CPU Scheduling (Part I)

Amir H. Payberah
amir@sics.se

Amirkabir University of Technology
(Tehran Polytechnic)

Motivation

CPU Scheduling

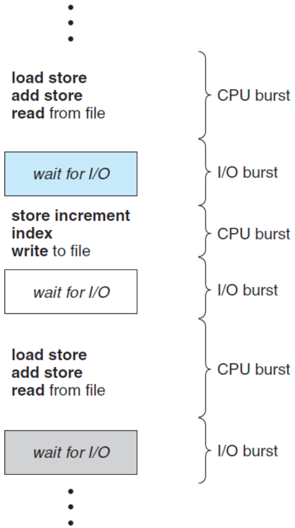
- CPU scheduling is the basis of multiprogrammed OSs.
- By switching the CPU among processes, the OS makes the computer more productive.

Basic Concepts

- In a single-processor system, only one process can run at a time.
- Others must wait until the CPU is free and can be rescheduled.
- The objective of multiprogramming is to have some process running at all times, to maximize CPU utilization.

Basic Concepts

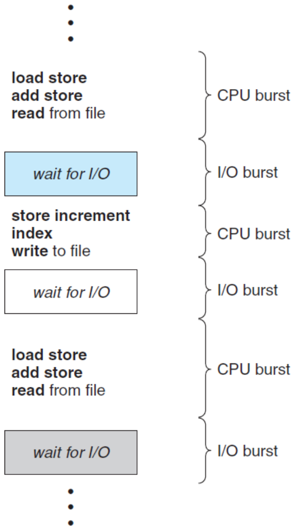
- CPU-I/O burst cycle: process execution consists of a cycle of CPU execution and I/O wait.



Basic Concepts

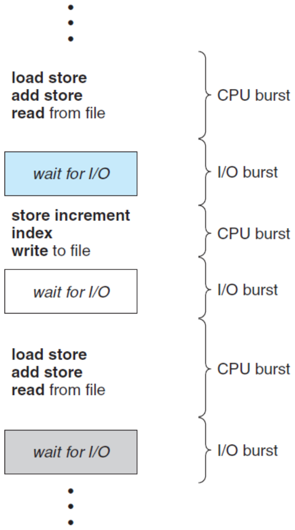
- CPU-I/O burst cycle: process execution consists of a cycle of CPU execution and I/O wait.

- CPU burst followed by I/O burst.

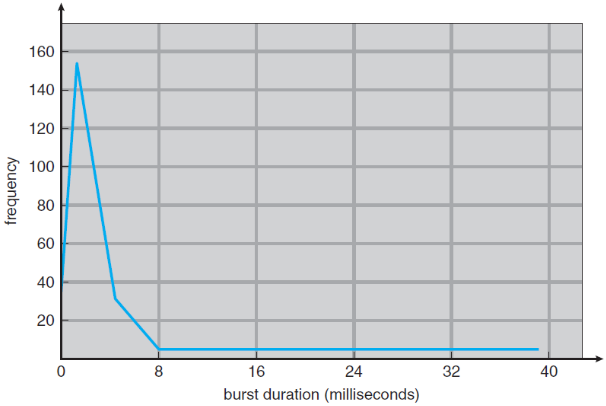


Basic Concepts

- 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



CPU Scheduler

- Short-term scheduler selects from among the processes in ready queue, and allocates the CPU to one of them.

CPU Scheduler

- Short-term scheduler selects from among the processes in ready queue, and allocates the CPU to one of them.

- CPU scheduling decisions may take place when a process:
  1. Switches from running to waiting (e.g., an I/O request).
  2. Switches from running to ready (e.g., interrupt).
  3. Switches from waiting to ready (e.g., I/O completion).
  4. Terminates.

CPU Scheduler

- Short-term scheduler selects from among the processes in ready queue, and allocates the CPU to one of them.

- CPU scheduling decisions may take place when a process:
  1. Switches from running to waiting (e.g., an I/O request).
  2. Switches from running to ready (e.g., interrupt).
  3. Switches from waiting to ready (e.g., I/O completion).
  4. Terminates.

- For situations 1 and 4, there is no choice in terms of scheduling. A new process must be selected for execution. There is a choice, however, for situations 2 and 3.

