

# 21CS2109AA

Operating Systems

Session 3

**CPU Scheduling, Multilevel Feedback,  
Lottery Scheduling code, Multiprocessor Scheduling.**



**K L University**

u/s 3 of UGC Act. 1956  
Koneru Lakshmalah Education Foundation



# Scheduling: Introduction

- The process scheduling is the activity of the process manager that handles the removal of the running process from the CPU and the selection of another process on the basis of a particular strategy.
- 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 metric:**

A metric is just something that we use to measure something, and there are a number of different metrics that make sense in scheduling.



# Scheduling Metrics

- Performance metric: **Turnaround time**
  - The time at which **the job completes** minus the time at which **the job arrived** in the system.
$$T_{turnaround} = T_{completion} - T_{arrival}$$
- Another metric is **fairness**.
  - Performance and fairness are often at odds in scheduling; a scheduler, for example, may optimize performance but at the cost of preventing a few jobs from running, thus decreasing fairness. This conundrum shows us that life isn't always perfect.



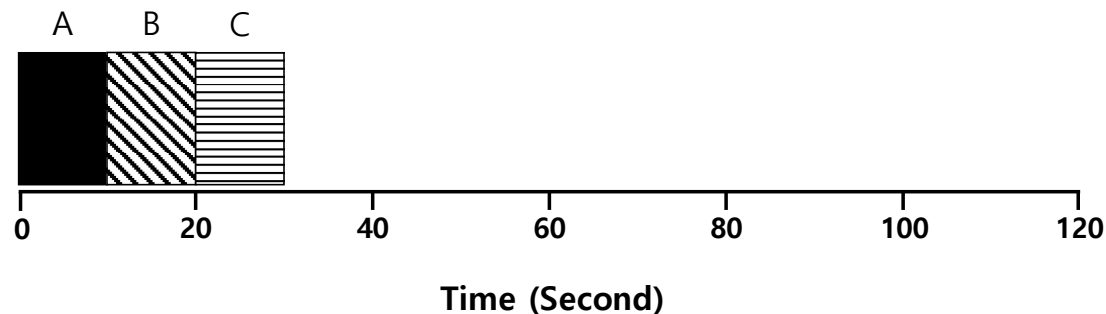
# Preemptive and Non-Preemptive

- The basic difference between **preemptive and non-preemptive scheduling** is that:
- In **preemptive scheduling**, the CPU is allocated to the processes for the limited time.
- While in **Non-preemptive scheduling**, the CPU is allocated to the process till it terminates or switches to waiting state.



# 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.

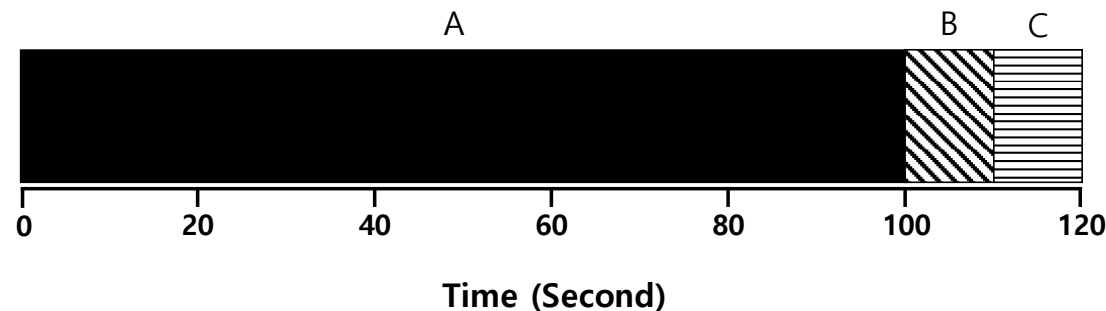


$$\text{Average turnaround time} = \frac{10 + 20 + 30}{3} = 20 \text{ sec}$$



# Why FIFO is not that great? – Convoy effect

- 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.

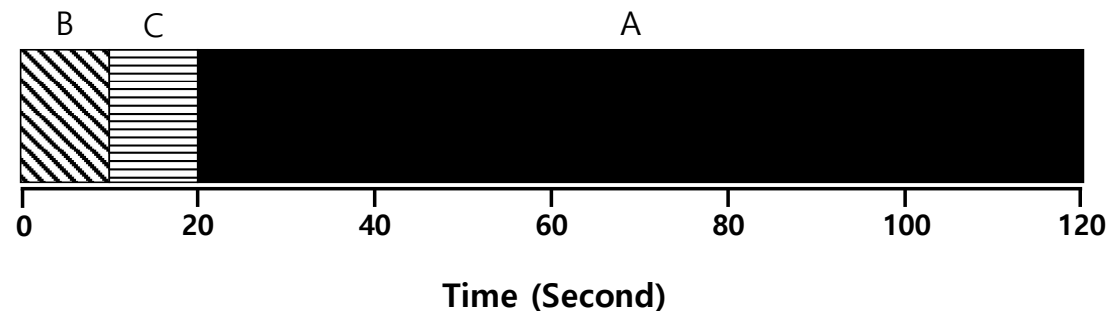


$$\text{Average turnaround time} = \frac{100 + 110 + 120}{3} = \mathbf{110 \text{ sec}}$$



# Shortest Job First (SJF)

- Run the shortest job first, then the next shortest, and so on
  - Non-preemptive scheduler
- Example:
  - A arrived just before B which arrived just before C.
  - A runs for 100 seconds, B and C run for 10 each.

