

CSCI3150 Introduction to Operating Systems

Lecture 9: CPU Scheduling

Hong Xu

<https://github.com/henryhxu/CSCI3150>

Roadmap: CPU Scheduling

- ▣ Introduction
- ▣ Multi-Level Feedback Queue
- ▣ Lottery Scheduling

Introduction

Scheduling: Introduction

- Workload assumptions:

1. Each job runs for the **same amount of time**.
2. All jobs **arrive** at the same time.
3. All jobs only use the **CPU** (i.e., they perform no I/O).
4. The **run-time** of each job is known.

Scheduling Metrics

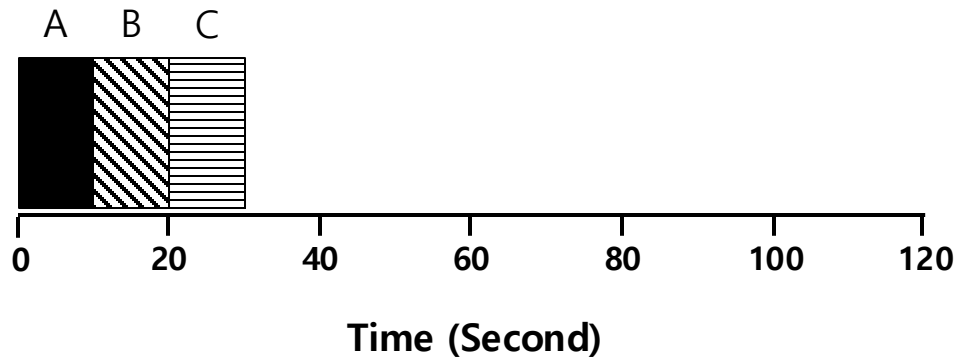
- Performance metric: Turnaround time, or completion time
 - ♦ The time at which **the job completes** minus the time at which **the job arrived** in the system.

$$T_{completion} = T_{done} - T_{arrival}$$

- Job completion time, JCT
- Another metric is fairness.
 - ♦ Performance and fairness are often at odds in scheduling.

First In, First Out (FIFO)

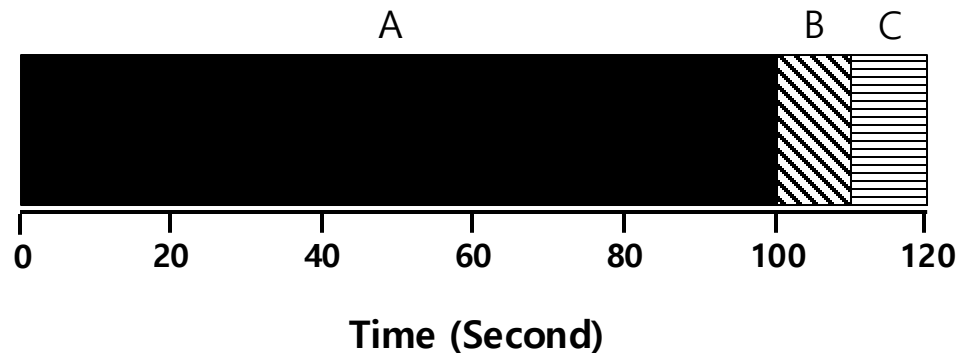
- ❑ First Come, First Served (FCFS)
 - ◆ Very simple and easy to implement
- ❑ Example:
 - ◆ A arrived just before B which arrived just before C.
 - ◆ Each job runs for 10 seconds.



$$\text{Average JCT} = \frac{10 + 20 + 30}{3} = 20 \text{ sec}$$

Why FIFO is not that great? – Head-of-line Blocking

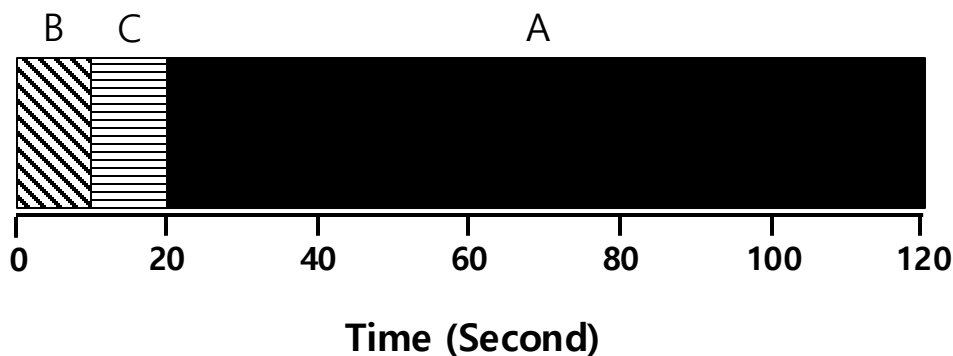
- ▣ Let's relax assumption 1: Each job **no longer** runs for the same amount of time.
- ▣ Example:
 - ◆ A arrived just before B which arrived just before C.
 - ◆ A runs for 100 seconds, B and C run for 10 each.



$$\text{Average JCT} = \frac{100 + 110 + 120}{3} = \mathbf{110 \text{ sec}}$$

Shortest Job First (SJF)

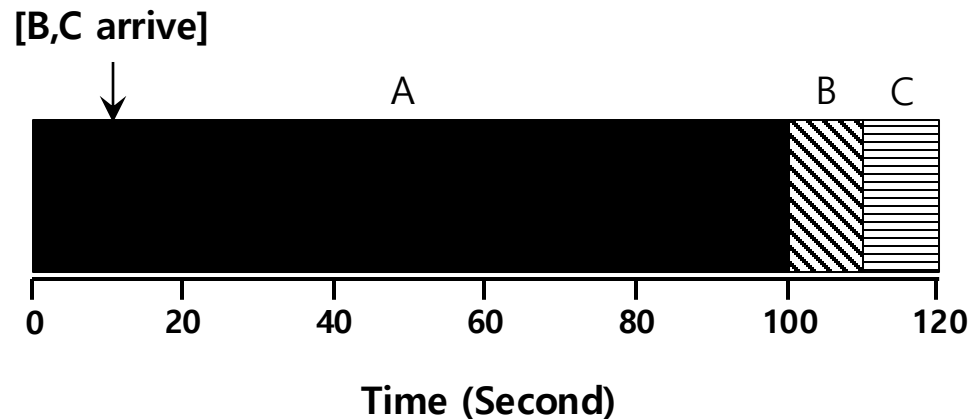
- Run the shortest job first, then the next shortest, and so on
 - ◆ Non-preemptive scheduler
- Same example:



$$\text{Average JCT} = \frac{10 + 20 + 120}{3} = 50 \text{ sec}$$

SJF with Late Arrivals from B and C

- ▣ Let's relax assumption 2: Jobs can arrive at any time.
- ▣ Example:
 - ◆ A arrives at $t=0$ and needs to run for 100 seconds.
 - ◆ B and C arrive at $t=10$ and each needs to run for 10 seconds



$$\text{Average JCT} = \frac{100 + (110 - 10) + (120 - 10)}{3} = 103.33 \text{ sec}$$