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

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

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

Scheduling Criteria (1/2)

- Different CPU-scheduling algorithms have different properties.
- The choice of a particular algorithm may favor one class of processes over another.

Scheduling Criteria (1/2)

- Different CPU-scheduling algorithms have different properties.
- The choice of a particular algorithm may favor one class of processes over another.
- CPU utilization: keep the CPU as busy as possible .

Scheduling Criteria (1/2)

- Different CPU-scheduling algorithms have different properties.
- The choice of a particular algorithm may favor one class of processes over another.
- CPU utilization: keep the CPU as busy as possible (Max).

Scheduling Criteria (1/2)

- Different CPU-scheduling algorithms have different properties.
- The choice of a particular algorithm may favor one class of processes over another.
- CPU utilization: keep the CPU as busy as possible (Max).
- Throughput: # of processes that complete their execution per time unit .

Scheduling Criteria (1/2)

- Different CPU-scheduling algorithms have different properties.
- The choice of a particular algorithm may favor one class of processes over another.
- CPU utilization: keep the CPU as busy as possible (Max).
- Throughput: # of processes that complete their execution per time unit (Max).

Scheduling Criteria (2/2)

- Turnaround time: amount of time to execute a particular process .

Scheduling Criteria (2/2)

- Turnaround time: amount of time to execute a particular process (Min).

Scheduling Criteria (2/2)

- Turnaround time: amount of time to execute a particular process (Min).

- Waiting time: amount of time a process has been waiting in the ready queue .

Scheduling Criteria (2/2)

- Turnaround time: amount of time to execute a particular process (Min).
- Waiting time: amount of time a process has been waiting in the ready queue (Min).

Scheduling Criteria (2/2)

- Turnaround time: amount of time to execute a particular process (Min).
- Waiting time: amount of time a process has been waiting in the ready queue (Min).
- 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 (2/2)

- Turnaround time: amount of time to execute a particular process (Min).
- Waiting time: amount of time a process has been waiting in the ready queue (Min).
- 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) (Min).

Scheduling Algorithms

Scheduling Algorithms

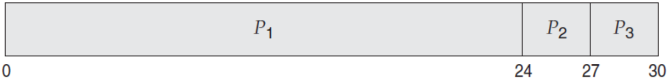
- First-Come, First-Served Scheduling
- Shortest-Job-First Scheduling
- Priority Scheduling
- Round-Robin Scheduling
- Multilevel Queue Scheduling
- Multilevel Feedback Queue Scheduling

First-Come, First-Served (FCFS) Scheduling

FCFS Scheduling (1/2)

Process	Burst Time
P_1	24
P_2	3
P_3	3

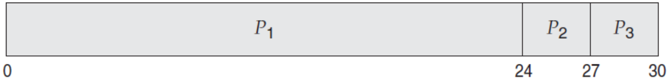
- Suppose that the processes arrive in the order: P_1, P_2, P_3
- The gantt chart for the schedule is:



FCFS Scheduling (1/2)

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:

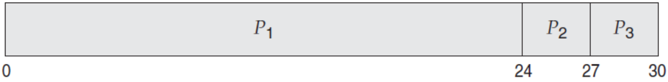


- Waiting time for $P_1 = 0$; $P_2 = 24$; $P_3 = 27$

FCFS Scheduling (1/2)

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:

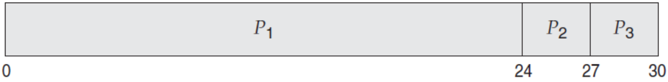


- Waiting time for $P_1 = 0$; $P_2 = 24$; $P_3 = 27$
- Average waiting time: $\frac{0+24+27}{3} = 17$

FCFS Scheduling (1/2)

Process	Burst Time
P_1	24
P_2	3
P_3	3

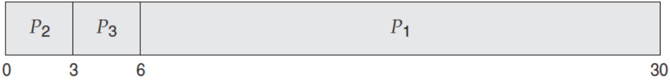
- Suppose that the processes arrive in the order: P_1, P_2, P_3
- The gantt chart for the schedule is:



- Waiting time for $P_1 = 0$; $P_2 = 24$; $P_3 = 27$
- Average waiting time: $\frac{0+24+27}{3} = 17$
- FCFS scheduling algorithm is non-preemptive: process keeps the CPU until it releases the CPU, either requesting I/O.