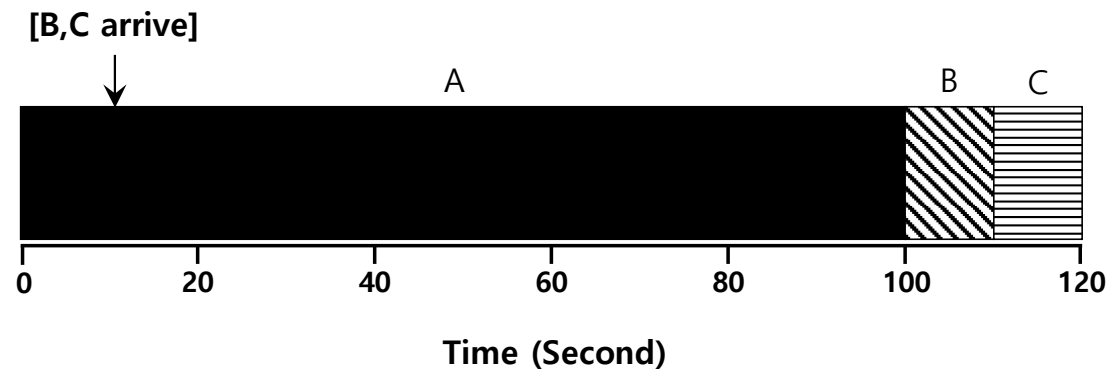


$$\text{Average turnaround time} = \frac{10 + 20 + 120}{3} = 50 \text{ sec}$$



# 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 need to run for 10 seconds



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





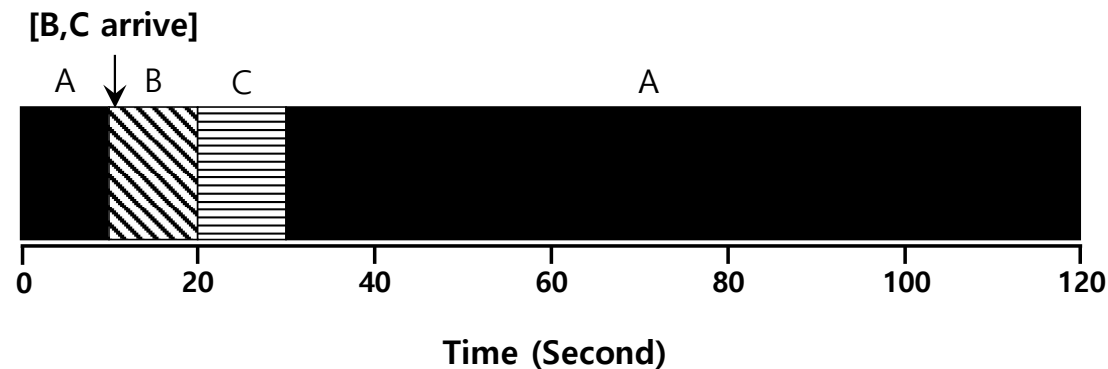
# Shortest Time-to-Completion First (STCF)

- Add **preemption** to SJF
  - Also known as Preemptive Shortest Job First (PSJF)
- A new job enters the system:
  - Determine of the remaining jobs and new job
  - Schedule the job which has the least time left



# Shortest Time-to-Completion First (STCF)

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



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



# New scheduling metric: Response time

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

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

- STCF and related disciplines are not particularly good for response time.
- Why means? if we had the schedule above (with A arriving at time 0, and B and C at time 10), the response time of each job is as follows: 0 for job A, 0 for B, and 10 for C for previous example.

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

- If three jobs arrive at the same time, for example, the third job has to wait for the previous two jobs to run in their entirety before being scheduled just once.
- While great for turnaround time, this approach is quite bad for response time and interactivity.
- Indeed, imagine sitting at a terminal, typing, and having to wait 10 seconds to see a response from the system just because some other job got scheduled in front of yours: not too pleasant.

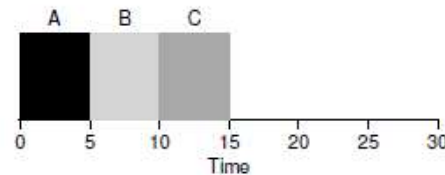


Figure 7.6: SJF Again (Bad for Response Time)



# Round Robin (RR) Scheduling

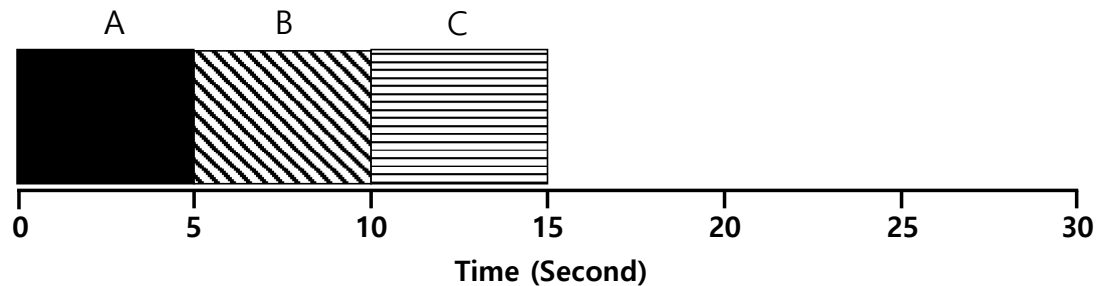
- Time slicing Scheduling
  - Run a job for a **time slice** and then switch to the next job in the **run queue** until the jobs are finished.
    - Time slice is sometimes called a scheduling quantum.
  - It repeatedly does so until the jobs are finished.
  - The length of a time slice must be *a multiple of* the timer-interrupt period.
- Thus if the timer interrupts every 10 milliseconds, the time slice could be 10, 20, or any other multiple of 10 ms.

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



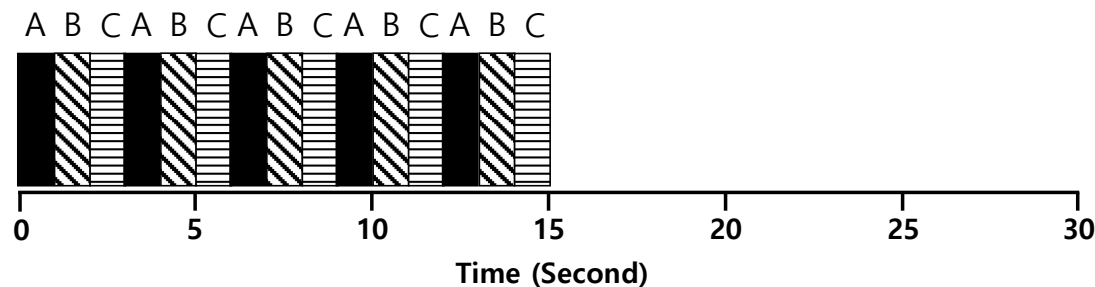
# RR Scheduling Example

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



**SJF (Bad for Response Time)**

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



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

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



# The length of the time slice is critical

- The shorter it is, the better the performance of RR under the response-time metric.
- However, making the time slice too short is problematic: suddenly the cost of context switching will dominate overall performance.
- Thus, deciding on the length of the time slice presents a trade-off to a system designer, making it long enough to **amortize the cost of switching without** making it so long that the system is no longer responsive.
- Note that the cost of context switching does not arise solely from the OS actions of saving and restoring a few registers.