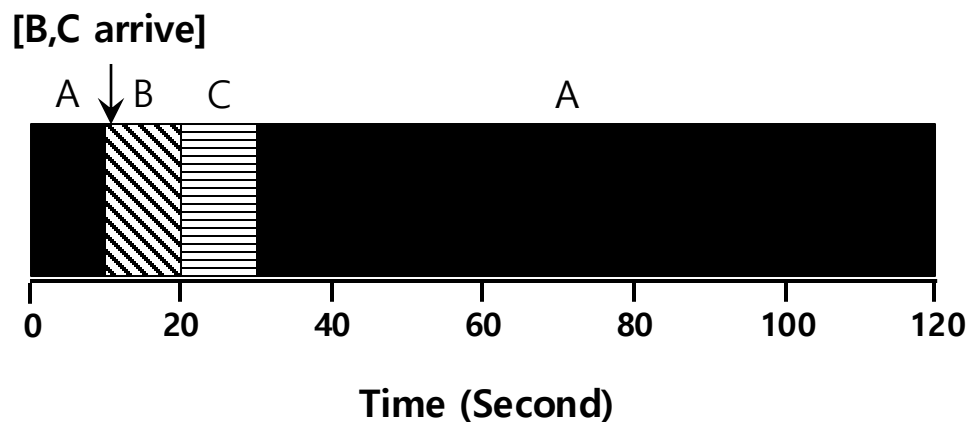
Shortest Remaining Time First (SRTF)

- ▣ Also known as: Shortest Time-to-Completion First
- ▣ Add **preemption** to SJF
- ▣ When a new job enters the system:
 - ◆ Determine the remaining times of all jobs
 - ◆ Schedule the job which has the least time left

Shortest Remaining Time First (SRTF)

□ Example:

- ♦ A arrives at $t=0$ and needs to run for 100 seconds.
- ♦ B and C arrive at $t=10$ and each needs to run for 10 seconds



$$\text{Average JCT} = \frac{(120 - 0) + (20 - 10) + (30 - 10)}{3} = 50 \text{ sec}$$

New scheduling metric: Response time

- ▣ The time from **when the job arrives** to the **first time it is scheduled**.

$$T_{response} = T_{firstrun} - T_{arrival}$$

- ◆ SRTF and related disciplines are not particularly good for response time.

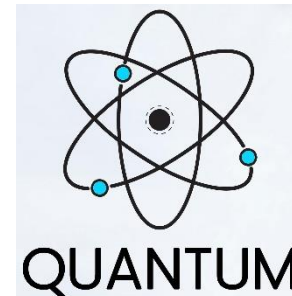
**How can we build a scheduler that is
sensitive to response time?**

Round Robin (RR) Scheduling

▣ Time slice:

- ◆ Run a job for a **time slice** and then switch to the next job in the **run queue** until all jobs are finished.

- Time slice is sometimes also called a scheduling quantum.

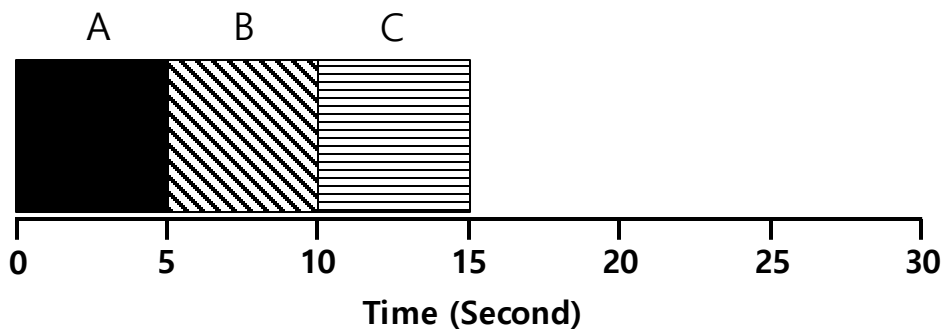


- ◆ The length of a time slice must be *a multiple of* the timer-interrupt period.

**RR is fair, but performs poorly on metrics
such as JCT**

RR Scheduling Example

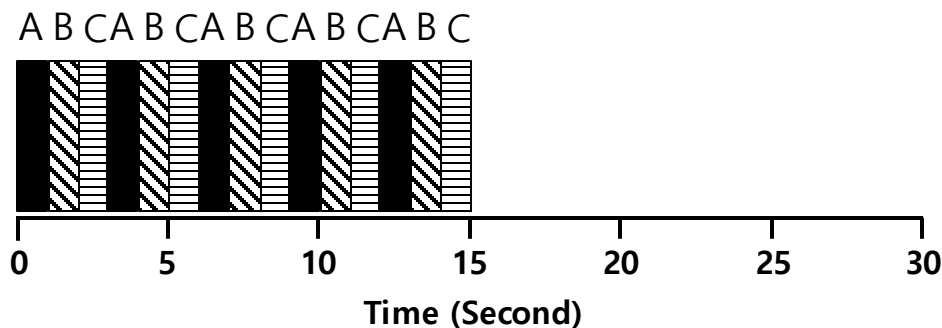
- ▣ A, B and C arrive at the same time.
- ▣ They each wishes to run for 5 seconds.



SJF (Bad for Response Time)

$$T_{average\ response} = \frac{0 + 5 + 10}{3} = 5sec$$

What about JCT?



RR with a time-slice of 1sec (Good for Response Time)

$$T_{average\ response} = \frac{0 + 1 + 2}{3} = 1sec$$

Length of the time slice is critical

- The shorter time slice

- ◆ Better response time
- ◆ But the cost of context switching will dominate overall performance.