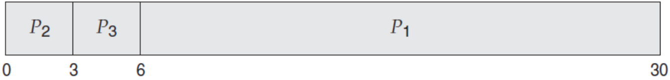
FCFS Scheduling (2/2)

- Suppose that the processes arrive in the order: P_2, P_3, P_1
- The gantt chart for the schedule is:



FCFS Scheduling (2/2)

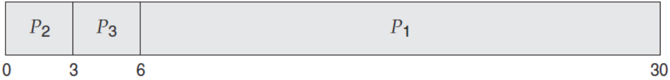
- Suppose that the processes arrive in the order: P_2, P_3, P_1
- The gantt chart for the schedule is:



- Waiting time for $P_1 = 6$; $P_2 = 0$; $P_3 = 3$

FCFS Scheduling (2/2)

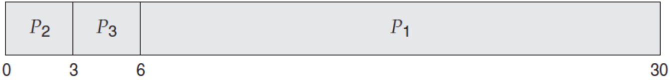
- Suppose that the processes arrive in the order: P_2, P_3, P_1
- The gantt chart for the schedule is:



- Waiting time for $P_1 = 6$; $P_2 = 0$; $P_3 = 3$
- Average waiting time: $\frac{6+0+3}{3} = 3$

FCFS Scheduling (2/2)

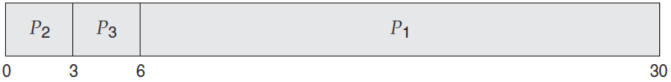
- Suppose that the processes arrive in the order: P_2, P_3, P_1
- The gantt chart for the schedule is:



- Waiting time for $P_1 = 6$; $P_2 = 0$; $P_3 = 3$
- Average waiting time: $\frac{6+0+3}{3} = 3$
- Much better than previous case.

FCFS Scheduling (2/2)

- 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: $\frac{6+0+3}{3} = 3$
- 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

SJF Scheduling (1/2)

- Associate with each process the length of its next CPU burst.
  - Use these lengths to schedule the process with the shortest time.

SJF Scheduling (1/2)

- 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.

SJF Scheduling (1/2)

- Associate with each process the length of its next CPU burst.
  - Use these lengths to schedule the process with the shortest time.

- SJF is optimal - gives minimum average waiting time for a given set of processes.
  - The difficulty is knowing the length of the next CPU request.

SJF Scheduling (1/2)

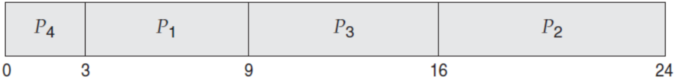
- Associate with each process the length of its next CPU burst.
  - Use these lengths to schedule the process with the shortest time.

- SJF is optimal - gives minimum average waiting time for a given set of processes.
  - The difficulty is knowing the length of the next CPU request.
  - Could ask the user (batch systems with long-term scheduling).

SJF Scheduling (2/2)

Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3

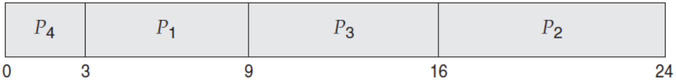
- The gantt chart for the schedule is:



SJF Scheduling (2/2)

Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3

- The gantt chart for the schedule is:

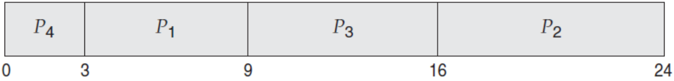


- Waiting time for $P_1 = 3;\ P_2 = 16;\ P_3 = 9,\ P_4 = 0$

SJF Scheduling (2/2)

Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3

- The gantt chart for the schedule is:



- Waiting time for $P_1 = 3;\ P_2 = 16;\ P_3 = 9,\ P_4 = 0$

- Average waiting time: $\frac{3+16+9+0}{4} = 7$

Determining Length of Next CPU Burst (1/2)

- Estimate the length, and pick process with shortest predicted next CPU burst.

Determining Length of Next CPU Burst (1/2)

- Estimate the length, and pick process with shortest predicted next CPU burst.
- The next CPU burst: predicted as an exponential average of the measured lengths of previous CPU bursts.

