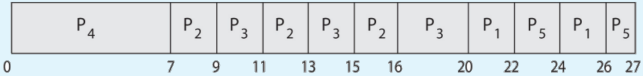
## Priority Scheduling w/ Round-Robin

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

## Priority Scheduling w/ Round-Robin

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

- Gantt Chart with time quantum = 2

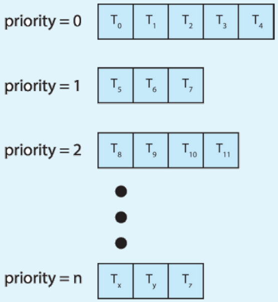


## Multilevel Queue

- The ready queue consists of multiple queues
- Multilevel queue scheduler defined by the following parameters:
  - Number of queues
  - Scheduling algorithms for each queue
  - Method used to determine which queue a process will enter when that process needs service
  - Scheduling among the queues

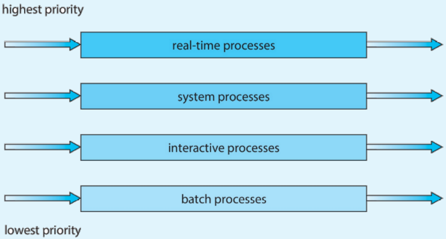
## Multilevel Queue (cont'd)

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



## Multilevel Queue (cont’d)

Prioritization based upon process type



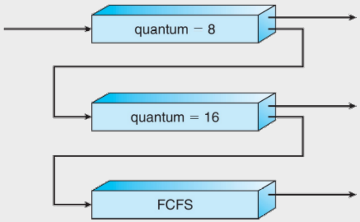
## Multilevel Queue Feedback

Prioritization based upon process type

- A process can move between the various queues.
- Multilevel-feedback-queue scheduler defined by the following parameters:
  - Number of queues
  - Scheduling algorithms for each queue
  - Method used to determine when to upgrade a process
  - Method used to determine when to demote a process
  - Method used to determine which queue a process will enter when that process needs service
- Aging can be implemented using multilevel feedback queue

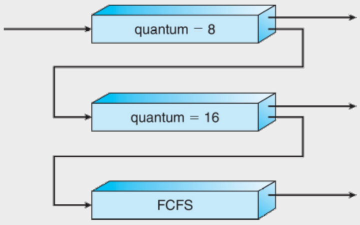
## Example (Multilevel Feedback Queue)

- Three queues:



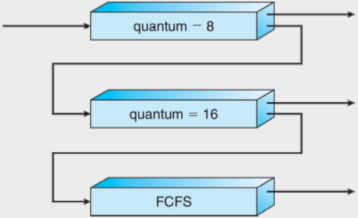
## Example (Multilevel Feedback Queue)

- Three queues:
  - $Q_0$ – RR with time quantum 8 milliseconds



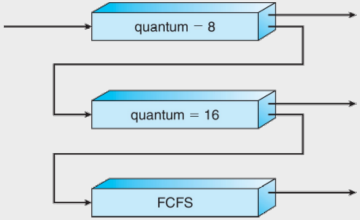
## Example (Multilevel Feedback Queue)

- Three queues:
  - $Q_0$ – RR with time quantum 8 milliseconds
  - $Q_1$ – RR time quantum 16 milliseconds



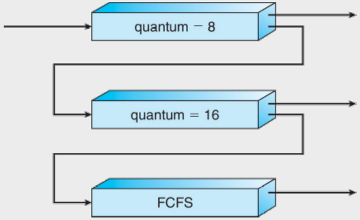
## Example (Multilevel Feedback Queue)

- Three queues:
  - $Q_0$ – RR with time quantum 8 milliseconds
  - $Q_1$ – RR time quantum 16 milliseconds
  - $Q_2$ – FCFS



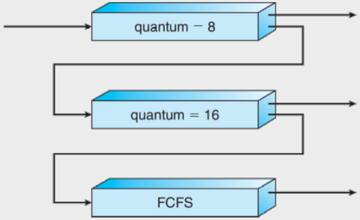
## Example (Multilevel Feedback Queue)