- The longer time slice

- ◆ Amortize the cost of switching
- ◆ Worse response time

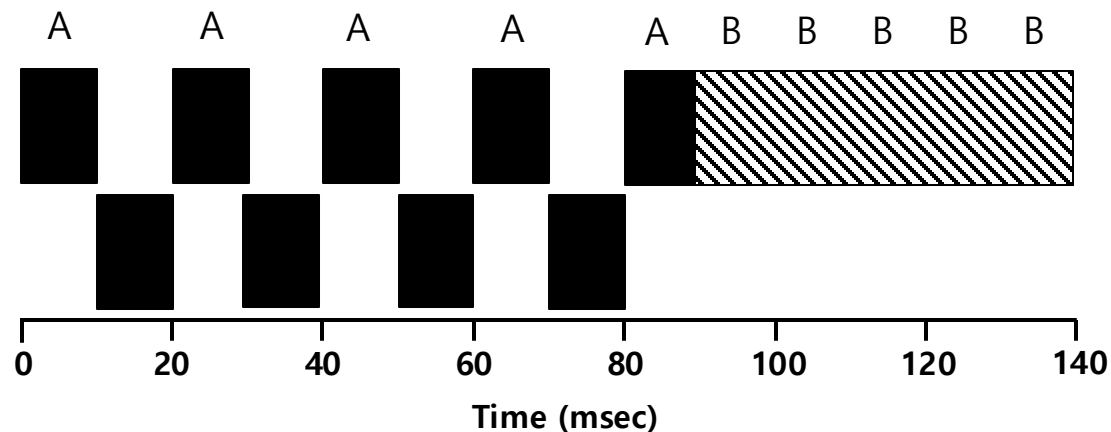


**Length of the time slice presents
a **trade-off** to system design**

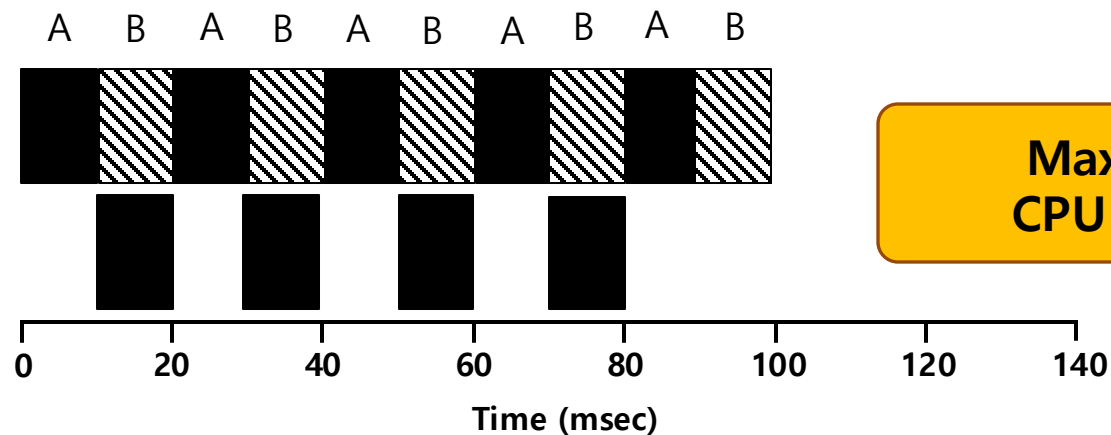
Incorporating I/O

- ▣ Let's relax assumption 3: All programs perform I/O
- ▣ Example:
 - ◆ A and B need 50ms of CPU time each.
 - ◆ A runs for 10ms and then issues an I/O request
 - I/Os takes 10ms each
 - ◆ B simply uses the CPU for 50ms and performs no I/O
 - ◆ The scheduler runs A first, then B

Incorporating I/O (Cont.)



Poor Use of Resources



Overlap Allows Better Use of Resources

Maximize the
CPU utilization

Incorporating I/O (Cont.)

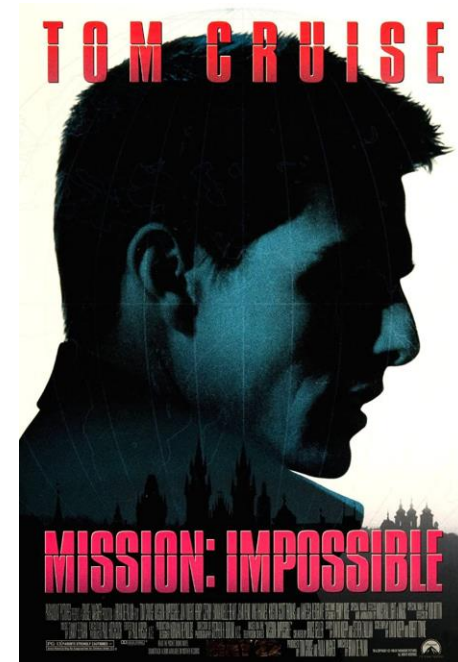
- When a job initiates an I/O request
 - ◆ The job is blocked waiting for I/O completion.
 - ◆ The scheduler should schedule jobs on the CPU.

- When the I/O completes
 - ◆ An interrupt is raised.
 - ◆ The OS moves the process from the wait to the ready state.

Multi-Level Feedback Queue

Motivation

- ❑ In reality, an OS does not know how long a job would run for!
 - ◆ Do you know how long you'll play your game...
- ❑ But we still want to schedule jobs well!
- ❑ Is it possible to still design a good scheduler without any prior knowledge of job's running time?
 - ◆ Optimize **JCT** → Run shorter jobs first
 - ◆ Also **response time**
 - ◆ "huh?"



MLFQ: Key idea

- Upon arrival, each job is **assumed** to be a short, latency-sensitive job
 - ◆ It can be scheduled right away with the highest priority for better response time.
- If the job keeps running
 - ◆ It's obvious that it's not a short job anymore, and more likely to be a long or loong or looong job
 - ◆ Decrease its priority level, so other (shorter) jobs can run first for better JCT

We can be wrong at the beginning, but that's a necessary small price to pay for practicality

MLFQ: Basic Rules

- ▣ MLFQ has a number of **queues**.
 - ◆ Each queue is assigned a different priority level.
- ▣ A job that is ready to run is on exactly one queue.
 - ◆ A job **on a higher queue** is chosen to run.
 - ◆ Use round-robin scheduling among jobs in the same queue

