

Processor Scheduling

2017-18 Comp3230A

Contents

- Basic scheduling terms & definitions
- Scheduling algorithms
 - First In First Out
 - Shortest Job First
 - Highest-Response-Ratio-Next
 - Shortest Time-to-Completion First
 - Round Robin
 - Multi-level Feedback Queue
 - Fair Share Scheduling (Proportional Share)
- Case study – Scheduling in Linux

Related Learning Outcomes

- ILO 2a - explain how OS manages processes/threads and **discuss** the mechanisms and **policies** in **efficiently sharing** of CPU resources.
- ILO 3 [Performance] - **analyze and evaluate** the algorithms of . . . and **explain** the major performance issues . . .

Readings & Reference

- Required Readings

- Chapter 7, Scheduling: Introduction
 - <http://pages.cs.wisc.edu/~remzi/OSTEP/cpu-sched.pdf>
- Chapter 8, Scheduling: The Multi-Level Feedback Queue
 - <http://pages.cs.wisc.edu/~remzi/OSTEP/cpu-sched-mlfq.pdf>

- References

- Chapter 8 of Operating Systems, 3rd edition by Deitel et. Al
- Chapter 9, Scheduling: Proportional Share
 - <http://pages.cs.wisc.edu/~remzi/OSTEP/cpu-sched-lottery.pdf>

Processor Scheduling

- What is processor scheduling?
 - Is the activity of selecting the next process/thread to be serviced by the processor(s)
- Scheduling policy used by the OS would influence
 - Quality of service provided to users
 - A good scheduler can make a big difference in perceived performance and user satisfaction
 - Effective use of resources
 - Process switching is expensive; scheduling decision has an impact on the efficient use of CPU
 - System performance
 - We would like to maximize the number of processes that completed per unit time

Scheduling Levels

- High-level / Job / Long-term scheduling
 - Deals with creating a new process
 - Controls number of processes in system at one time
i.e., degree of multiprogramming
- Intermediate-level / Medium-term scheduling
 - Determines which processes shall be allowed to compete for processors
 - Deals with swapping processes in/out
 - Responds to fluctuations in system load
- Low-level / Short-term scheduling
 - What process should we run next?
 - Assigns processors to processes

Scheduling Terms & Concepts

- CPU-bound (compute-bound) process
 - When running, tends to use all the processor time that allocated to it
- I/O-bound process
 - When running, tends to use the processor only briefly before generating I/O request and relinquishes the processor
- **Turnaround time** - amount of time to execute a particular process (***from submission to complete servicing***)
 - $T_{\text{turnaround}} = T_{\text{completion}} - T_{\text{arrival}}$
 - Objective: minimize turnaround time
- Waiting time - amount of time a process was ***waiting in the ready queue***

Scheduling Terms & Concepts (2)

- Response time - amount of time it takes from when a process (job) is submitted to the first time it is scheduled
 - An important performance metric of interactive tasks
 - $T_{\text{response}} = T_{\text{firstrun}} - T_{\text{arrival}}$
 - Objective: minimize response time
- Throughput - average # of processes (jobs) that complete their execution per time unit
 - Objective: maximize throughput
- Fairness - all similar processes (jobs) are treated the same, and even if processes are in different priority classes, no process should suffer indefinite postponement (starvation) due to scheduling

When to Schedule

- When a new process is created
- When a process exits
- When a process blocks for I/O or other event
- When a process invokes **system call**
- When an Interrupt occurs
 - I/O interrupt: a process blocked waiting for the I/O now be ready
 - Clock interrupt: a periodical signal to invoke the scheduler

Workload Assumptions

- Here are the assumptions about processes running in our evaluation system
 - All processes (jobs) **only use** the CPU, i.e. no I/O
 - Although not realistic, *jobs with I/O can be treated with each CPU burst (separate by I/O) as an independent job*
 - The duration of runtime (CPU burst) of each process (job) is **known beforehand**

Algorithm Evaluation

- Deterministic modeling
 - Given a **predetermined workload** and evaluate the performance of each algorithm
- Example for performance analysis of different algorithms