- Three queues:
  - $Q_0$ – RR with time quantum 8 milliseconds
  - $Q_1$ – RR time quantum 16 milliseconds
  - $Q_2$ – FCFS
- Scheduling – A new process enters queue $Q_0$ which is served in RR



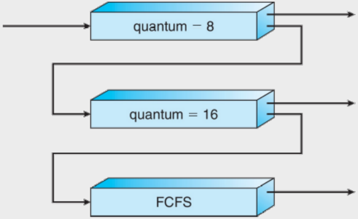
## Example (Multilevel Feedback Queue)

- Three queues:
  - $Q_0$ – RR with time quantum 8 milliseconds
  - $Q_1$ – RR time quantum 16 milliseconds
  - $Q_2$ – FCFS
- Scheduling – A new process enters queue $Q_0$ which is served in RR
  - When it gains CPU, the process receives 8 milliseconds



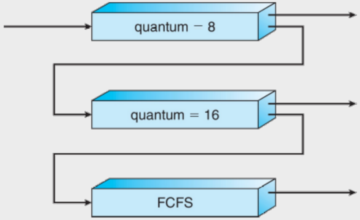
## Example (Multilevel Feedback Queue)

- Three queues:
  - $Q_0$ – RR with time quantum 8 milliseconds
  - $Q_1$ – RR time quantum 16 milliseconds
  - $Q_2$ – FCFS
- Scheduling – A new 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 $Q_1$



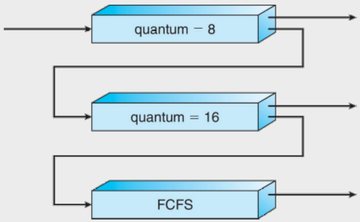
## Example (Multilevel Feedback Queue)

- Three queues:
  - $Q_0$ – RR with time quantum 8 milliseconds
  - $Q_1$ – RR time quantum 16 milliseconds
  - $Q_2$ – FCFS
- Scheduling – A new 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 $Q_1$ job is again served in RR and receives 16 additional milliseconds



## Example (Multilevel Feedback Queue)

- Three queues:
  - $Q_0$ – RR with time quantum 8 milliseconds
  - $Q_1$ – RR time quantum 16 milliseconds
  - $Q_2$ – FCFS
- Scheduling – A new 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 $Q_1$ job 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

## Thread Scheduling

- Distinction between user-level and kernel-level threads

## Thread Scheduling

- Distinction between user-level and kernel-level threads
- When threads supported, threads scheduled, not processes

## Thread Scheduling

- Distinction between user-level and kernel-level threads
- When threads supported, threads scheduled, not processes
- Many-to-one and many-to-many models, thread library schedules user-level threads to run on LWP

## Thread Scheduling

- Distinction between user-level and kernel-level threads
- When threads supported, threads scheduled, not processes
- Many-to-one and many-to-many models, thread library schedules user-level threads to run on LWP
    - Known as process-contention scope (PCS) since scheduling competition is within the process

## Thread Scheduling

- Distinction between user-level and kernel-level threads
- When threads supported, threads scheduled, not processes
- Many-to-one and many-to-many models, thread library schedules user-level threads to run on LWP
    - Known as process-contention scope (PCS) since scheduling competition is within the process
    - Typically done via priority set by programmer

## Thread Scheduling

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

## Pthread Scheduling

- API allows specifying either PCS or SCS during thread creation
  - `PTHREAD_SCOPE_PROCESS` schedules threads using PCS scheduling
  - `PTHREAD_SCOPE_SYSTEM` schedules threads using SCS scheduling
- Can be limited by OS – Linux and macOS only allow `PTHREAD_SCOPE_SYSTEM`