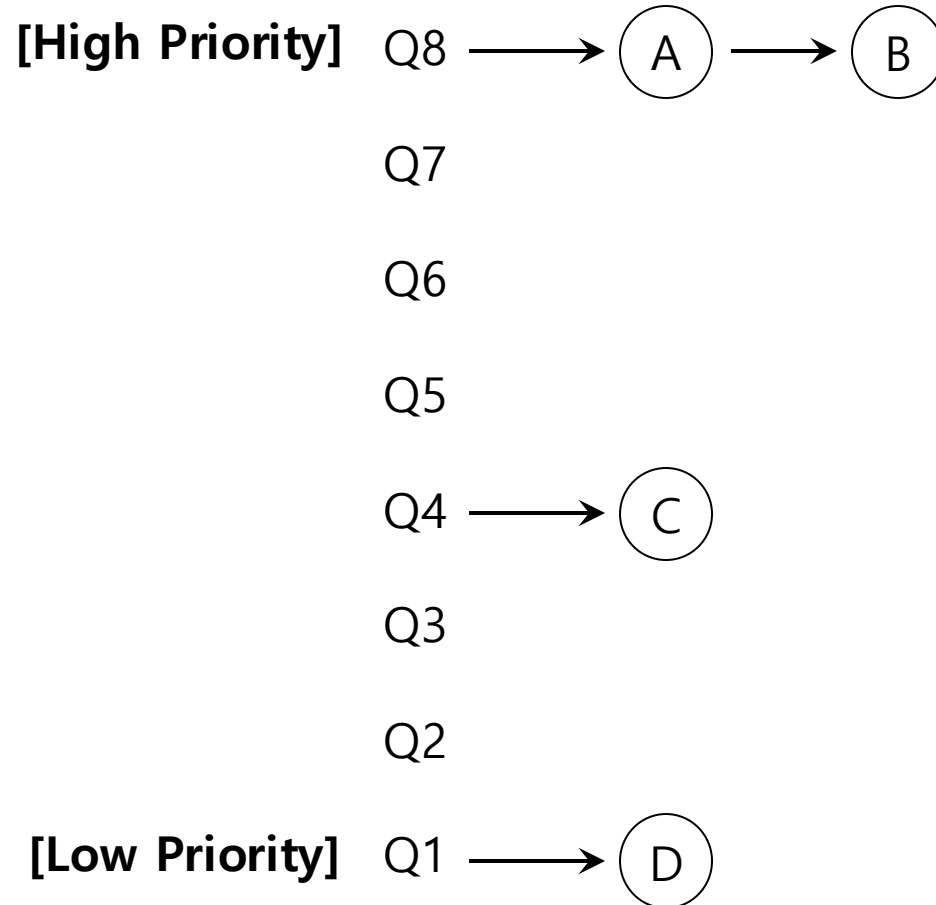
Rule 1: If $\text{Priority}(A) > \text{Priority}(B)$, A runs (B doesn't).

Rule 2: If $\text{Priority}(A) = \text{Priority}(B)$, A & B run in RR.

MLFQ: Basic Rules (Cont.)

- ▣ MLFQ varies the priority of a job based on its observed behavior.
- ▣ Example:
 - ◆ A job repeatedly relinquishes the CPU while waiting IOs → Keep its priority high
 - ◆ A job uses the CPU intensively for long periods of time → Reduce its priority.

MLFQ Example



MLFQ: How to Change Priority

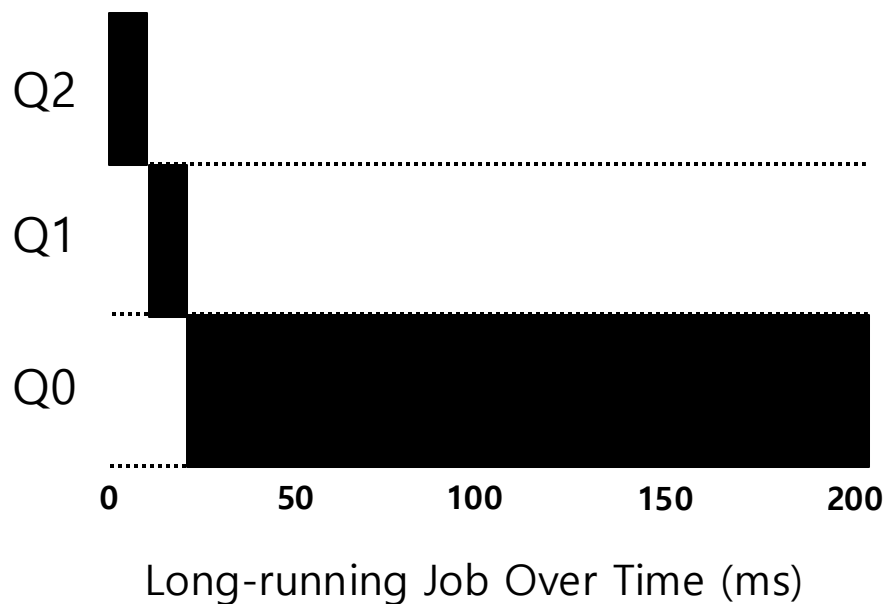
- ▣ MLFQ priority adjustment:

- ◆ **Rule 3:** When a job enters the system, it is placed at the highest priority
- ◆ **Rule 4a:** If a job uses up an entire time slice while running, its priority is reduced (i.e., it moves down on queue).
- ◆ **Rule 4b:** If a job gives up the CPU before the time slice is up, it stays at the same priority level

In this manner, MLFQ approximates SJF

Example 1: A Single Long-Running Job

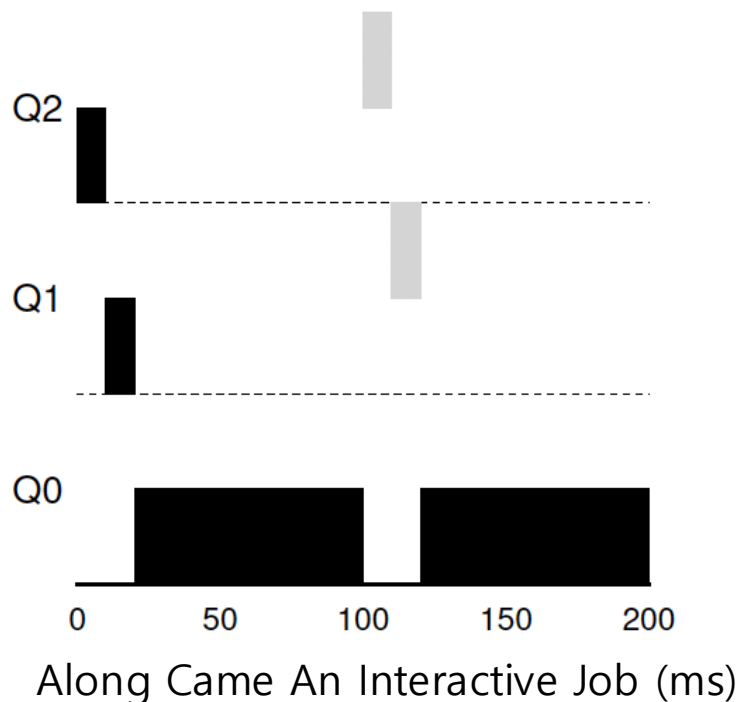
- A three-queue scheduler with 10ms time slices