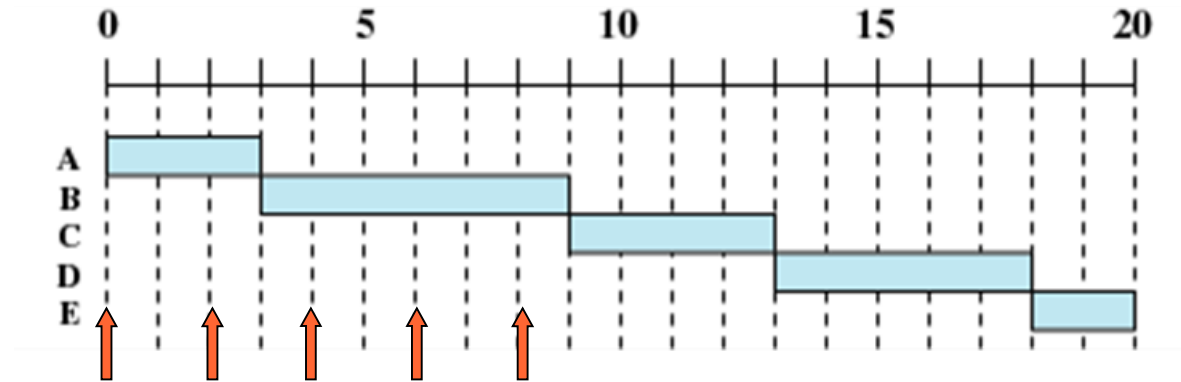
Assume that
new processes
**always arrive
just before** the
clock tick

Process	Arrival Time	CPU burst
A	0	3
B	2	6
C	4	4
D	6	5
E	8	2

First-In-First-Out (FIFO) Scheduling

- Also known as FCFS (First Come First Serve)
- Processes dispatched according to arrival time
 - Order in a list (queue) according to the arrival time
- Is a **non-preemptive** scheme
 - Once given the CPU, run until completion or voluntary release of CPU
- Advantages
 - **Fair** - all processes are treated equally
 - Easy to implement

FIFO Scheduling



Time	Queue	CPU
0	A	
0		A
2	B	A
3		B
4	C	B
6	C ← D	B
8	C ← D ← E	B
9	D ← E	C
13	E	D
18		E

- Turnaround time for
 - $A = 3 - 0 = 3$
 - $B = 9 - 2 = 7$
 - $C = 13 - 4 = 9$
 - $D = 18 - 6 = 12$
 - $E = 20 - 8 = 12$
- Average turnaround time: $(3+7+9+12+12)/5 = 8.6$

FIFO Scheduling

- Disadvantages

- Short processes may have to wait relatively longer time when they are queued behind a CPU-bound process
- Not good for interactive processes

Response time = Wait time

Response time of

A=0

B=1

C=5

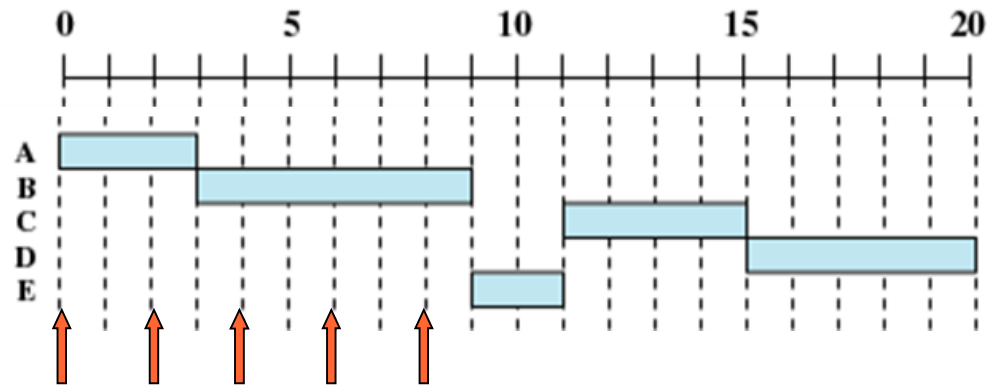
D=7

E=10

Shortest-Job-First (SJF) Scheduling

- Also known as Shortest-Process-First
- A **nonpreemptive** scheme that selects process with shortest “CPU burst” to run next
- Advantage
 - **Lower average wait time** than FIFO
 - Reduces the number of waiting processes
 - Known to be better in terms of average waiting time