## Pthread Scheduling API

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

## Pthread Scheduling API (cont'd)

```
   /* 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

## Multiple-Processor Scheduling

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

## Multiple-Processor Scheduling (cont'd)

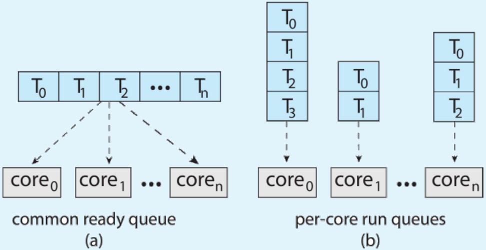
- Symmetric multiprocessing (SMP) is where each processor is self scheduling

## Multiple-Processor Scheduling (cont'd)

- Symmetric multiprocessing (SMP) is where each processor is self scheduling
- All threads may be in a common ready queue (a)

## Multiple-Processor Scheduling (cont'd)

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



## Multicore Processors

- Recent trend to place multiple processor cores on same physical chip

## Multicore Processors

- Recent trend to place multiple processor cores on same physical chip
- Faster and consumes less power

## Multicore Processors

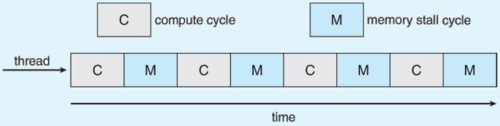
- Recent trend to place multiple processor cores on same physical chip
- Faster and consumes less power
- Multiple threads per core also growing

## Multicore Processors

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

## Multicore Processors

- Recent trend to place multiple processor cores on same physical chip
- Faster and consumes less power
- Multiple threads per core also growing
    - Takes advantage of memory stall to make progress on another thread while memory retrieve happens
- Figure



## Multithreaded Multicore System

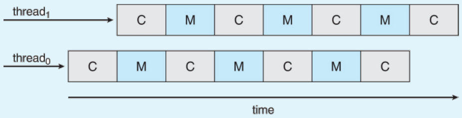
- Each core has > 1 hardware threads

## Multithreaded Multicore System

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

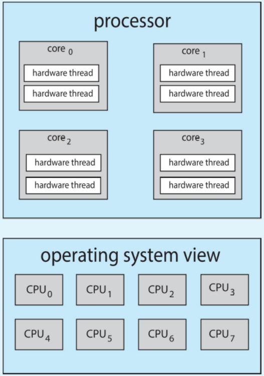
## Multithreaded Multicore System

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



## Multithreaded Multicore System (cont'd)

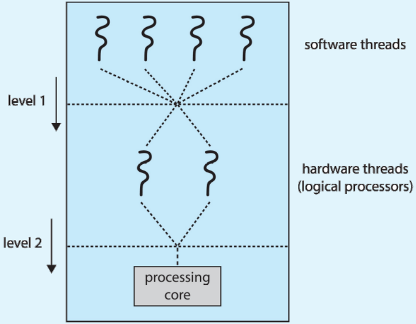
- Chip-multithreading (CMT) assigns each core multiple hardware threads – Intel refers to this as hyperthreading
- On a quad-core system with 2 hardware threads per core, the operating system sees 8 logical processors



## Multithreaded Multicore System (cont'd)

Two levels of scheduling:

1. The operating system deciding which software thread to run on a logical CPU
2. How each core decides which hardware thread to run on the physical core



## Multiple-Processor Scheduling – Load Balancing

- If SMP, need to keep all CPUs loaded for efficiency

## Multiple-Processor Scheduling – Load Balancing

- If SMP, need to keep all CPUs loaded for efficiency
- Load balancing attempts to keep workload evenly distributed

## Multiple-Processor Scheduling – Load Balancing

- If SMP, need to keep all CPUs loaded for efficiency
- Load balancing attempts to keep workload evenly distributed
- Push migration – periodic task checks load on each processor, and if found pushes task from overloaded CPU to other CPUs

## Multiple-Processor Scheduling – Load Balancing

- If SMP, need to keep all CPUs loaded for efficiency