Example 2: Along Came a Short Job

□ Assumption:

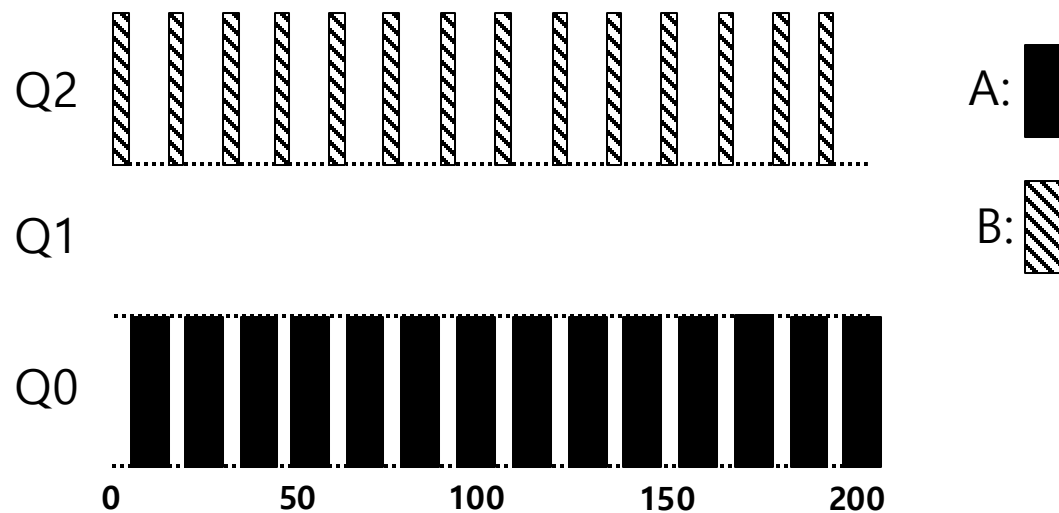
- ♦ **Job A:** A long-running CPU-intensive job
- ♦ **Job B:** A short-running interactive job (20ms runtime)
- ♦ A has been running for some time, and then B arrives at time $T=100$.



Example 3: What About I/O?

Assumption:

- ♦ **Job A:** A long-running CPU-intensive job
- ♦ **Job B:** An interactive job that needs CPU only for 1ms before performing an I/O



A Mixed I/O-intensive and CPU-intensive Workload (msec)

The MLFQ approach keeps an interactive job at the highest priority

Problems with the Basic MLFQ

- ▣ Starvation
 - ◆ If there are “too many” interactive jobs in the system.
 - ◆ Lon-running jobs will never receive any CPU time.

- ▣ Game the scheduler, or “cheat”
 - ◆ After running 99% of a time slice, issue an I/O operation.
 - ◆ The job gain a higher percentage of CPU time.

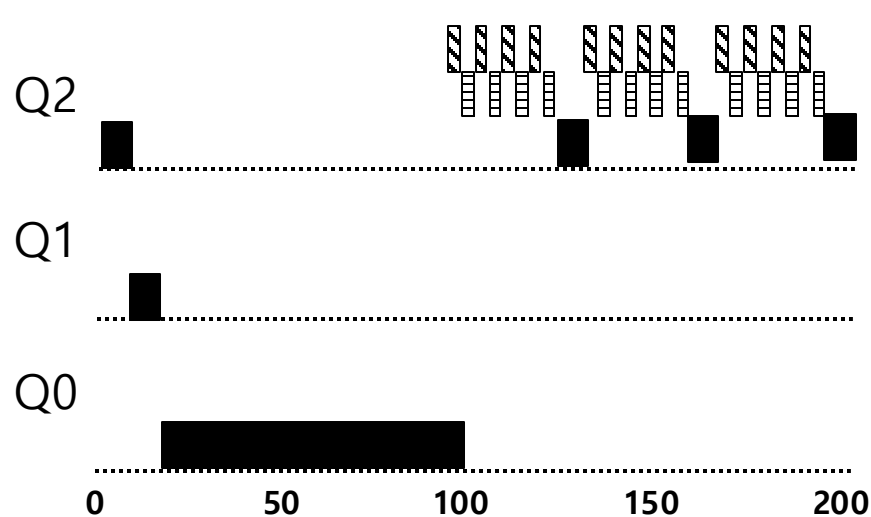
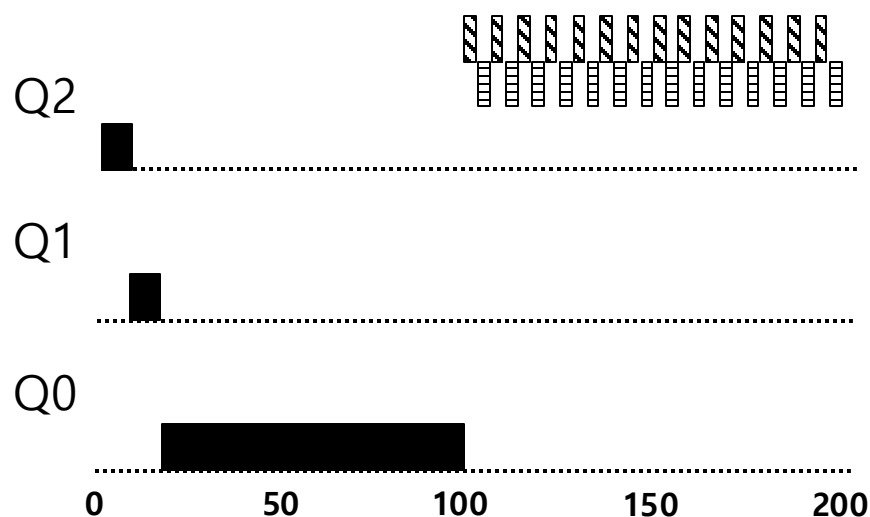
- ▣ A program may change its behavior over time
 - ◆ CPU bound → I/O process

The Priority Boost




- ▣ **Rule 5:** After some time periods S , move all jobs in the system to the top queue.