Determining Length of Next CPU Burst (1/2)

- Estimate the length, and pick process with shortest predicted next CPU burst.

- The next CPU burst: predicted as an exponential average of the measured lengths of previous CPU bursts.
  1. t_n = actual length of n^{th} CPU burst

Determining Length of Next CPU Burst (1/2)

- Estimate the length, and pick process with shortest predicted next CPU burst.

- The next CPU burst: predicted as an exponential average of the measured lengths of previous CPU bursts.
  1. t_n = actual length of n^{th} CPU burst
  2. τ_{n+1} = predicted value for the next CPU burst

Determining Length of Next CPU Burst (1/2)

- Estimate the length, and pick process with shortest predicted next CPU burst.
- The next CPU burst: predicted as an exponential average of the measured lengths of previous CPU bursts.
  1. t_n = actual length of n^{th} CPU burst
  2. τ_{n+1} = predicted value for the next CPU burst
  3. $\alpha, 0 \leq \alpha \leq 1$

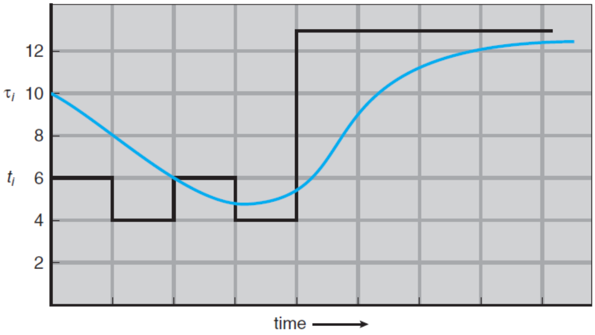
Determining Length of Next CPU Burst (1/2)

- Estimate the length, and pick process with shortest predicted next CPU burst.
- The next CPU burst: predicted as an exponential average of the measured lengths of previous CPU bursts.
  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: $\tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n$

Determining Length of Next CPU Burst (1/2)

- Estimate the length, and pick process with shortest predicted next CPU burst.
- The next CPU burst: predicted as an exponential average of the measured lengths of previous CPU bursts.
  1. t_n = actual length of n^{th} CPU burst
  2. τ_{n+1} = predicted value for the next CPU burst
  3. $\alpha, 0 \leq \alpha \leq 1$
  4. Define: $\tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n$
- Commonly, α set to $\frac{1}{2}$

Determining Length of Next CPU Burst (2/2)



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

Examples of Exponential Averaging

- $\alpha = 0$

Examples of Exponential Averaging

- $\alpha = 0$
  - $\tau_{n+1} = \tau_n$
  - Recent history does not count

Examples of Exponential Averaging

- $\alpha = 0$
  - $\tau_{n+1} = \tau_n$
  - Recent history does not count
- $\alpha = 1$

Examples of Exponential Averaging

- $\alpha = 0$
  - $\tau_{n+1} = \tau_n$
  - Recent history does not count
- $\alpha = 1$
  - $\tau_{n+1} = t_n$
  - Only the actual last CPU burst counts

Examples of Exponential Averaging

- $\alpha = 0$
  - $\tau_{n+1} = \tau_n$
  - Recent history does not count
- $\alpha = 1$
  - $\tau_{n+1} = t_n$
  - Only the actual last CPU burst counts
- If we expand the formula, we get:
  - $\tau_{n+1} = \alpha t_n + (1-\alpha)\alpha t_{n-1} + \cdots$
    $+(1-\alpha)^j \alpha t_{n-j} + \cdots$
    $+(1-\alpha)^{n+1}\tau_0$
  - Since both α and $(1-\alpha)$ are less than or equal to 1, each successive term has less weight than its predecessor.

Preemptive SJF

- The SJF algorithm can be either preemptive or non-preemptive.