SJF Scheduling



Time	Queue	CPU
0		A
2	B	A
3		B
4	C	B
6	C ← D	B
8	E ← C ← D	B
9	C ← D	E
11	D	C
15		D

- Average turnaround time: $(3+7+11+14+3)/5 = 7.6$

Response time of

A=0

B=1

C=7

D=9

E=1

Shortest-Job-First (SJF) Scheduling

- Disadvantages
 - Because of **non-preemptive nature**, newly short/interactive jobs may be forced to wait for a running CPU-bound job
 - may results in slow response times
 - Possibility of **starvation** for longer processes
 - If there always have short interactive processes, longer processes have to wait longer
 - Potentially large variance in wait times
- These policies are not good for interactive jobs which demand to have good response time

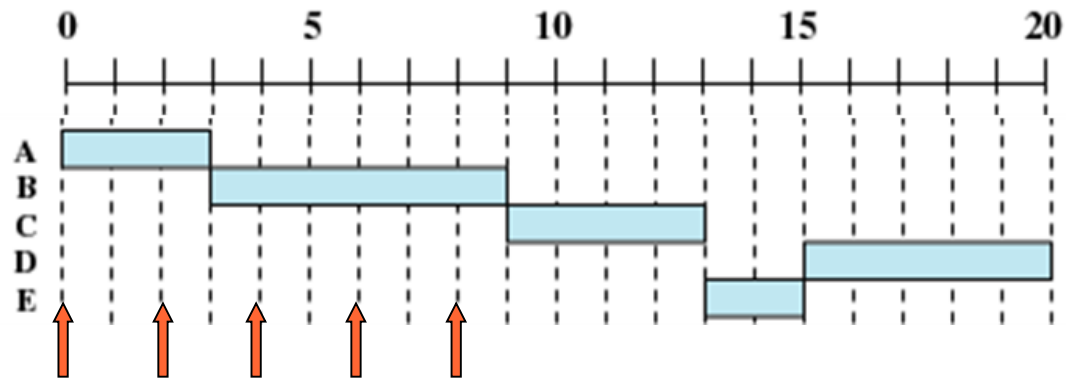
Highest-Response-Ratio-Next (HRRN) Scheduling

- A **nonpreemptive** scheme; is an improvement of SJF scheduling
- We take into consideration of how long process has been waiting
- Calculate the priority based on **predicted service time** as well as how long has this process been **waiting**

$$\text{priority} = \frac{\text{time waiting} + \text{service time}}{\text{service time}}$$

- Shorter processes are favored for scheduling
- **Aging effect**; prevents indefinite postponement

HRRN



Time	Queue	CPU
0		A
2	B	A
3		B
4	C	B
6	C ← D	B
8	C ← D ← E	B
9	D ← E	C
13	D	E
15		D

- Average turnaround time: $(3+7+9+14+7)/5 = 8$

Response time of

A=0

B=1

C=5

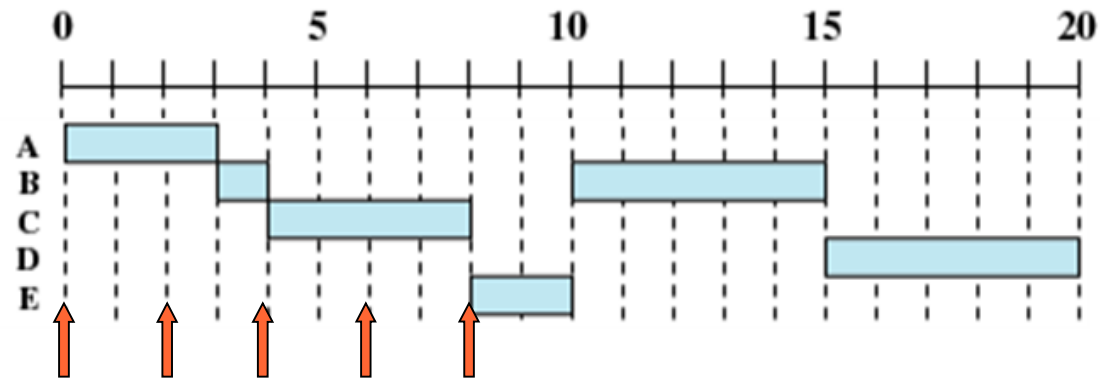
D=9

E=5

Shortest Time-to-Completion First (STCF) Scheduling

- **Preemptive** version of SJF
 - Running process may be **interrupted and switched** to Ready state by the scheduler
- **Arriving processes** will trigger scheduler to check against the **remaining running time** of existing and new processes, and schedules the process with the **shortest run-time-to-completion**
 - Remaining = CPU burst – elapsed **service** time
 - Must maintain information about the elapsed **service** time