- ◆ Example:

- A long-running job (A) with two short-running interactive jobs (B, C)

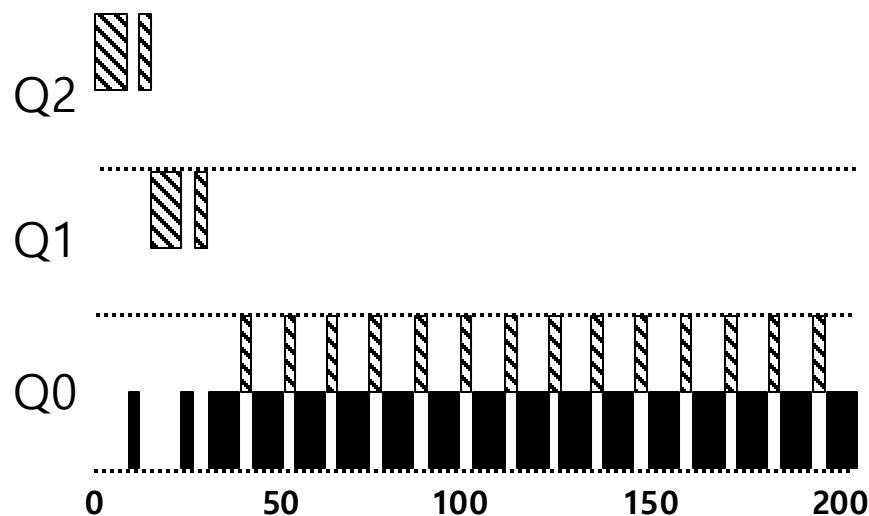
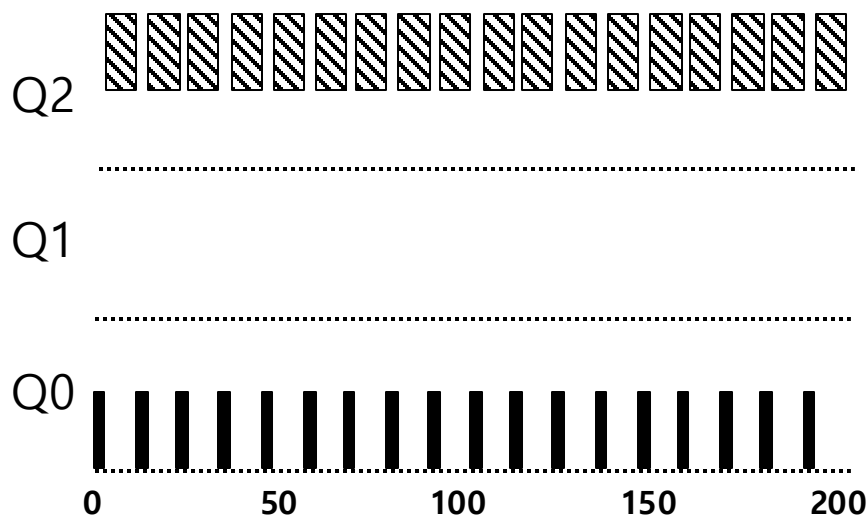


Without(Left) and With(Right) Priority Boost

A:  B:  C: 

Better Accounting

- How to prevent jobs from gaming our scheduler?
- Solution:
 - ◆ **Rule 4** (Rewrite Rules 4a and 4b): Once a job **uses up its time allotment** at a given level (regardless of how many times it has given up the CPU), **its priority is reduced** (i.e., it moves down on queue).

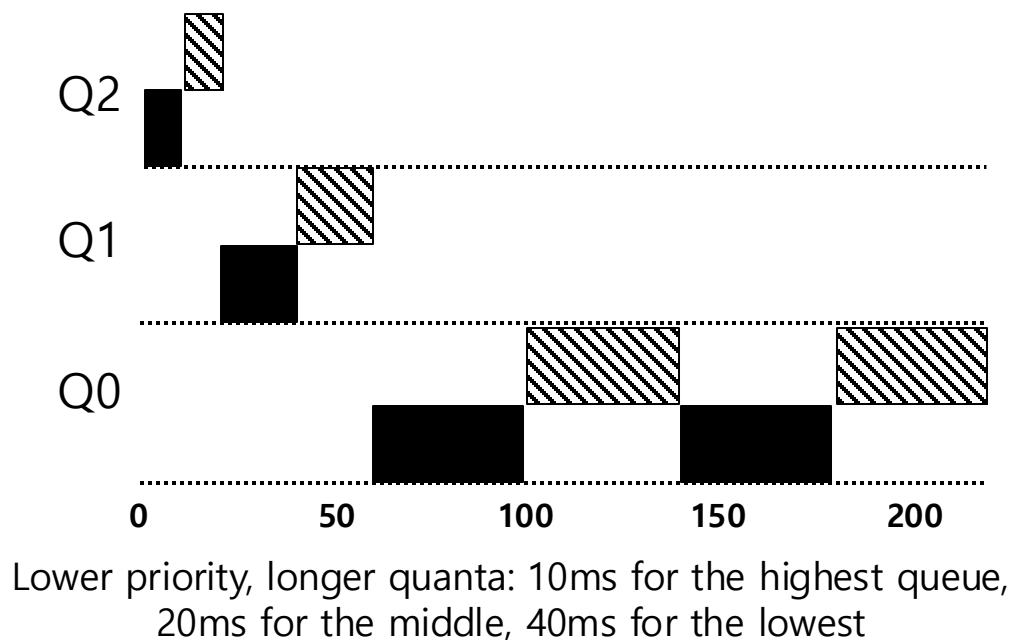


Without(Left) and With(Right) Gaming Tolerance

Tuning MLFQ and Other Issues

Lower Priority, Longer Quanta

- ♦ High-priority queues → Short time slices
 - E.g., 10 or fewer milliseconds
- ♦ Low-priority queue → Long time slices
 - E.g., 100 milliseconds



MLFQ implementation: on Solaris

- For the Time-Sharing scheduling class (TS)
 - ◆ 60 Queues
 - The highest priority: 20ms
 - The lowest priority: A few hundred ms
 - ◆ Priorities boosted around every 1 second or so.



MLFQ: Summary

- ▣ The refined set of MLFQ rules:
 - ◆ **Rule 1:** If $\text{Priority}(A) > \text{Priority}(B)$, A runs (and B doesn't).
 - ◆ **Rule 2:** If $\text{Priority}(A) = \text{Priority}(B)$, A and B run in RR.
 - ◆ **Rule 3:** When a job enters the system, it is placed at the highest priority.
 - ◆ **Rule 4:** Once a job uses up its time allotment at a given level (regardless of how many times it has given up the CPU), its priority is reduced (i.e., it moves down one queue).
 - ◆ **Rule 5:** After some time S , move all the jobs to the topmost queue.
- ▣ Beauty of MLFQ
 - ◆ It does not require prior knowledge on the CPU usage of a process.