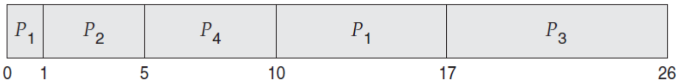
Preemptive SJF

- The SJF algorithm can be either preemptive or non-preemptive.
- Preemptive version called shortest-remaining-time-first

Example of Shortest-Remaining-Time-First

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

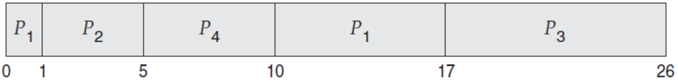
- Now we add the concepts of varying arrival times and preemption to the analysis.
- The gantt chart for the schedule is:



Example of Shortest-Remaining-Time-First

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

- Now we add the concepts of varying arrival times and preemption to the analysis.
- The gantt chart for the schedule is:



- Average waiting time: $\frac{(10-1)+(1-1)+(17-2)+(5-3)}{4} = \frac{26}{4} = 6.5$

Priority Scheduling

Priority Scheduling (1/2)

- A priority number (integer) is associated with each process.

Priority Scheduling (1/2)

- 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 and non-preemptive

Priority Scheduling (1/2)

- 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 and non-preemptive
- SJF is priority scheduling where priority is the inverse of predicted next CPU burst time.

Priority Scheduling (1/2)

- 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 and non-preemptive
- SJF is priority scheduling where priority is the inverse of predicted next CPU burst time.
- Problem: starvation - low priority processes may never execute

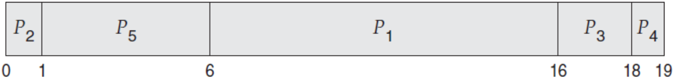
Priority Scheduling (1/2)

- 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 and non-preemptive
- 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

Priority Scheduling (2/2)

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

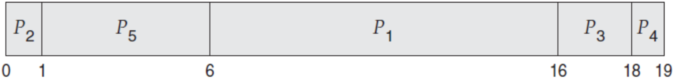
- The gantt chart for the schedule is:



Priority Scheduling (2/2)

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

- The gantt chart for the schedule is:



- Average waiting time: $\frac{0+1+6+16+18}{5} = 8.2$

Round-Robin (RR) Scheduling

RR Scheduling (1/3)

- Each process gets a small unit of CPU time (time quantum q), usually 10-100 milliseconds.

RR Scheduling (1/3)

- 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.

RR Scheduling (1/3)

- 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.

RR Scheduling (1/3)

- 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.

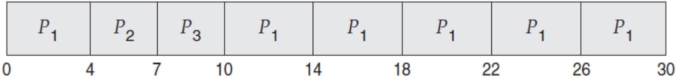
RR Scheduling (2/3)

- Timer interrupts every quantum to schedule next process.
- Typically, higher average turnaround than SJF, but better response time.

RR Scheduling (3/3)

Process	Burst Time
P_1	24
P_2	3
P_3	3

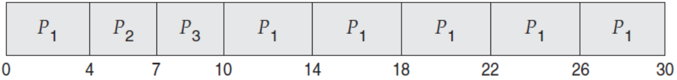
- Time quantum $q = 4$
- The gantt chart for the schedule is:



RR Scheduling (3/3)

Process	Burst Time
P_1	24
P_2	3
P_3	3

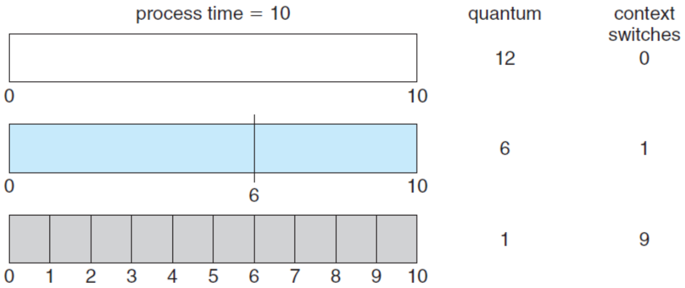
- Time quantum $q = 4$
- The gantt chart for the schedule is:



- Average waiting time: $\frac{(10-4)+4+7}{3} = 5.66$

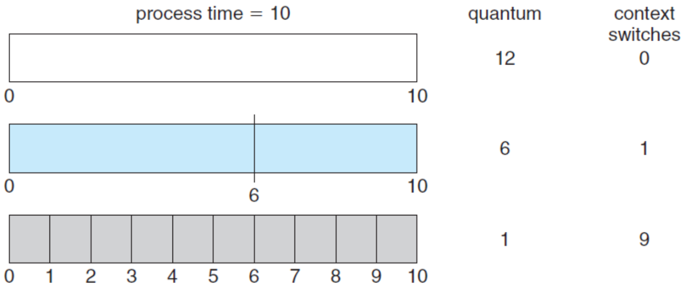
Time Quantum and Context Switch Time

- q large $\Rightarrow$ FIFO

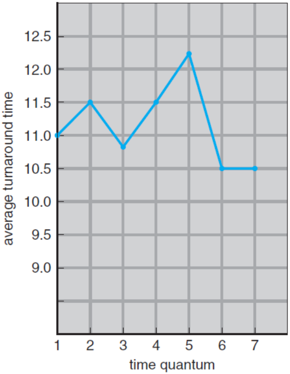


Time Quantum and Context Switch Time

- q large $\Rightarrow$ FIFO
- q small $\Rightarrow$ q must be large with respect to context switch, otherwise overhead is too high.



Turnaround Time Varies With The Time Quantum



process	time
P_1	6
P_2	3
P_3	1
P_4	7

- Turnaround time depends on the size of the time quantum.
- The average turnaround time can be improved if most processes finish their next CPU burst in a single time quantum.
- 80% of CPU bursts should be shorter than q.

Multilevel Queue Scheduling

Multilevel Queue Scheduling (1/3)

- Ready queue is partitioned into separate queues, e.g.:
  - foreground (interactive)
  - background (batch)

Multilevel Queue Scheduling (1/3)

- Ready queue is partitioned into separate queues, e.g.:
  - foreground (interactive)
  - background (batch)

- Process permanently in a given queue.

Multilevel Queue Scheduling (1/3)

- Ready queue is partitioned into separate queues, e.g.:
  - foreground (interactive)
  - background (batch)

- Process permanently in a given queue.

- Each queue has its own scheduling algorithm:
  - foreground: RR
  - background: FCFS

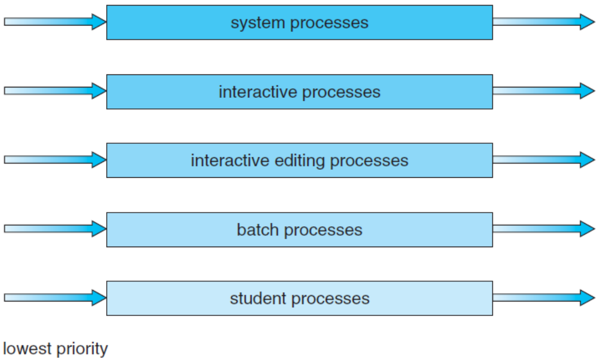
Multilevel Queue Scheduling (2/3)

- Scheduling must be done between the queues:
  - Fixed priority scheduling, i.e., serve all from foreground then from background: possibility of starvation.

Multilevel Queue Scheduling (2/3)

- Scheduling must be done between the queues:
  - Fixed priority scheduling, i.e., serve all from foreground then from background: possibility of starvation.
  - 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.

Multilevel Queue Scheduling (3/3)



Multilevel Feedback Queue Scheduling

Multilevel Feedback Queue

- A process can move between the various queues, e.g.,
  - Aging can be implemented this way
  - If a process uses too much CPU time, it will be moved to a lower-priority queue.

Multilevel Feedback Queue

- A process can move between the various queues, e.g.,
  - Aging can be implemented this way
  - If a process uses too much CPU time, it will be moved to a lower-priority queue.

- Multilevel-feedback-queue scheduler defined by the following parameters:

Multilevel Feedback Queue

- A process can move between the various queues, e.g.,
  - Aging can be implemented this way
  - If a process uses too much CPU time, it will be moved to a lower-priority queue.

- Multilevel-feedback-queue scheduler defined by the following parameters:
  - Number of queues.

Multilevel Feedback Queue

- A process can move between the various queues, e.g.,
  - Aging can be implemented this way
  - If a process uses too much CPU time, it will be moved to a lower-priority queue.

- Multilevel-feedback-queue scheduler defined by the following parameters:
  - Number of queues.
  - Scheduling algorithms for each queue.

Multilevel Feedback Queue

- A process can move between the various queues, e.g.,
  - Aging can be implemented this way
  - If a process uses too much CPU time, it will be moved to a lower-priority queue.

- Multilevel-feedback-queue scheduler defined by the following parameters:
  - Number of queues.
  - Scheduling algorithms for each queue.
  - When to upgrade/demote a process.