STCF Scheduling



Time	Queue	CPU
0		A
2	B	A (1)
3		B
4	B	C
6	B ← D	C (2)
8	B ← D	E
10	D	B
15		D

- Average turnaround time: $(3+13+4+14+2)/5 = 7.2$

Response time of

A=0

B=1

C=0

D=9

E=0

Waiting time of

A=0

B=7

C=0

D=9

E=0

STCF Scheduling

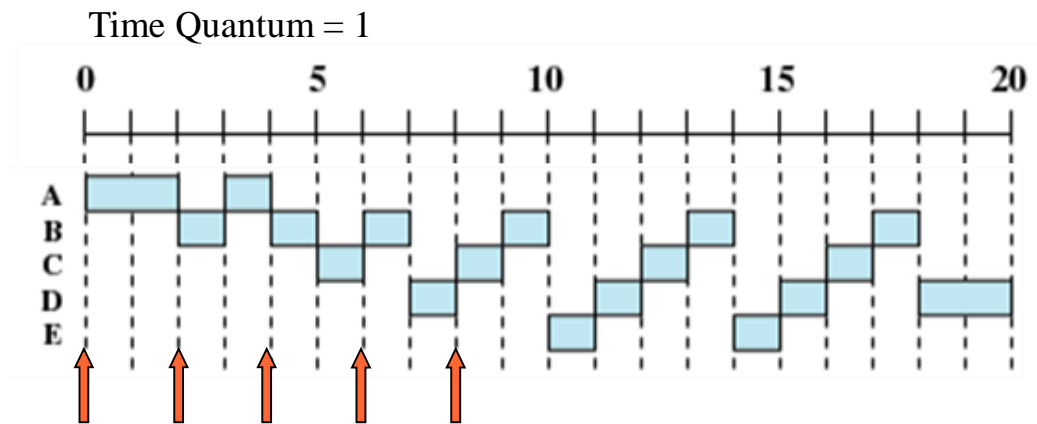
- Disadvantages

- Very large variance of waiting times: long processes may wait even longer than under SJF; because with SJF, long process won't be preempted as being scheduled
- Not always optimal
 - Short incoming process can preempt a running process that is near completion
 - Would experience more context switches and affect overall performance

Round-Robin (RR) Scheduling

- Keep processes in ready queue in FIFO
 - New processes are added to the tail of the ready queue
- Processes run only for a limited amount of time called a **time slice** or a **time quantum**
- **Preemption**
 - Upon clock interrupt, if currently running process has its quantum expires, place it to the tail of ready queue; next ready process is selected and dispatched
- Advantages
 - Fair
 - Good for interactive processes

RR Scheduling



- Average turnaround time:
 $(4+16+13+14+7)/5 = 10.8$

Response time of
 A=0
 B=0
 C=1
 D=1
 E=2

Waiting time of
 A=1
 B=10
 C=9
 D=9
 E=5

Time	Queue	CPU
0		A
1		A
2	A	B
3	B	A
4	C	B
5	B	C
6	D ← C	B
7	C ← B	D
8	B ← E ← D	C
9	E ← D ← C	B
10	D ← C ← B	E
11	C ← B ← E	D
12	B ← E ← D	C
13	E ← D ← C	B
14	D ← C ← B	E
15	C ← B	D
16	B ← D	C
17	D	B
18		D

B ← C
 B ← D

RR Scheduling

- Disadvantage
 - The RR policy does not go well in terms of turnaround time
- Quantum size
 - Determines response time to interactive requests
 - Very large quantum size
 - Processes run for long periods
 - Degenerates to FIFO
 - Very small quantum size
 - System spends more time context switching than running processes
 - Middle-ground
 - Long enough for interactive processes to issue I/O request
 - Long processes still get majority of processor time