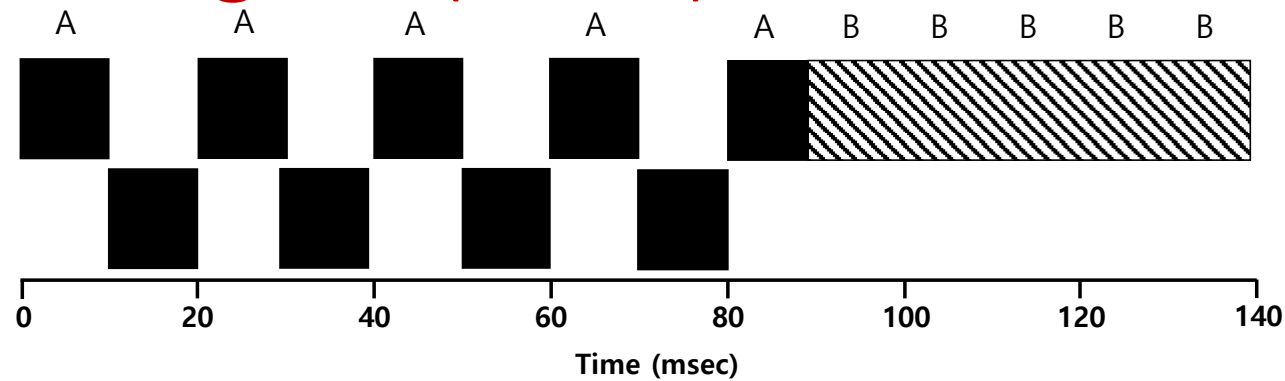


# Incorporating I/O

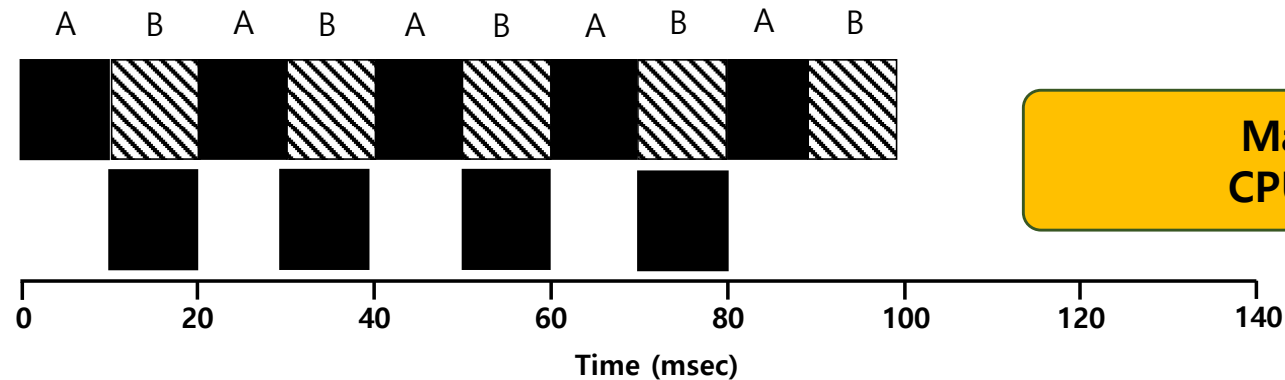
- Let's relax assumption 3: All programs perform I/O
- Example:
  - A and B need 50ms of CPU time each.
  - A runs for 10ms and then issues an I/O request
    - I/Os each take 10ms
  - B simply uses the CPU for 50ms and performs no I/O
  - The scheduler runs A first, then B after



# 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 another job on the CPU.
- When the I/O completes
  - An interrupt is raised.
  - The OS moves the process from blocked back to the ready state.



# Multi-Level Feedback Queue (MLFQ)

- A Scheduler that learns from the past to predict the future.
- Objective:
  - Optimize **turnaround time** → Run shorter jobs first
  - Minimize **response time** without *a priori knowledge of job length*.



# MLFQ: Basic Rules

- MLFQ has a number of distinct **queues**.
  - Each queues is assigned a different priority level.
- At any given time, a job that is ready to run is on a single queue.
  - A job **on a higher queue** is chosen to run.
  - Use round-robin scheduling among jobs in the same queue



## MLFQ: Basic Rules (Cont.)

- Rather than giving a fixed priority to each job, 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.

**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 Example

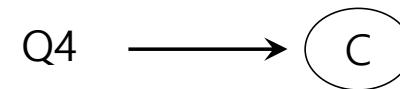
**[High Priority]**



Q7

Q6

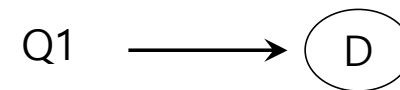
Q5



Q3

Q2

**[Low Priority]**





- In the figure, two jobs (A and B) are at the highest priority level, while job C is in the middle and Job D is at the lowest priority.
- Given our current knowledge of how MLFQ works, the scheduler would just alternate time slices between A and B because they are the highest priority jobs in the system.
- poor jobs C and D would never even get to run – an outrage!
- We need to understand how job priority changes over time.



# MLFQ: How to Change Priority

- MLFQ priority adjustment algorithm:
  - **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

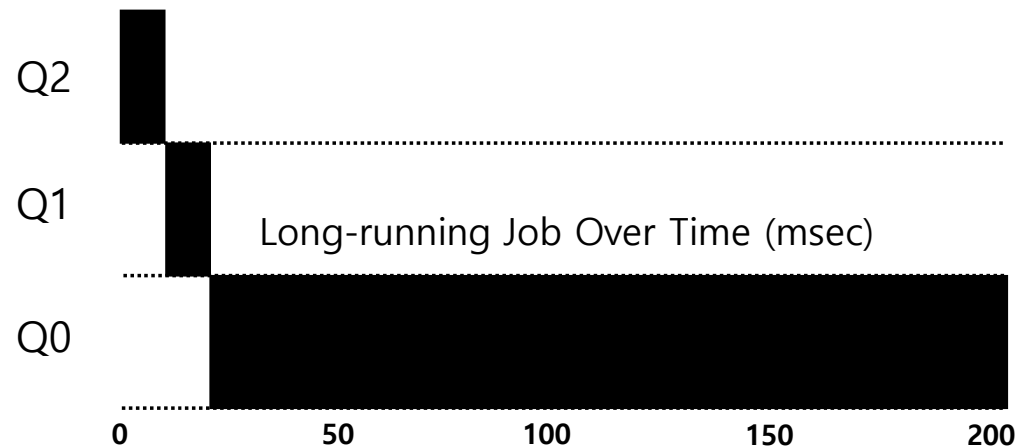
**In this manner, MLFQ approximates SJF**