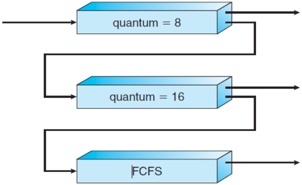
Multilevel Feedback Queue

- A process can move between the various queues, e.g.,
  - Aging can be implemented this way
  - If a process uses too much CPU time, it will be moved to a lower-priority queue.

- Multilevel-feedback-queue scheduler defined by the following parameters:
  - Number of queues.
  - Scheduling algorithms for each queue.
  - When to upgrade/demote a process.
  - Which queue a process will enter when that process needs service.

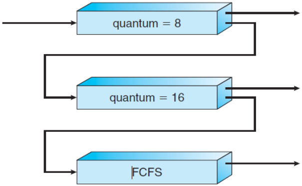
Multilevel Feedback Queue - Example

- For example, three queues:
  - Q_0: RR with time quantum 8 milliseconds
  - Q_1: RR time quantum 16 milliseconds
  - Q_2: FCFS



Multilevel Feedback Queue - Example

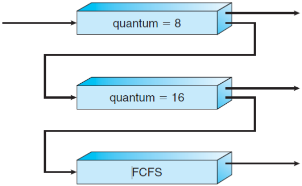
- For example, three queues:
  - Q_0: RR with time quantum 8 milliseconds
  - Q_1: RR time quantum 16 milliseconds
  - Q_2: FCFS



- A new job enters queue Q_0 which is served FCFS:
  - When it gains CPU, job receives 8 milliseconds.
  - If it does not finish in 8 milliseconds, job is moved to queue Q_1.

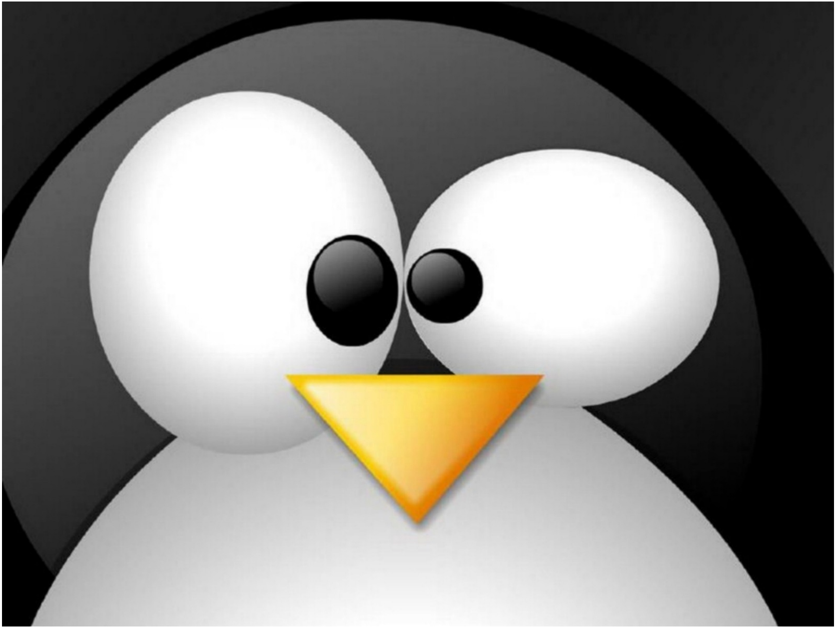
Multilevel Feedback Queue - Example

- For example, three queues:
  - Q_0: RR with time quantum 8 milliseconds
  - Q_1: RR time quantum 16 milliseconds
  - Q_2: FCFS



- A new job enters queue Q_0 which is served FCFS:
  - When it gains CPU, job receives 8 milliseconds.
  - If it does not finish in 8 milliseconds, job is moved to queue Q_1.

- At Q_1 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.



Linux Scheduling Through Version 2.5

- Prior to kernel version 2.5, ran variation of standard UNIX scheduling algorithm.

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.

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.
- Ran in constant time regardless of the number of tasks in the system.