The Crux

- How can we design a scheduler that both
 - **Minimizes response time** for interactive jobs
 - Like RR, make the system be responsive to interactive users
 - **Minimizes turnaround time** **without a priori knowledge** of CPU burst time
 - Like SJF or STCF, gives preference to shorter processes

Priority

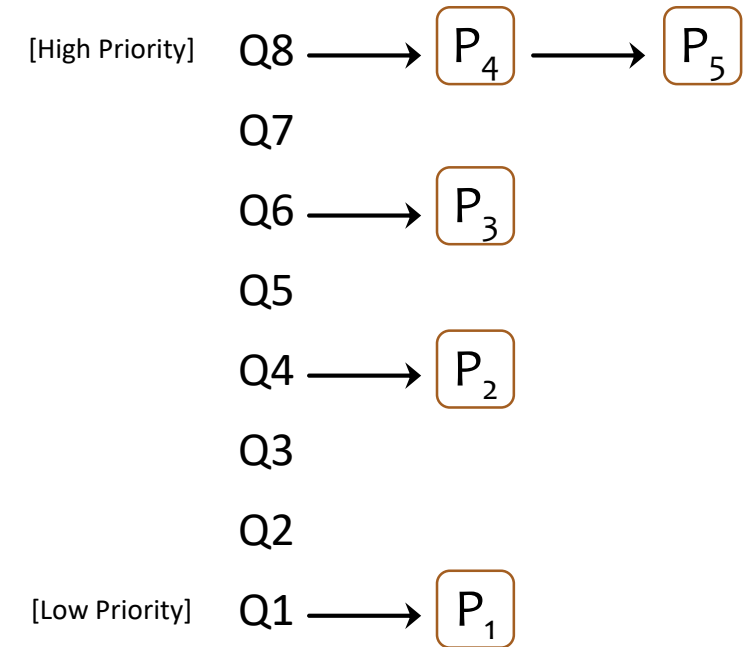
- A priority number is assigned to each process
 - Scheduler chooses a process of higher priority over one with lower priority
 - i.e., selection of tasks other than just using the arrival order
- Static priorities - priority does not change during process's lifetime
 - Adv: easy to implement and low overhead
 - Disadv: not responsive to changes in environment
 - Disadv: lower-priority processes may suffer starvation
- Dynamic priorities
 - Adv: responsive to change
 - Disadv: incur more overhead than static priorities as scheduler needs to determine the priority of all processes when making scheduling decision

Multilevel Feedback Queues (MLFQ) Scheduling

- Multilevel Feedback Queues
 - The system consists of a number of ready queues, each has a different priority level
 - Scheduler always select a process that appears in the highest priority queue to run first
 - Processes in lower-priority queues will run only when higher-priority queues are empty
 - Within the same priority queue, scheduler selects process using round-robin scheduling
- Principle of assigning priority
 - Using dynamic priority – process's priority is changing
 - MLFQ varies the priority of a process based on applications' runtime characteristic
 - MLFQ tries to learn about processes as they run, and uses the history of a process to predict its future behavior

MLFQ – Assigning Priority

- New submitted processes enter the highest-priority queue
- If a process **uses up** its quantum, it is preempted and positioned at the **end of the next lower level** queue
 - Long processes repeatedly descend into lower priority levels
- If a process relinquishes the CPU before quantum expires, it stays at the same priority level
 - **We don't want to penalize interactive job and keep them at the same priority level**