- Load balancing attempts to keep workload evenly distributed
- Push migration – periodic task checks load on each processor, and if found pushes task from overloaded CPU to other CPUs
- Pull migration – idle processors pulls waiting task from busy processor

## Multiple-Processor Scheduling – Processor Affinity

- When a thread has been running on one processor, the cache contents of that processor stores the memory accesses by that thread

## Multiple-Processor Scheduling – Processor Affinity

- When a thread has been running on one processor, the cache contents of that processor stores the memory accesses by that thread
- We refer to this as a thread having affinity for a processor (i.e., “processor affinity”)

## Multiple-Processor Scheduling – Processor Affinity

- When a thread has been running on one processor, the cache contents of that processor stores the memory accesses by that thread
- We refer to this as a thread having affinity for a processor (i.e., "processor affinity")
- Load balancing may affect processor affinity as a thread may be moved from one processor to another to balance loads, yet that thread loses the contents of what it had in the cache of the processor it was moved off of

## Multiple-Processor Scheduling – Processor Affinity

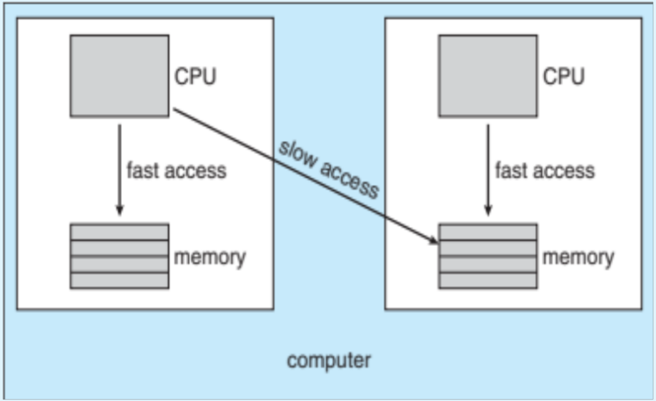
- When a thread has been running on one processor, the cache contents of that processor stores the memory accesses by that thread
- We refer to this as a thread having affinity for a processor (i.e., “processor affinity”)
- Load balancing may affect processor affinity as a thread may be moved from one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

## Multiple-Processor Scheduling – Processor Affinity

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

## NUMA and CPU Scheduling

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



# Outline

## Real-Time CPU Scheduling

- Can present obvious challenges

## Real-Time CPU Scheduling

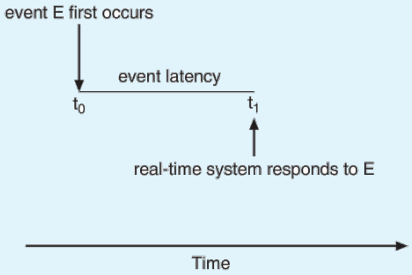
- Can present obvious challenges
- Soft real-time systems – Critical real-time tasks have the highest priority, but no guarantee as to when tasks will be scheduled

## Real-Time CPU Scheduling

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

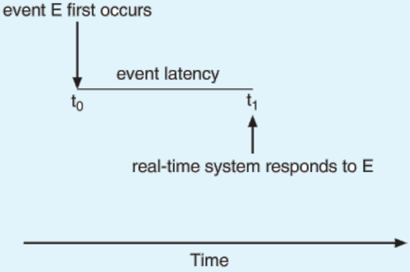
## Real-Time CPU Scheduling (cont'd)

- Event latency – the amount of time that elapses from when an event occurs to when it is serviced.



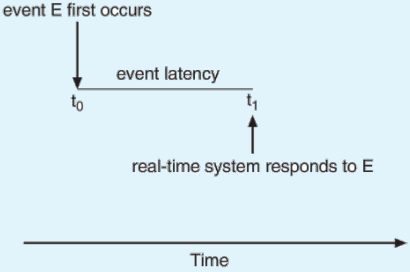
## Real-Time CPU Scheduling (cont'd)

- Event latency – the amount of time that elapses from when an event occurs to when it is serviced.
- Two types of latencies affect performance



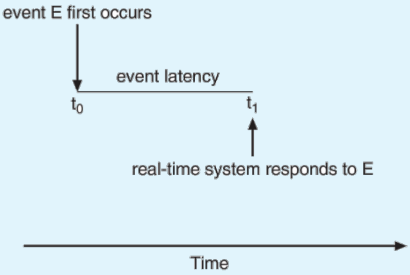
## Real-Time CPU Scheduling (cont'd)

- Event latency – the amount of time that elapses from when an event occurs to when it is serviced.
- Two types of latencies affect performance
  1. Interrupt latency – time from arrival of interrupt to start of routine that services interrupt