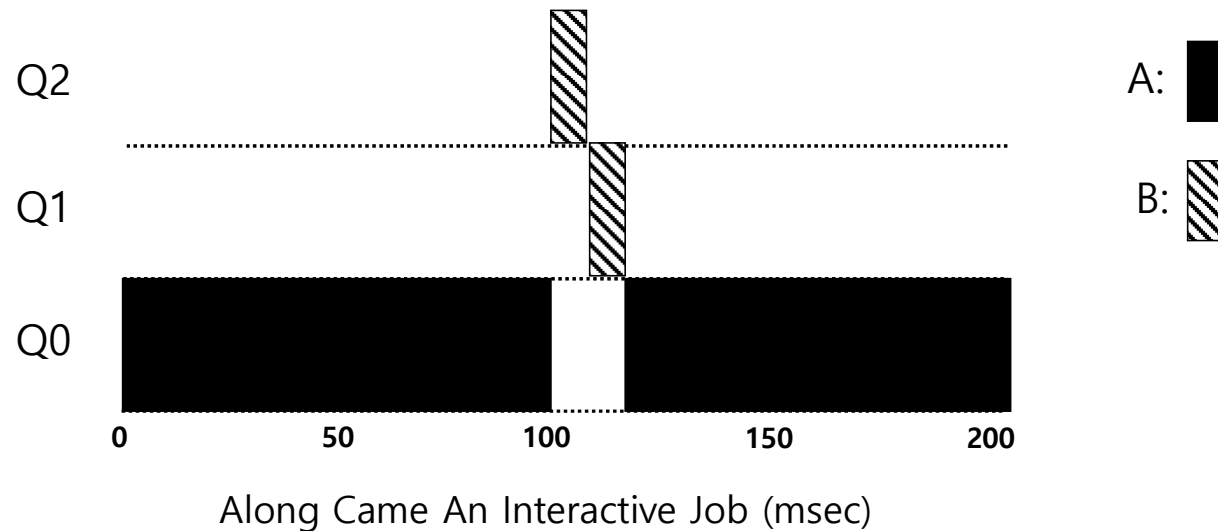
## Example 1: A Single Long-Running Job

- The job enters at the highest priority(Q2). After a single time-slice of 10 ms, the scheduler reduces the job's priority by one, and thus the job is on Q1. After running at Q1 for a time slice, the job is finally lowered to the lowest priority in the system (Q0), where it remains.



## 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  $T=100$ .

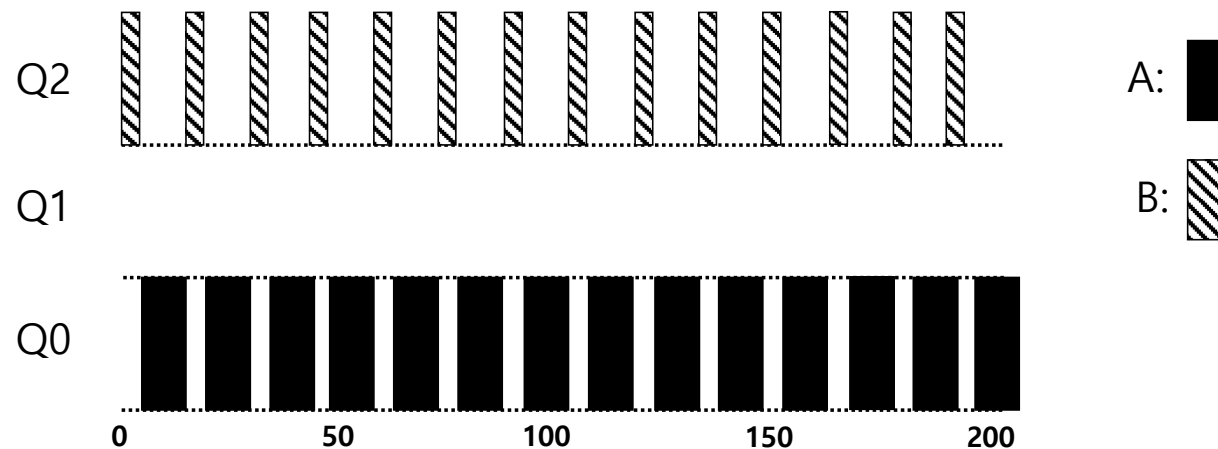




- A is running along in the lowest-priority queue; B arrives at time  $T = 100$ , and thus is inserted into the highest queue; as its run-time is short (only 20 ms), B completes before reaching the bottom queue, in two time slices; then A resumes running (at low priority).
- From this example, you can hopefully understand one of the major goals of the algorithm: because it doesn't know whether a job will be a short job or a long-running job, it first assumes it might be a short job, thus giving the job high priority.
- If it actually is a short job, it will run quickly and complete; if it is not a short job, it will slowly move down the queues, and thus soon prove itself to be a long-running more batch-like process.
- In this manner, MLFQ approximates SJF.

## Example 3: What About I/O?

- Assumption:
  - Job A:** A long-running CPU-intensive job
  - Job B:** An interactive job that need the 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