Linux Scheduling Through Version 2.5

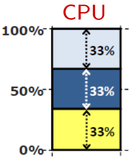
- Prior to kernel version 2.5, ran variation of standard UNIX scheduling algorithm.
- Version 2.5 moved to constant order O(1) scheduling time.
- Ran in constant time regardless of the number of tasks in the system.
- Preemptive, priority based

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.
- Ran in constant time regardless of the number of tasks in the system.
- Preemptive, priority based
- Worked well, but poor response times for interactive processes.

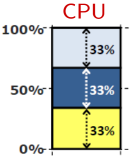
Linux Scheduling in Version 2.6.23+ (1/3)

- Completely Fair Scheduler (CFS)
- n users want to share a resource, e.g., CPU.
  - Solution: allocate each $\frac{1}{n}$ of the shared resource.

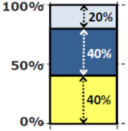


Linux Scheduling in Version 2.6.23+ (1/3)

- Completely Fair Scheduler (CFS)



- n users want to share a resource, e.g., CPU.
  - Solution: allocate each $\frac{1}{n}$ of the shared resource.

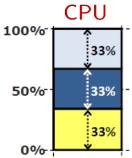


- Generalized by max-min fairness.
  - Handles if a user wants less than its fair share.
  - E.g., user 1 wants no more than 20%.

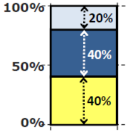
Linux Scheduling in Version 2.6.23+ (1/3)

- Completely Fair Scheduler (CFS)

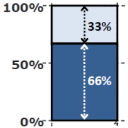
- n users want to share a resource, e.g., CPU.
  - Solution: allocate each $\frac{1}{n}$ of the shared resource.



- Generalized by max-min fairness.
  - Handles if a user wants less than its fair share.
  - E.g., user 1 wants no more than 20%.

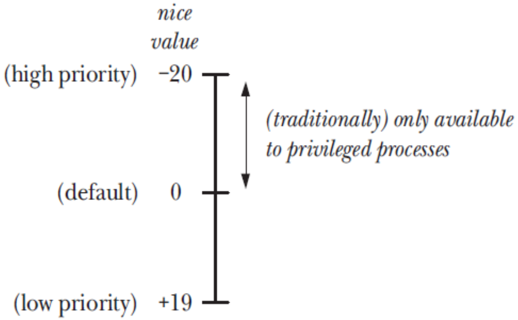


- Generalized by weighted max-min fairness.
  - Give weights to users according to importance.
  - E.g., user 1 gets weight 1, user 2 weight 2.



Linux Scheduling in Version 2.6.23+ (2/3)

- Quantum calculated based on nice value from -20 to +19.



Linux Scheduling in Version 2.6.23+ (3/3)

- To run the next task: the scheduler selects the highest-priority task belonging to the highest-priority scheduling class.

Linux Scheduling in Version 2.6.23+ (3/3)

- To run the next task: the scheduler selects the highest-priority task belonging to the highest-priority scheduling class.

- Standard Linux kernels implement two scheduling classes:
  1. A default scheduling class using the CFS scheduling algorithm.
  2. A real-time scheduling class.

Modifying the Nice Value

- `nice()` increments a process's nice value by `inc` and returns the newly updated value.
- Only processes owned by root may provide a negative value for inc.

```
#include <unistd.h>

int nice(int inc);
```

Retrieving and Modifying Priorities

- The getpriority() and setpriority() system calls allow a process to retrieve and change its own nice value or that of another process.

```
#include <sys/resource.h>

int getpriority(int which, id_t who);

int setpriority(int which, id_t who, int priority);
```

Example

- Returns the current process's priority.

```
int ret;

ret = getpriority(PRIO_PROCESS, 0);
printf("nice value is %d\n", ret);
```

- Sets the priority of all processes in the current process group to 10.

```
int ret;

ret = setpriority(PRIO_PGRP, 0, 10);
if (ret == -1)
  perror("setpriority");
```

Summary

Summary

- CPU scheduling

Summary

- CPU scheduling
- Scheduling criteria: cpu utilization, throughput, turnaround time, waiting time, response time

Summary

- CPU scheduling
- Scheduling criteria: cpu utilization, throughput, turnaround time, waiting time, response time
- Scheduling algorithms
  - FCFS
  - SJF
  - Priority
  - RR
  - Multilevel
  - Multilevel feedback

Questions?

Acknowledgements

Some slides were derived from Avi Silberschatz slides.