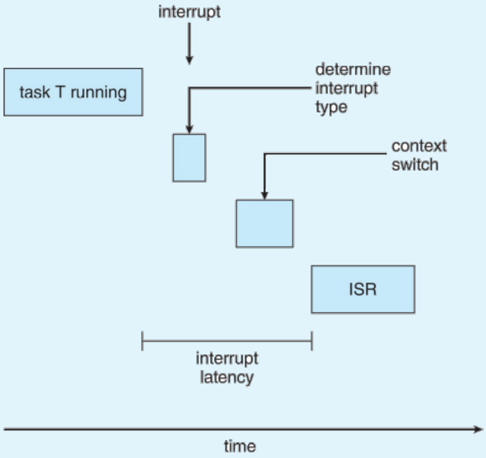


## Real-Time CPU Scheduling (cont'd)

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



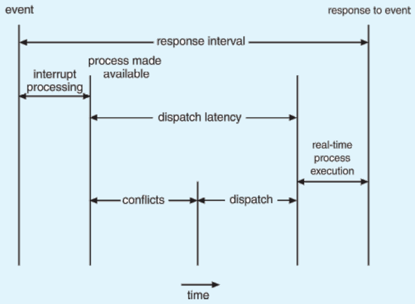
## Interrupt Latency



## Dispatch Latency

Conflict phase of dispatch latency:

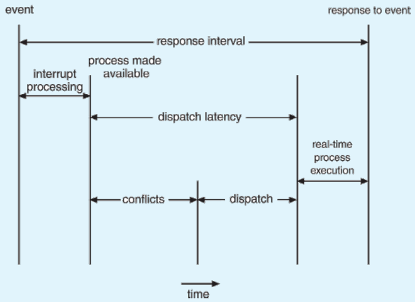
1. Preemption of any process running in kernel mode



## Dispatch Latency

Conflict phase of dispatch latency:

1. Preemption of any process running in kernel mode
2. Release by low-priority process of resources needed by high-priority processes



## Priority-based Scheduling

- For real-time scheduling, scheduler must support preemptive, priority-based scheduling

## Priority-based Scheduling

- For real-time scheduling, scheduler must support preemptive, priority-based scheduling
  - But only guarantees soft real-time

## 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

## 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

## Priority-based Scheduling

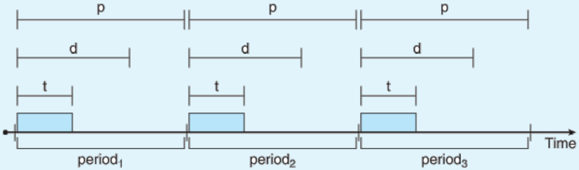
- For real-time scheduling, scheduler must support preemptive, priority-based scheduling
  - But only guarantees soft real-time
- For hard real-time must also provide ability to meet deadlines
- Processes have new characteristics: periodic ones require CPU at constant intervals
  - Has processing time $t$, deadline $d$, period $p$

## 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 $p$
  - $0 \leqslant t \leqslant d \leqslant p$

## 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 $p$
  - $0 \leqslant t \leqslant d \leqslant p$
  - Rate of periodic task is $1/p$



## Rate Monotonic Scheduling

- A priority is assigned based on the inverse of its period

## Rate Monotonic Scheduling

- A priority is assigned based on the inverse of its period
- Shorter periods = higher priority

## Rate Monotonic Scheduling

- A priority is assigned based on the inverse of its period
- Shorter periods = higher priority
- Longer periods = lower priority

## Rate Monotonic Scheduling

- A priority is assigned based on the inverse of its period
- Shorter periods = higher priority
- Longer periods = lower priority
-

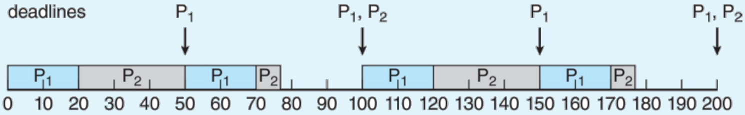
| process | period ($P$) | deadline ($d$) | processing time ($t$) |
|---|---|---|---|
| $P_1$ | 50 | 50 | 20 |
| $P_2$ | 100 | 100 | 35 |

## Rate Monotonic Scheduling

- A priority is assigned based on the inverse of its period
- Shorter periods = higher priority
- Longer periods = lower priority
-

| process | period ($P$) | deadline ($d$) | processing time ($t$) |
|---|---|---|---|
| $P_1$ | 50 | 50 | 20 |
| $P_2$ | 100 | 100 | 35 |

- $P_1$ is assigned a higher priority than $P_2$



## Missed Deadlines with Rate Monotonic Scheduling

| process | period ($P$) | deadline ($d$) | processing time ($t$) |
|---|---|---|---|
| $P_1$ | 50 | 50 | 25 |
| $P_2$ | 100 | 80 | 35 |

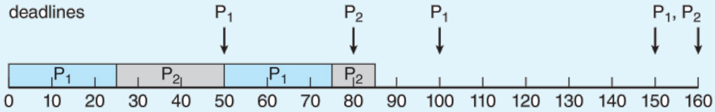
## Missed Deadlines with Rate Monotonic Scheduling

| process | period ($P$) | deadline ($d$) | processing time ($t$) |