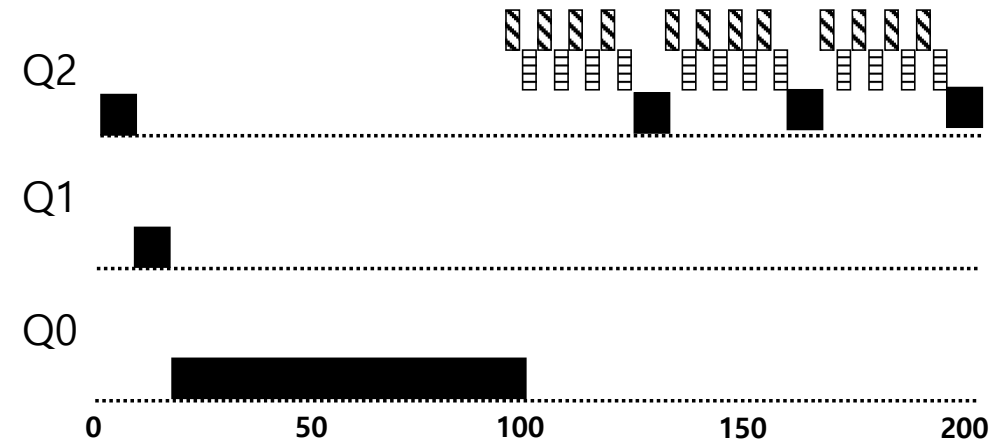
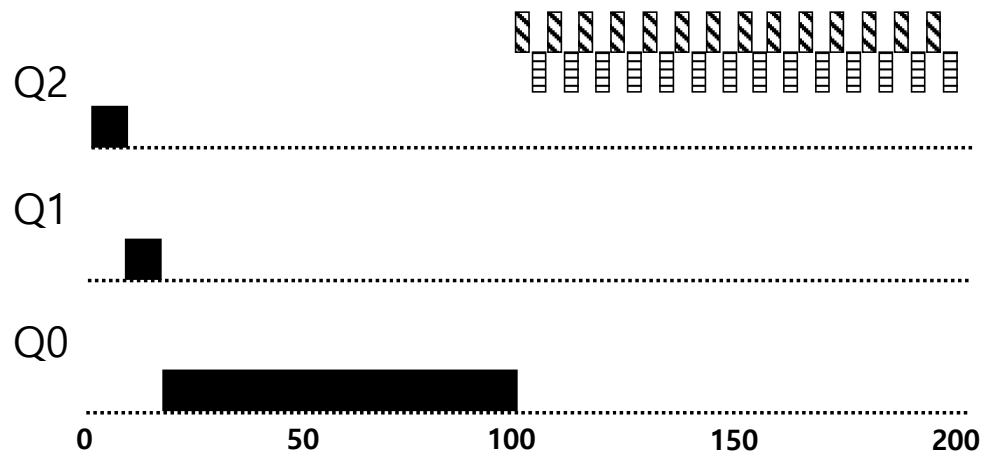
- First, there is the problem of **starvation**:  
**if there are “too many” interactive jobs** in the system, they will combine to consume all CPU time, and thus long-running jobs will never receive any CPU time (they **starve**).
- Second, a smart user could rewrite their program to **game the scheduler**.
  - The algorithm we have described is susceptible to the following attack: before the time slice is over, issue an I/O operation and thus relinquish the CPU; doing so allows you to remain in the same queue, and thus gain a higher percentage of CPU time. When done right (e.g., by running for 99% of a time slice before relinquishing the CPU), a job could nearly monopolize the CPU.






- Finally, a program may change its behavior over time; what was CPU bound may transition to a phase of interactivity. With our current approach, such a job would be out of luck and not be treated like the other interactive jobs in the system.

# The Priority Boost

- **Rule 5:** After some time period  $S$ , move all the jobs in the system to the topmost queue.
- Example:
  - A long-running job(A) with two short-running interactive job(B, C)

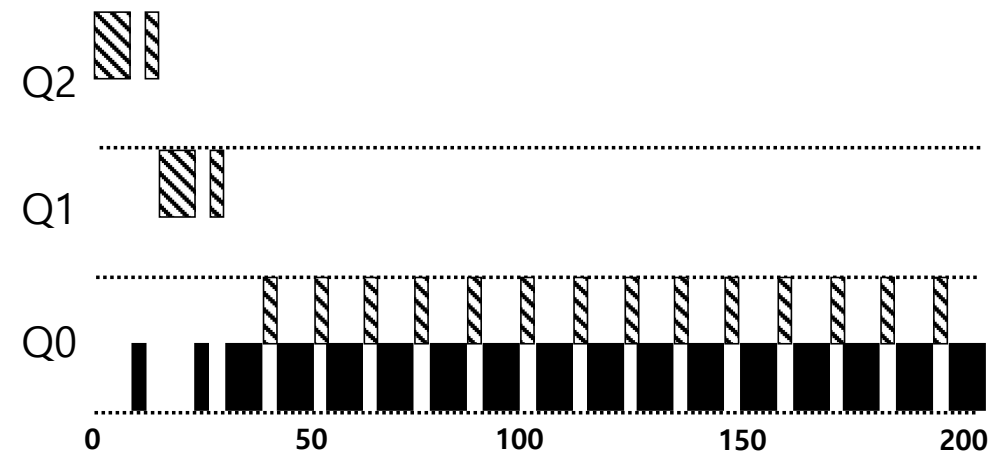
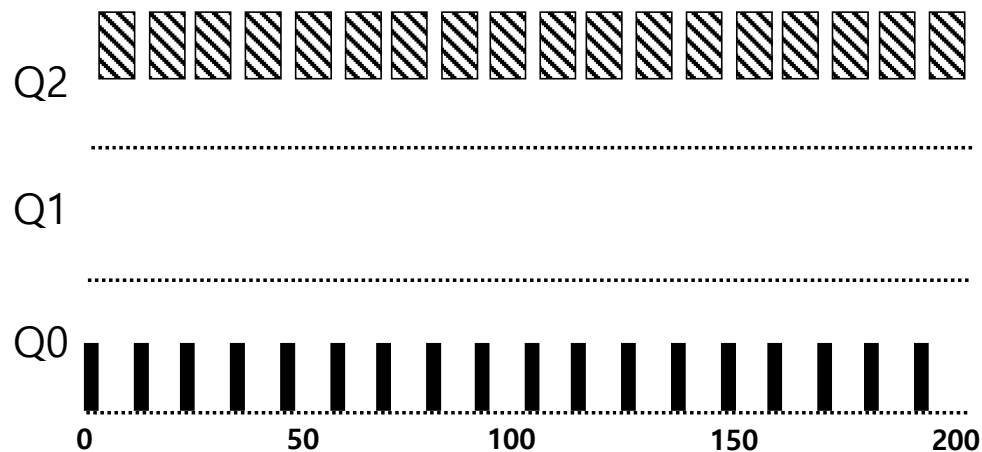


Without(Left) and With(Right) Priority Boost

A:  B:  C: 

# Better Accounting

- How to prevent gaming of 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





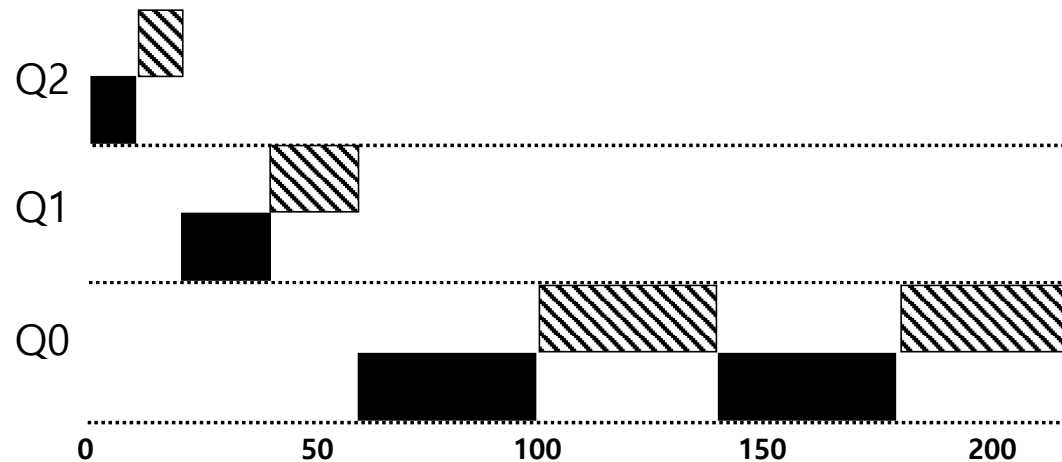
## Tuning MLFQ And Other Issues

- How many queues should there be?
- How big should the time slice be per queue?
- How often should priority be boosted in order to avoid starvation and account for changes in behavior?