Lottery Scheduling, Fair-share

Proportional Share Scheduler

- ▣ Fair-share scheduler

- ◆ Guarantee that each job obtains *a certain percentage* of CPU time.
- ◆ Not optimized for JCT or response time

Basic Concept

▣ Tickets

- ◆ Represents a scheduling opportunity
- ◆ The percentage of tickets represents a process's share of the system resource in question.

▣ Example

- ◆ There are two processes, A and B.
 - Process A has 75 tickets → receive 75% of the CPU
 - Process B has 25 tickets → receive 25% of the CPU

Lottery scheduling

- The scheduler picks a winning ticket
 - ◆ Load the state of that *winning process* and runs it.

- Example

- ◆ There are 100 tickets
 - Process A has 75 tickets: 0 ~ 74
 - Process B has 25 tickets: 75 ~ 99

Scheduler's winning tickets: 63 85 70 39 76 17 29 41 36 39 10 99 68 83 63

Resulting scheduler: A B A A B A A A A A A B A B A

**The longer these two jobs compete,
the more likely they are to achieve the desired percentages.**

Ticket Mechanisms

▣ Ticket currency

- ◆ A user allocates tickets among their own jobs in whatever currency they would like.
- ◆ The system converts the currency into the correct global value.
- ◆ Example
 - There are 200 tickets (Global currency)
 - Process A has 100 tickets
 - Process B has 100 tickets

User A $\rightarrow 500$ (A's currency) to A1 $\rightarrow 50$ (global currency)
 $\rightarrow 500$ (A's currency) to A2 $\rightarrow 50$ (global currency)

User B $\rightarrow 10$ (B's currency) to B1 $\rightarrow 100$ (global currency)

Ticket Mechanisms (Cont.)

▣ Ticket transfer

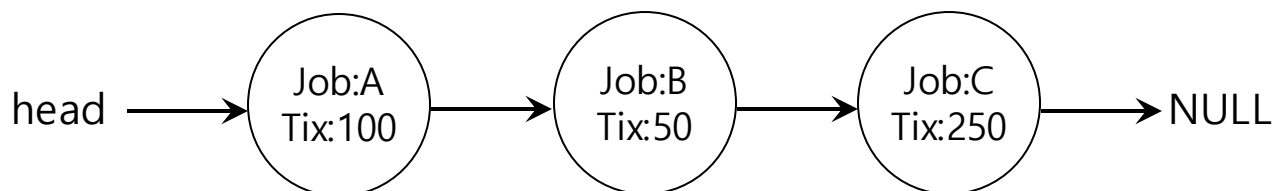
- ◆ A process can temporarily hand off *its tickets* to another process.

▣ Ticket inflation

- ◆ A process can temporarily raise or lower the number of tickets it owns.
- ◆ If any one process needs *more CPU time*, it can boost its tickets.

Implementation

- Example: There are three processes, A, B, and C.
 - Keep the processes in a list



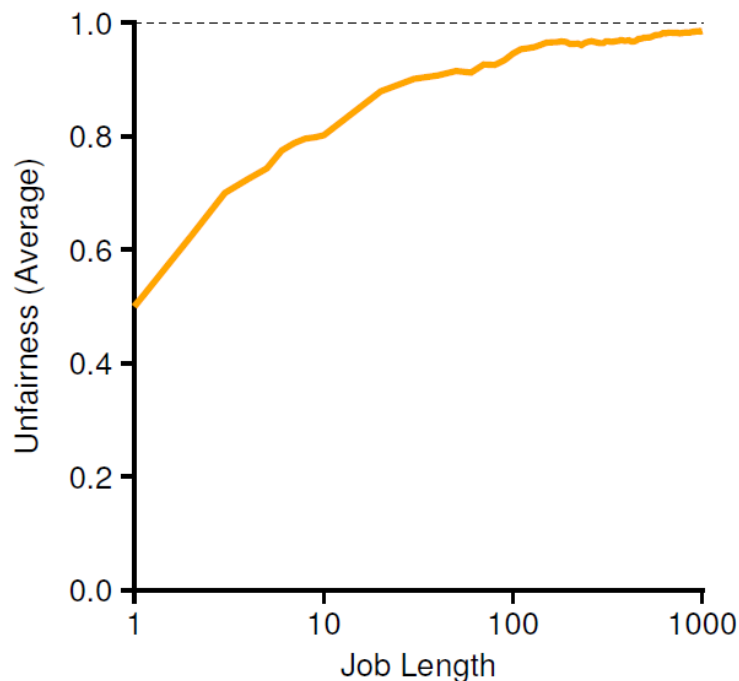
```
1      // counter: used to track if we've found the winner yet
2      int counter = 0;
3
4      // winner: use some call to a random number generator to
5      // get a value, between 0 and the total # of tickets
6      int winner = getrandom(0, totaltickets);
7
8      // current: use this to walk through the list of jobs
9      node_t *current = head;
10
11     // loop until the sum of ticket values is > the winner
12     while (current) {
13         counter = counter + current->tickets;
14         if (counter > winner)
15             break; // found the winner
16         current = current->next;
17     }
18     // 'current' is the winner: schedule it...
```

Implementation (Cont.)

- ▣ **Random** can be much more difficult than you thought!
 - ◆ <https://stackoverflow.com/questions/2509679/how-to-generate-a-random-integer-number-from-within-a-range>
- ▣ U: unfairness metric
 - ◆ The time the first job completes divided by the time that the second job completes.
- ▣ Example:
 - ◆ There are two jobs, each jobs has runtime 10
 - First job finishes at time 10; Second job finishes at time 20
 - ◆ $U = \frac{10}{20} = 0.5$
 - ◆ U will be close to 1 when both jobs finish at nearly the same time.

Lottery Fairness Study

- ▣ There are two jobs.
 - ◆ Each jobs has the same number of tickets (100).



**When the job length is not very long,
average unfairness can be quite severe.**

Deterministic Approach: Stride Scheduling

▣ Stride of each process

- ◆ A large number / number of tickets of the process
- ◆ Example: A large number = 10,000
 - Process A has 100 tickets → stride of A is 100
 - Process B has 50 tickets → stride of B is 200



- ## ▣ A process runs, increments a counter (pass value) for it by its stride
- ◆ Pick the process that has the lowest pass value to run

```
current = remove_min(queue);           // pick client with minimum pass
schedule(current);                     // use resource for quantum
current->pass += current->stride;       // compute next pass using stride
insert(queue, current);                 // put back into the queue
```

A pseudo code implementation

Stride Scheduling Example

Pass(A) (stride=100)	Pass(B) (stride=200)	Pass(C) (stride=40)	Who Runs?
0	0	0	A
100	0	0	B
100	200	0	C
100	200	40	C
100	200	80	C
100	200	120	A
200	200	120	C
200	200	160	C
200	200	200	...