|---|---|---|---|
| $P_1$ | 50 | 50 | 25 |
| $P_2$ | 100 | 80 | 35 |

- Process $P_2$ misses finishing its deadline at time 80

## Missed Deadlines with Rate Monotonic Scheduling

- | process | period ($P$) | deadline ($d$) | processing time ($t$) |
  |---|---|---|---|
  | $P_1$ | 50 | 50 | 25 |
  | $P_2$ | 100 | 80 | 35 |
- Process $P_2$ misses finishing its deadline at time 80
- Figure



## Earliest Deadline First Scheduling (EDF)

- Priorities are assigned according to deadlines:

## Earliest Deadline First Scheduling (EDF)

- Priorities are assigned according to deadlines:
  - The earlier the deadline, the higher the priority

## 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

## 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
-

| process | period ($P$) | deadline ($d$) | processing time ($t$) |
|---|---|---|---|
| $P_1$ | 50 | 50 | 25 |
| $P_2$ | 100 | 80 | 35 |

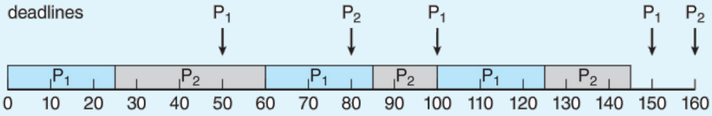
## Earliest Deadline First Scheduling (EDF)

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

- 

| process | period ($P$) | deadline ($d$) | processing time ($t$) |
|---|---|---|---|
| $P_1$ | 50 | 50 | 25 |
| $P_2$ | 100 | 80 | 35 |

- Figure



## Proportional Share Scheduling

- $T$ shares are allocated among all processes in the system

## Proportional Share Scheduling

- $T$ shares are allocated among all processes in the system
- An application receives $N$ shares where $N < T$

## Proportional Share Scheduling

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

## POSIX Real-Time Scheduling

- The POSIX.1b standard

## POSIX Real-Time Scheduling

- The POSIX.1b standard
- API provides functions for managing real-time threads

## POSIX Real-Time Scheduling

- The POSIX.1b standard
- API provides functions for managing real-time threads
- Defines two scheduling classes for real-time threads:

## POSIX Real-Time Scheduling

- The POSIX.1b standard
- API provides functions for managing real-time threads
- 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

## POSIX Real-Time Scheduling

- The POSIX.1b standard
- API provides functions for managing real-time threads
- 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

## POSIX Real-Time Scheduling

- The POSIX.1b standard
- API provides functions for managing real-time threads
- 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
- Defines two functions for getting and setting scheduling policy:

## POSIX Real-Time Scheduling

- The POSIX.1b standard
- API provides functions for managing real-time threads
- 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
- Defines two functions for getting and setting scheduling policy:
  1. `pthread_attr_getsched_policy(pthread_attr_t *attr, int *policy)`

## POSIX Real-Time Scheduling

- The POSIX.1b standard
- API provides functions for managing real-time threads
- 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
- 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)`

## POSIX Real-Time Scheduling API