# Tuning MLFQ And Other Issues

## Lower Priority, Longer Quanta

- The high-priority queues → Short time slices
  - E.g., 10 or fewer milliseconds
- The Low-priority queue → Longer time slices
  - E.g., 100 milliseconds



Example) 10ms for the highest queue, 20ms for the middle,  
40ms for the lowest



# The Solaris MLFQ implementation

- For the Time-Sharing scheduling class (TS)
  - 60 Queues
  - Slowly increasing time-slice length
    - The highest priority: 20msec
    - The lowest priority: A few hundred milliseconds
  - 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 (B doesn't).
  - **Rule 2:** If  $\text{Priority}(A) = \text{Priority}(B)$ , A & 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 on queue).
  - **Rule 5:** After some time period S, move all the jobs in the system to the topmost queue.



# Proportional Share Scheduler

- Also referred as **Fair-share** scheduler.
- Proportional-share is based around a simple concept:
- instead of optimizing for turnaround or response time, a scheduler might instead try to guarantee that each job obtain a certain percentage of CPU time.



# Basic Concept for Lottery scheduling

- Tickets
  - Represent the share of a resource that a process should receive
  - The percent of tickets represents its 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

- Lottery scheduling also provides a number of mechanisms to manipulate tickets in different and sometimes useful ways.
- One way is with the concept of **ticket currency**.
- Currency allows a user with a set of tickets to allocate tickets among their own jobs in whatever currency they would like; the system then automatically converts said currency into the correct global value.
- Scheduler give a certain number of tickets to different users in a currency and users can give it to there processes in a different currency





# Ticket Mechanisms

- Example

- There are 200 tickets (Global currency)
- Process A has 100 tickets
- Process B has 100 tickets

**User A** → 500 (A's currency) to A1 → 50 (global currency)  
→ 500 (A's currency) to A2 → 50 (global currency)

**User B** → 10 (B's currency) to B1 → 100 (global currency)

- The lottery will then be held over the global ticket currency (200 total) to determine which job runs.



## Ticket Mechanisms (Cont.)

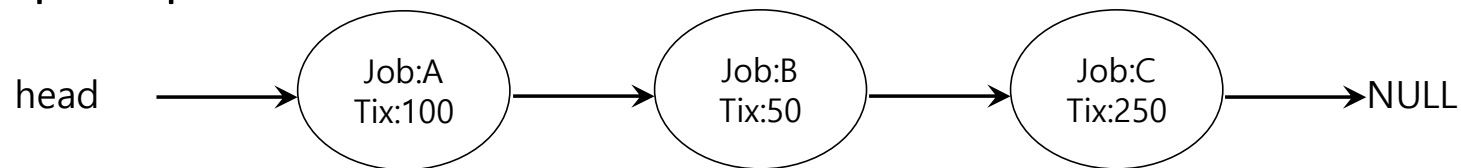
- Another useful mechanism is **ticket transfer**.
- With transfers, a process can temporarily hand off its tickets to another process.
- This ability is especially useful in a client/server setting, where a client process sends a message to a server asking it to do some work on the client's behalf.
- To speed up the work, the client can pass the tickets to the server and thus try to maximize the performance of the server while the server is handling the client's request.
- When finished, the server then transfers the tickets back to the client and all is as before.