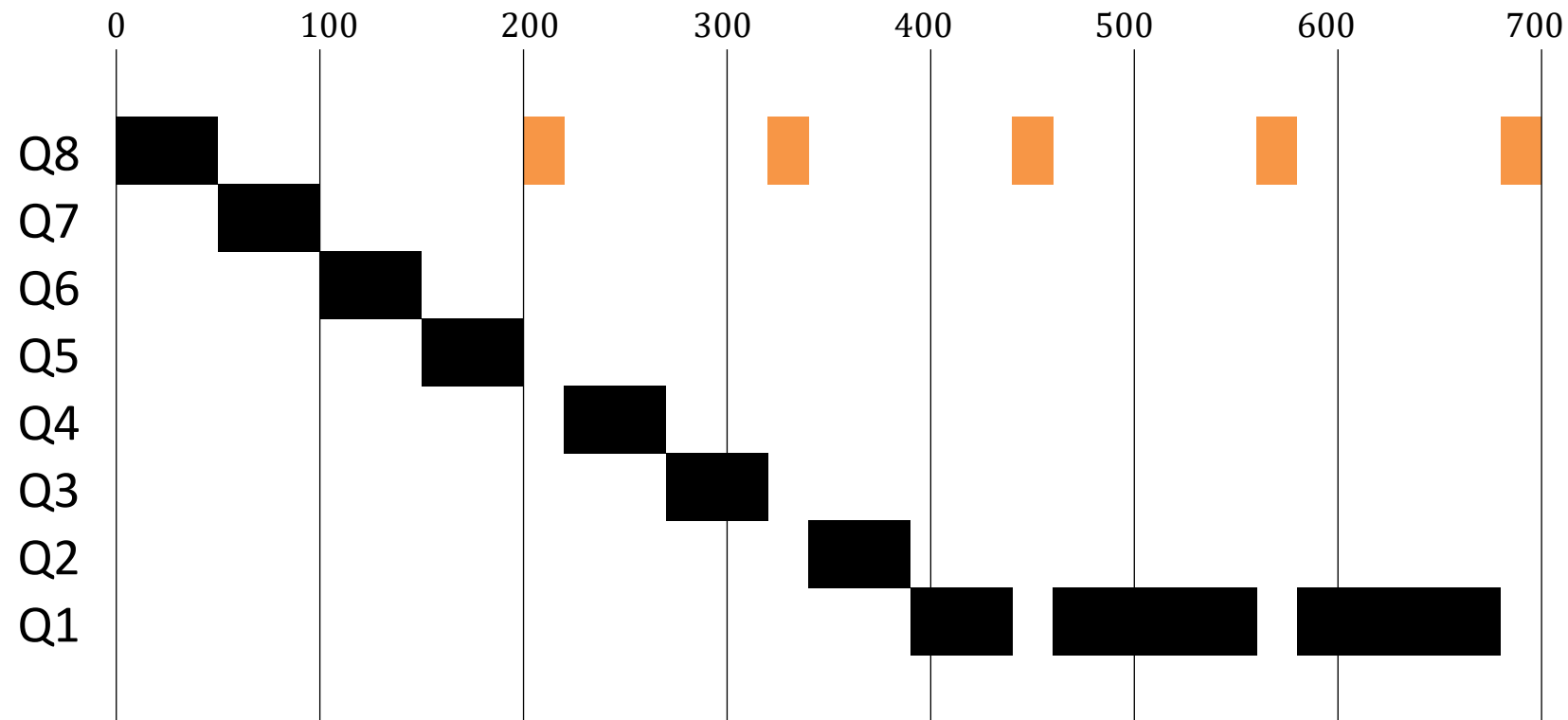


MLFQ – in action

P_0 is a long process and arrives at $T = 0$

P_1 is an interactive process, which arrives at $T = 200$ and uses the CPU for 20 time units and blocks for I/O for 100 time units

Assume the time quantum at each level is 50 time units



MLFQ - Deficiency

- Process in lower queue can suffer indefinite postponement (**starvation**)
 - If too many interactive processes, they will eat up CPU time and long processes will be starved
 - A serious issue for priority-based scheduling scheme (especially with static priority)
- **Misbehaved** processes can steal lots of CPU time
 - Process can pretend to be a short job by voluntarily relinquish of CPU before expiration of time quantum
 - Just to trick the scheduler
- Process may **change** its behavior from CPU-bound to I/O-bound in its lifetime
 - Unfortunately, a CPU-bound process has descended to the lowest queue; cannot be treated as interactive process

MLFQ - Adjustment

- Starvation
 - Periodically **boost** the priority of all processes in system
 - Example: Every time period S , move all processes to the topmost queue
 - Long processes will **get the chance** to run AND
 - A CPU-bound process that **turns** interactive can be treated correctly
- **Different time quantum length for different queues**
 - The scheduler increases a process's quantum size as the process moves to each lower-level queue
 - Example: weight = 2^{k-i} (simply double the quantum, where i is current level and k is the no. of priority level)
 - High-priority queues are given short time slices
- Unfair usage by misbehaved process
 - Memorize a process CPU time at each level
 - When a process has **used its time allotment at a given level**, it is demoted to the next priority queue