```c
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5
int main(int argc, char *argv[])
{
   int i, policy;
   pthread_t_tid[NUM_THREADS];
   pthread_attr_t attr;
   /* get the default attributes */
   pthread_attr_init(&attr);
   /* get the current scheduling policy */
   if (pthread_attr_getschedpolicy(&attr, &policy) != 0)
      fprintf(stderr, "Unable to get policy.\n");
   else {
      if (policy == SCHED_OTHER) printf("SCHED_OTHER\n");
      else if (policy == SCHED_RR) printf("SCHED_RR\n");
      else if (policy == SCHED_FIFO) printf("SCHED_FIFO\n");
   }
```

## POSIX Real-Time Scheduling API (cont'd)

```
   /* 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

## Linux Scheduling Through Version 2.5

- Prior to kernel version 2.5, ran variation of standard UNIX scheduling algorithm
- Version 2.5 moved to constant order $O(1)$ scheduling time
  - Preemptive, priority based
  - Two priority ranges: time-sharing and real-time
  - Real-time range from 0 to 99 and nice value from 100 to 140
  - Map into global priority with numerically lower values indicating higher priority
  - Higher priority gets larger q
  - Task run-able as long as time left in time slice (active)
  - If no time left (expired), not run-able until all other tasks use their slices
  - All run-able tasks tracked in per-CPU runqueue data structure
    - Two priority arrays (active, expired)
    - Tasks indexed by priority
    - When no more active, arrays are exchanged
  - Worked well, but poor response times for interactive processes

## Linux Scheduling in Version 2.6.23 +

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

## Linux Scheduling in Version 2.6.23 + (cont's)

- 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

## Linux Scheduling

- Real-time scheduling according to POSIX.1b

## Linux Scheduling

- Real-time scheduling according to POSIX.1b
  - Real-time tasks have static priorities

## Linux Scheduling

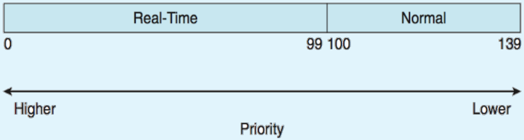
- Real-time scheduling according to POSIX.1b
  - Real-time tasks have static priorities
- Real-time plus normal map into global priority scheme

## 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

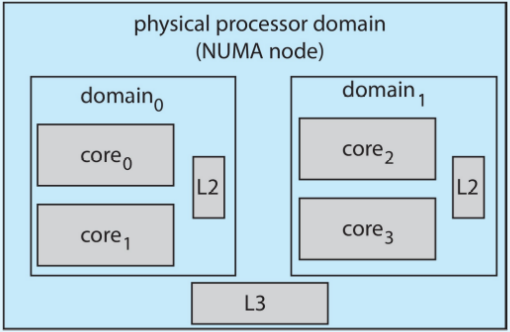
## Linux Scheduling

- 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



## Linux Scheduling (cont'd)

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



## Windows Scheduling

- Windows uses priority-based preemptive scheduling

## Windows Scheduling

- Windows uses priority-based preemptive scheduling
- Highest-priority thread runs next

## Windows Scheduling

- Windows uses priority-based preemptive scheduling
- Highest-priority thread runs next
- Dispatcher is scheduler

## 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

## 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

## 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

## 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

## 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

## 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

## Windows Scheduling

- Windows uses priority-based preemptive scheduling
- Highest-priority thread runs next
- Dispatcher is scheduler
- Thread runs until (1) blocks, (2) uses time slice, (3) preempted by higher-priority thread
- Real-time threads can preempt non-real-time
- 32-level priority scheme
- Variable class is 1-15, real-time class is 16-31
- Priority 0 is memory-management thread
- Queue for each priority
- If no run-able thread, runs idle thread

## Windows Priority Classes

- Win32 API identifies several priority classes to which a process can belong

## 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`

## 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

## 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

## 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`

## 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

## 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

## Windows Priority Classes

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

## Windows Priorities

| | real-time | high | above normal | normal | below normal | idle priority |
|---|---|---|---|---|---|---|
| time-critical | 31 | 15 | 15 | 15 | 15 | 15 |
| highest | 26 | 15 | 12 | 10 | 8 | 6 |
| above normal | 25 | 14 | 11 | 9 | 7 | 5 |
| normal | 24 | 13 | 10 | 8 | 6 | 4 |
| below normal | 23 | 12 | 9 | 7 | 5 | 3 |
| lowest | 22 | 11 | 8 | 6 | 4 | 2 |
| idle | 16 | 1 | 1 | 1 | 1 | 1 |