- Finally, **ticket inflation can sometimes be a useful technique**
- With 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...
```



- The code walks the list of processes, adding each ticket value to counter until the value exceeds winner. Once that is the case, the current list element is the winner.
- With our example of the winning ticket being 300, the following takes place.
  - First, counter is incremented to 100 to account for A's tickets; because 100 is less than 300, the loop continues. Then counter would be updated to 150 (B's tickets), still less than 300 and thus again we continue. Finally, counter is updated to 400 (clearly greater than 300), and thus we break out of the loop with current pointing at C (the winner).



# Implementation (Cont.)

- ▣  $U$ : unfairness metric

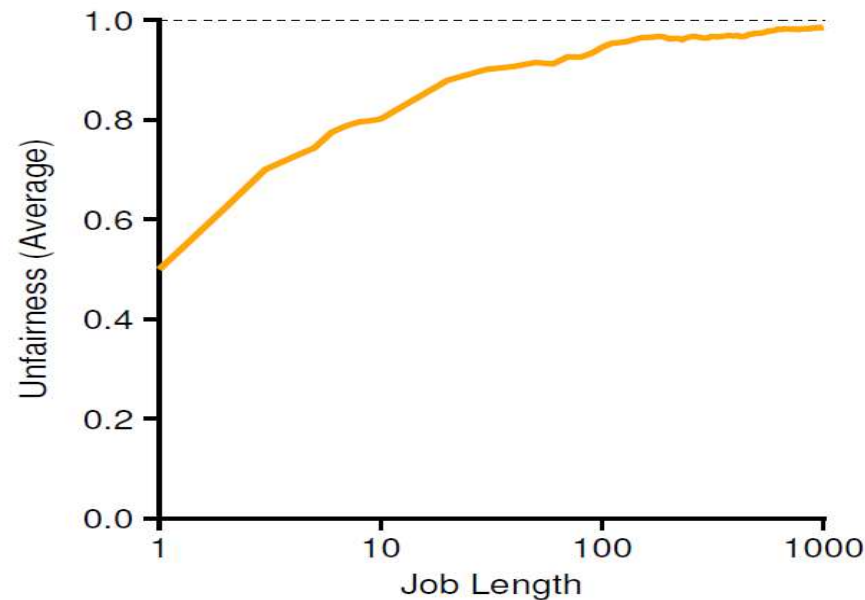
- ◆ The time the first job completes divided by the time that the second job completes.

- ▣ 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.**



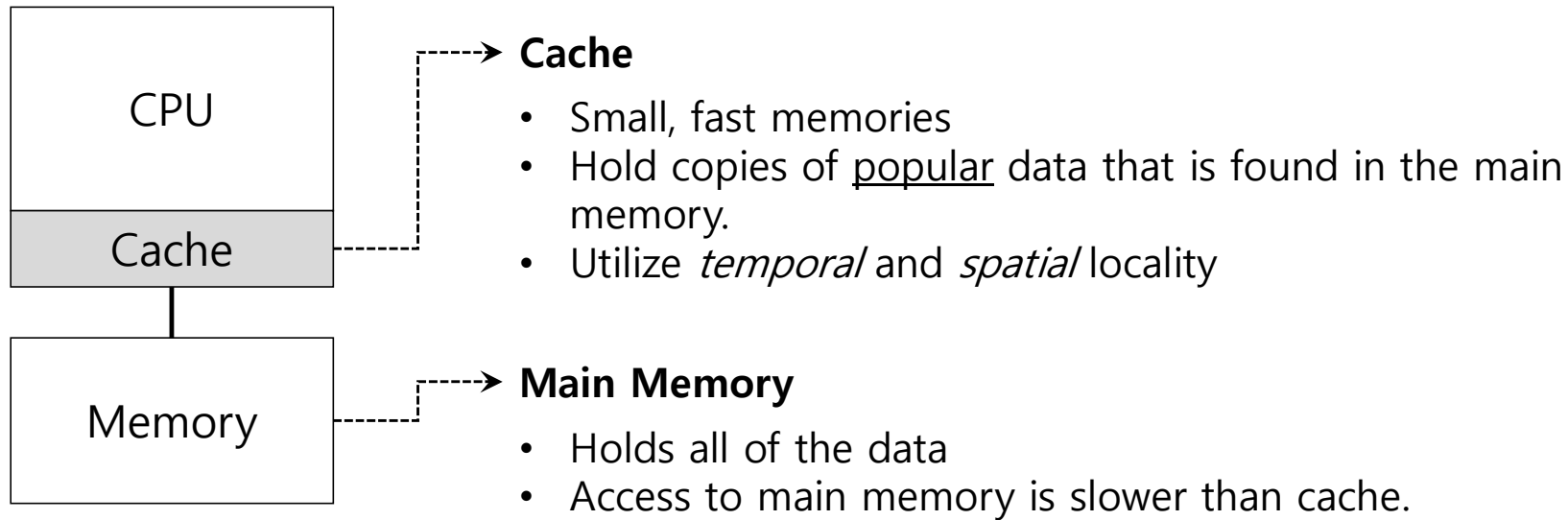
# Multiprocessor Scheduling

- The rise of the **multicore processor** is the source of multiprocessor-scheduling proliferation.
  - **Multicore**: Multiple CPU cores are packed onto a single chip.
- Adding more CPUs does not make that single application run faster.  
→ You'll have to rewrite application to run in parallel, using **threads**.

How to schedule jobs on **Multiple CPUs**?



# Single CPU with cache

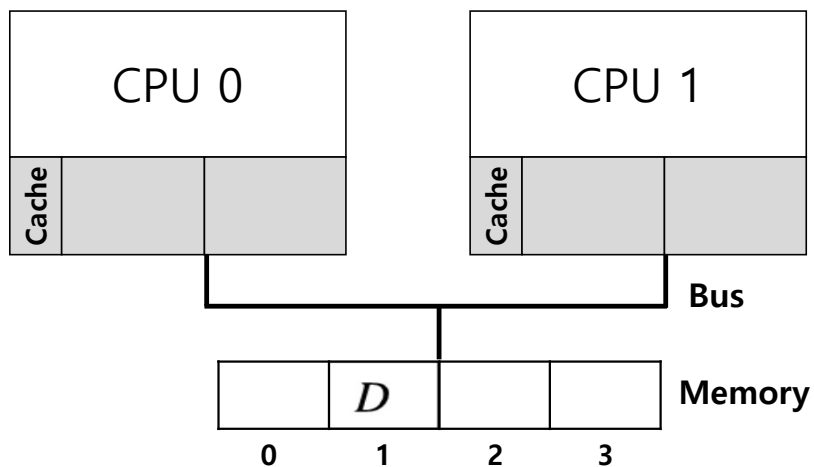


By keeping data in cache, the system can make slow memory **appear to be a fast one**

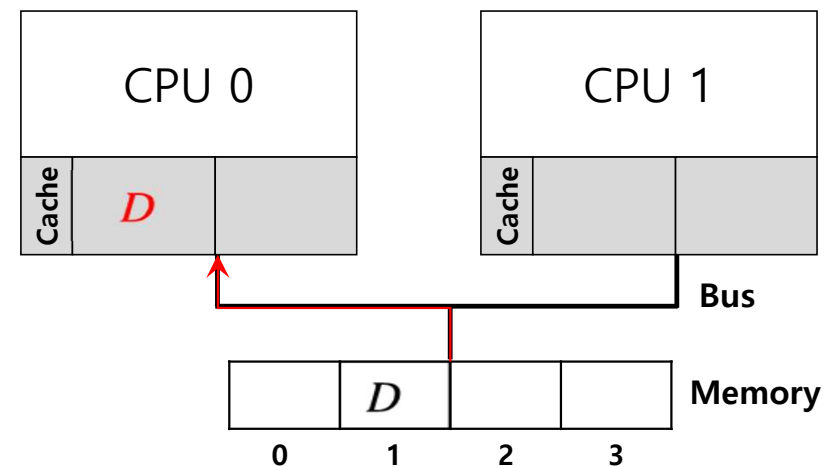
# Cache coherence

- Consistency of shared resource data stored in multiple caches.

0. Two CPUs with caches sharing memory



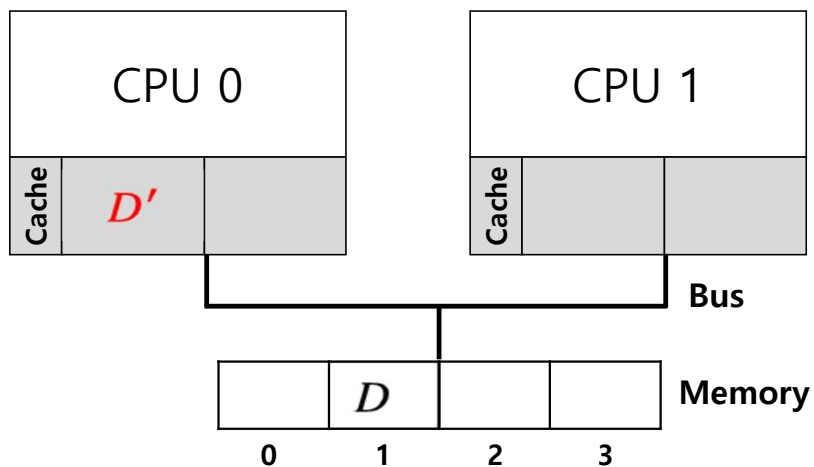
1. CPU0 reads a data at address 1.



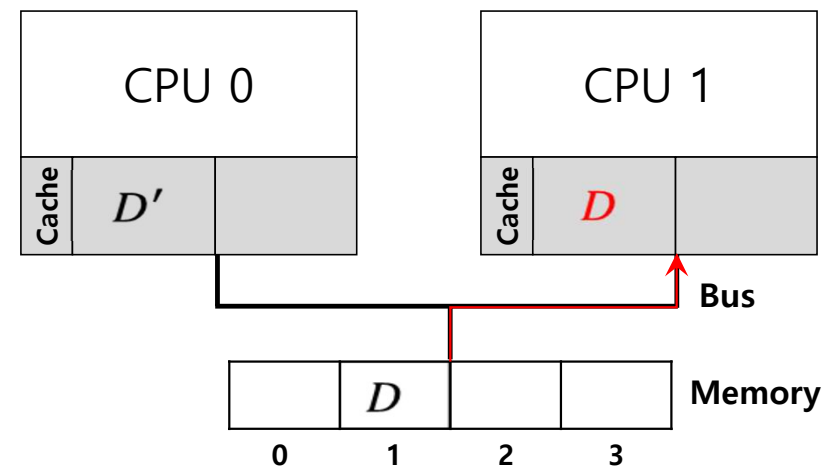


# Cache coherence (Cont.)

2.  $D$  is updated and CPU1 is scheduled.



3. CPU1 re-reads the value at address A



**CPU1 gets the old value  $D$  instead of the correct value  $D'$ .**



# Cache coherence solution

- Bus snooping
  - Each cache pays attention to memory updates by **observing the bus**.
  - When a CPU sees an update for a data item it holds in its cache, it will notice the change and either invalidate its copy or update it.



# Don't forget synchronization

- When accessing shared data across CPUs, **mutual exclusion** primitives should likely be used to guarantee correctness.