Multilevel Feedback Queues

- A good example of an adaptive mechanism
 - Responds to changing behavior of the environment
 - Move processes to different queues as they alternate between interactive and batch behavior
 - Incur more overhead but system has increased its sensitivity

Fair Share Scheduling

- Previous scheduling policies apply to individual processes
 - Consider an application may spawn many processes which gives this application more CPU attention than others. Would that be fair?
 - The same issue happens when an user starts many application processes and gets more CPU share
 - How about a group of students running many processes and affects others using their share
- Fair Share Scheduling supports scheduling decisions based on process groups
 - Users (or groups) are assigned a weighting that indicates their share of system resource as a fraction of the total resource capacity
 - FSS tries to monitor resource usage to give fewer resources to users (or groups) who have had more than their fair share and more to those who have had less than their fair share

Lottery Scheduling

- Implementing the fair sharing concept by using **random** allocation
 - On long run, the resource consumption rates of users (or groups) are proportional to the relative shares that they are allocated
- Lottery tickets are distributed to all processes (or users or groups)
 - On the basis of their fair share of CPU time
 - A process/user/group is given 10 tickets out of 100 if its fair share is 10 percent
- When scheduling
 - **A lottery ticket is chosen at RANDOM**
 - **The process holding the winning ticket is allocated the CPU**
- Why is fair?
 - More important processes can be **given extra tickets** to increase their odds of winning
 - The more tickets a process has, the higher the chance
 - On long run, a process holding a fraction of f of the tickets will get about a fraction of f of the CPU resource

ConceptTest

Not for examination

Case Study

Linux Scheduler

Linux Scheduling

- A preemptive, priority-based algorithm
- There are two separate priority classes:
 - Real-time and non-real-time (normal)
- Real-time priority: 1 (low) to 99 (high)
 - SCHED_FIFO: First-in-first-out real-time process
 - Run until its exits, sleeps, blocks, or is preempted by another newly arrived higher priority process
 - SCHED_RR: Round-robin real-time process
 - Using RR scheme on real-time processes with the same priority
 - Real-time processes can indefinitely postpone other processes, so normal users cannot create a real-time process

Linux Scheduling – Completely Fair Scheduler (CFS)

- For **non**-real-time class from kernel 2.6.23 and after
 - This scheme does not label a process as interactive or CPU-bound; every process looks the same initially
 - The main idea behind the CFS is to **maintain fairness** in providing processor time to tasks
 - **Fairness**
 - Divide the CPU resource fairly by the available runnable processes
 - If currently has nr processes, each gets $1/nr$ units of CPU time
 - Priority is given by the weighting factor
 - 8 processes of 1 unit weighting and 1 process of 2 units of weighting
 - each process has $1/10$ units of CPU time except the higher priority process which receives $2/10$ units

Linux Scheduling – CFS

- With an **ideal**, precise, multitasking CPU, it can run all processes at the same time (i.e., truly concurrently) by giving each process its share of processor power
 - In the above example, each of the 8 processes gets 10% of the physical processor power and the higher priority process gets 20% of the power; they all run in parallel
- Implementation
 - On a real-world processor (core), only one process can be allocated to a processor at a particular time; thus, the other processes cannot get their share
 - CFS maintains the amount of time provided to a given task in what's called the virtual runtime
 - When a process is running, CFS keeps track of a process runtime by adding its execution duration (normalized to the total runnable processes and their priority) to its virtual runtime
 - The higher the number, the more CPU time it has consumed

Linux Scheduling – CFS

- Scheduling decision
 - CFS selects the process with the smallest virtual runtime value in the system
 - Use a time-ordered Red-Black tree to order runnable (ready) tasks
 - Red-Black Tree - Balance binary tree of $O(\log_2 n)$ time complexity of operations
 - By order the tree nodes by their virtual runtime values, CFS can quickly locate the process with the “gravest CPU need”, which is the leftmost node in the tree

Summary

- Scheduling policies (algorithms)
 - Decide when and for how long each process runs
- Make choices about
 - Preemptibility and time quantum – These affect the responsiveness
 - Priority – how to apply priority to processes and how to enforce priority
 - Turnaround time – the smaller the better
 - Fairness – No process should suffer starvation
- MLFQ is a good example of a system that learns from the past to predict the future
- FSS monitors resource usage and try to provide a guarantee that each process obtains a certain percentage of CPU time