# Outline

## Algorithm Evaluation

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

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

## Deterministic Evaluation

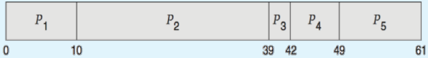
- For each algorithm, calculate minimum average waiting time

## Deterministic Evaluation

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

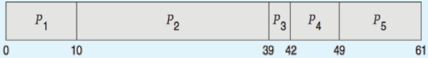
## Deterministic Evaluation

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

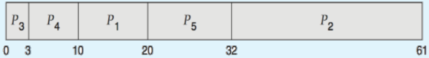


## Deterministic Evaluation

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

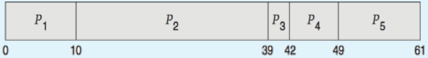



  - Non-preemptive SJF is 13ms:

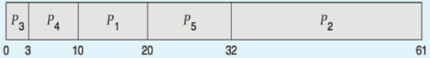


## Deterministic Evaluation

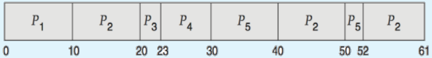
- For each algorithm, calculate minimum average waiting time
- Simple and fast, but requires exact numbers for input, applies only to those inputs
  - FCFS is 28ms:




  - Non-preemptive SJF is 13ms:




  - RR is 23ms:



## Queueing Models

- Describes the arrival of processes, and CPU and I/O bursts probabilistically

## Queueing Models

- Describes the arrival of processes, and CPU and I/O bursts probabilistically
    - Commonly exponential, and described by mean

## 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.

## 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

## 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

## Queueing Models

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

## Little’s Formula

- $n$ = average queue length

## Little's Formula

- $n$ = average queue length
- $W$ = average waiting time in queue

## Little's Formula

- $n$ = average queue length
- $W$ = average waiting time in queue
- $\lambda$ = average arrival rate into queue

## Little's Formula

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

$$n = \lambda \times W$$

## Little's Formula

- $n$ = average queue length
- $W$ = average waiting time in queue
- $\lambda$ = 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

## Little's Formula

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

$$n = \lambda \times W$$

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

## Simulations

- Queueing models limited

## Simulations

- Queueing models limited
- Simulations more accurate

## Simulations

- Queueing models limited
- Simulations more accurate
  - Programmed model of computer system

## Simulations

- Queueing models limited
- Simulations more accurate
  - Programmed model of computer system
  - Clock is a variable

## Simulations

- Queueing models limited
- Simulations more accurate
  - Programmed model of computer system
  - Clock is a variable
  - Gather statistics indicating algorithm performance

## 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

## 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

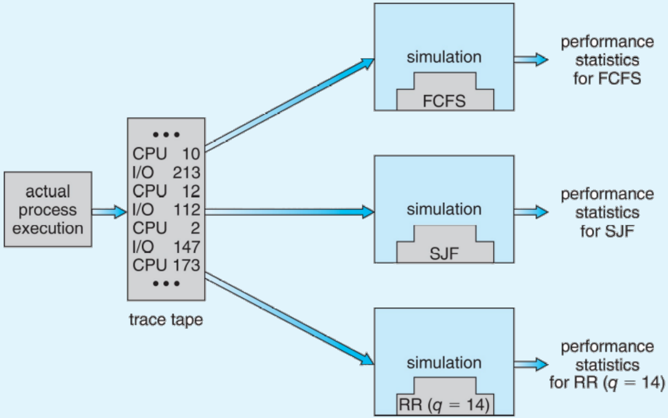
## Simulations

- 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

## Simulations

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

## Evaluation of CPU Schedulers by Simulation



## Implementations

- Even simulations have limited accuracy

## Implementations

- Even simulations have limited accuracy
- Just implement new scheduler and test in real systems

## Implementations

- Even simulations have limited accuracy
- Just implement new scheduler and test in real systems
  - High cost, high risk

## Implementations

- Even simulations have limited accuracy
- Just implement new scheduler and test in real systems
  - High cost, high risk
  - Environments vary

## Implementations

- 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

## Implementations

- 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

## Implementations

- 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

Thanks! & Questions?