Stride scheduling needs to maintain the per process pass value.
If new job enters with pass value 0 it will monopolize the CPU!

Advantage of lottery scheduling: no per-process state

The Linux Completely Fair Scheduling (CFS)

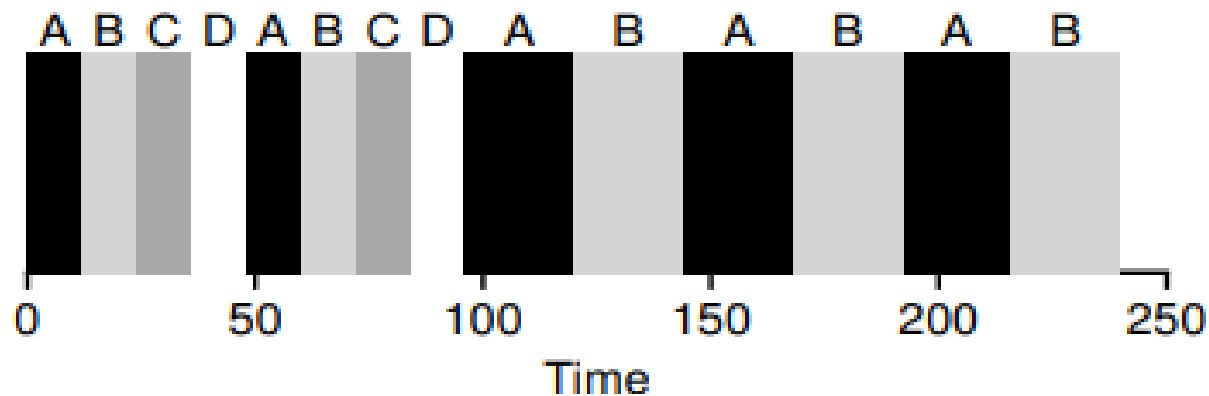
- ▣ Completely Fair Scheduling (CFS)
 - ◆ The current CPU scheduler in Linux
 - ◆ Non-fixed timeslice
 - CFS assigns process's timeslice a proportion of the processor.
 - ◆ Priority
 - Enables control over priority by using nice value.
 - ◆ Efficient data structures
 - Use red-black tree for efficient search, insertion and deletion of a process.

- ▣ Virtual runtime (vruntime)
 - ◆ Denote how long the process has been executing.
 - ◆ Per-process variable
 - ◆ Increase in **proportion with physical (real) time** when it runs.
 - ◆ CFS will pick the process with the **lowest vruntime** to run next.
- ▣ sched_latency
 - ◆ A typical value is 48 (milliseconds)
 - ◆ Process's timeslice = $\text{sched_latency} / (\text{the number of process})$

Example

- ◆ Simple Example

- 4 processes (A,B,C,D) and then 2 processes(C,D) complete.



- ◆ min_granularity

- The minimum timeslice (6ms)
- Ensure that not too much time is spent in scheduling overhead, When there are too many processes running.

- ◆ Nice value
 - CFS enables control over process priority
 - Nice parameter is an integer value and can be set from -20 to +19
 - The nice value is mapped to a weight (value is not important)

```
static const int prio_to_weight[40] = {  
    /* -20 */    88761,    71755,    56483,    46273,    36291,  
    /* -15 */    29154,    23254,    18705,    14949,    11916,  
    /* -10 */    9548,     7620,     6100,     4904,     3906,  
    /*  -5 */    3121,     2501,     1991,     1586,     1277,  
    /*   0 */    1024,      820,      655,      526,      423,  
    /*   5 */     335,      272,      215,      172,      137,  
    /*  10 */     110,       87,       70,       56,       45,  
    /*  15 */      36,       29,       23,       18,       15,  
};
```

Weighting (Niceness)

- ◆ New timeslice formula

$$time_slice_k = \frac{weight_k}{\sum_{n=0}^{n-1} weight_i} \cdot sched_latency$$

- ◆ Simple Example
 - Assign Process A a nice value of -5 and process B a nice value of 0.

Process	nice value	weight	Time slice
A	-5	3121	36 ms
B	0	1024	12 ms

vruntime with Weighting

▣ Weighting (Niceness)

◆ vruntime formula

- Calculate the actual run time; scale it **inversely** by the weight

$$vruntime_i = vruntime_i + \frac{weight_0}{weight_i} \cdot runtime_i$$

- $weight_0$ is the default weight, 1024

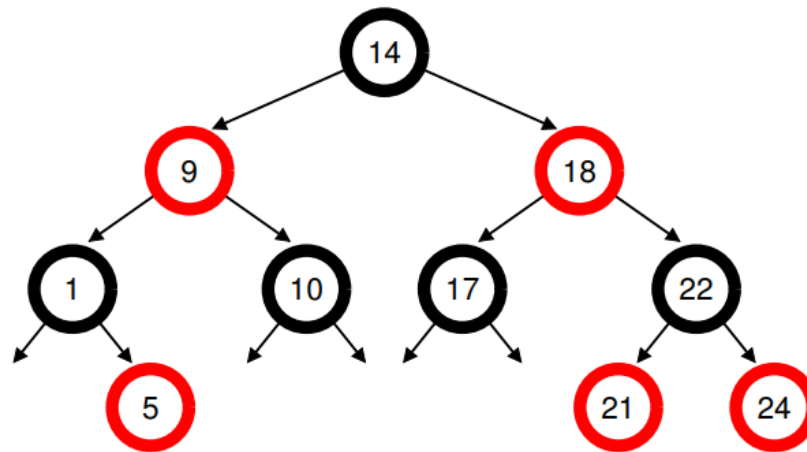
◆ Example

Process	nice value	weight	Accumulated value
A	-5	3121	1/3 * runtime
B	0	1024	1 * runtime

Structure of the Ready Queue

▣ Red-Black Tree

- ◆ Balanced binary tree (can address worst-case insertion)
- ◆ Ordering of Red-Black Tree: $O(\log n)$
- ◆ Efficiently find the process with minimum virtual runtime
- ◆ Only running (or runnable) processes are kept therein



Dealing with IO and Sleeping Processes

- ▣ Avoid the situation where some process monopolizes CPU, if it has significantly small vruntime after sleeping
- ▣ Set the vruntime of process to the minimum value found in tree when it wakes up.
- ▣ Cost: Process that sleeps for short periods of time frequently do not ever get their fair share of the CPU