```
1      typedef struct __Node_t {
2          int value;
3          struct __Node_t *next;
4      } Node_t;
5
6      int List_Pop() {
7          Node_t *tmp = head;          // remember old head ...
8          int value = head->value;      // ... and its value
9          head = head->next;            // advance head to next pointer
10         free(tmp);                   // free old head
11         return value;                 // return value at head
12     }
```

## Simple List Delete Code



# Don't forget synchronization (Cont.)

- Solution

```
1  pthread_mutex_t m;  
2  typedef struct __Node_t {  
3      int value;  
4      struct __Node_t *next;  
5  } Node_t;  
6  
7  int List_Pop() {  
8      lock(&m)  
9      Node_t *tmp = head;           // remember old head ...  
10     int value = head->value;       // ... and its value  
11     head = head->next;             // advance head to next pointer  
12     free(tmp);                    // free old head  
13     unlock(&m)  
14     return value;                 // return value at head  
15 }
```

**Simple List Delete Code with lock**



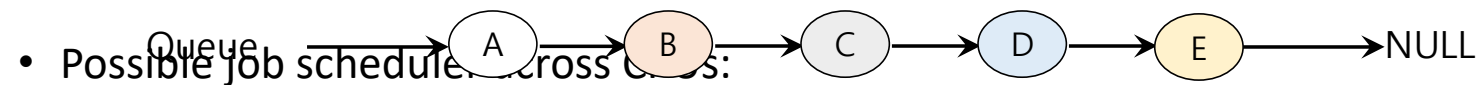
# Cache Affinity

- Keep a process on **the same CPU** if at all possible
  - A process builds up a fair bit of state in the cache of a CPU.
  - The next time the process run, it will run faster if some of its state is *already present* in the cache on that CPU.

**A multiprocessor scheduler should consider **cache affinity** when making its scheduling decision.**

# Single queue Multiprocessor Scheduling (SQMS)

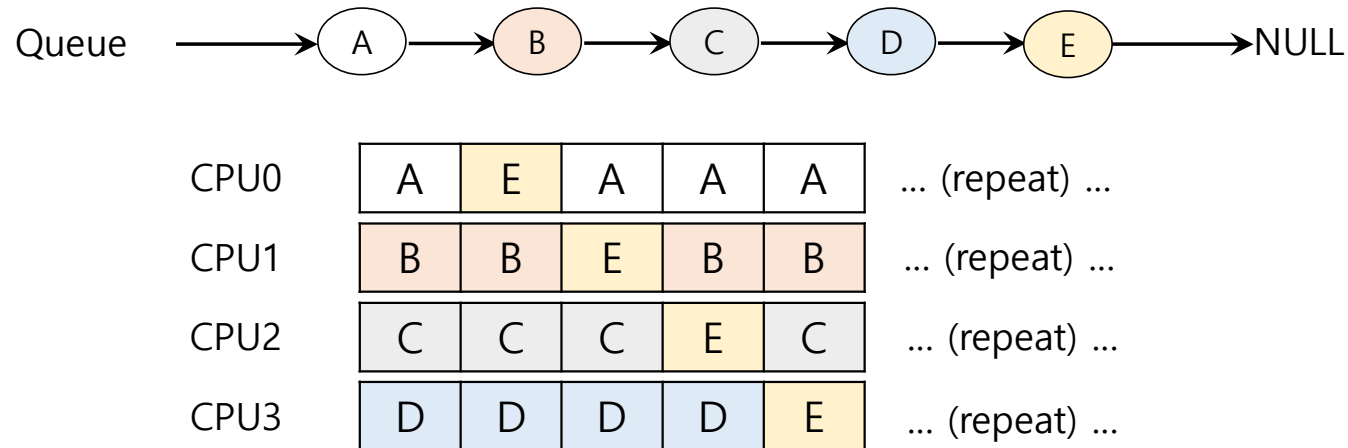
- Put all jobs that need to be scheduled into a single queue.
  - Each CPU simply picks the next job from the globally shared queue.
  - Cons:
    - Some form of **locking** have to be inserted → Lack of scalability
    - Cache affinity
    - Example:



|      |   |   |   |   |   |                  |
|------|---|---|---|---|---|------------------|
| CPU0 | A | E | D | C | B | ... (repeat) ... |
| CPU1 | B | A | E | D | C | ... (repeat) ... |
| CPU2 | C | B | A | E | D | ... (repeat) ... |
| CPU3 | D | C | B | A | E | ... (repeat) ... |



# Scheduling Example with Cache affinity



- Preserving affinity for most
  - Jobs A through D are not moved across processors.
  - Only job e Migrating from CPU to CPU.
- Implementing such a scheme can be **complex**.

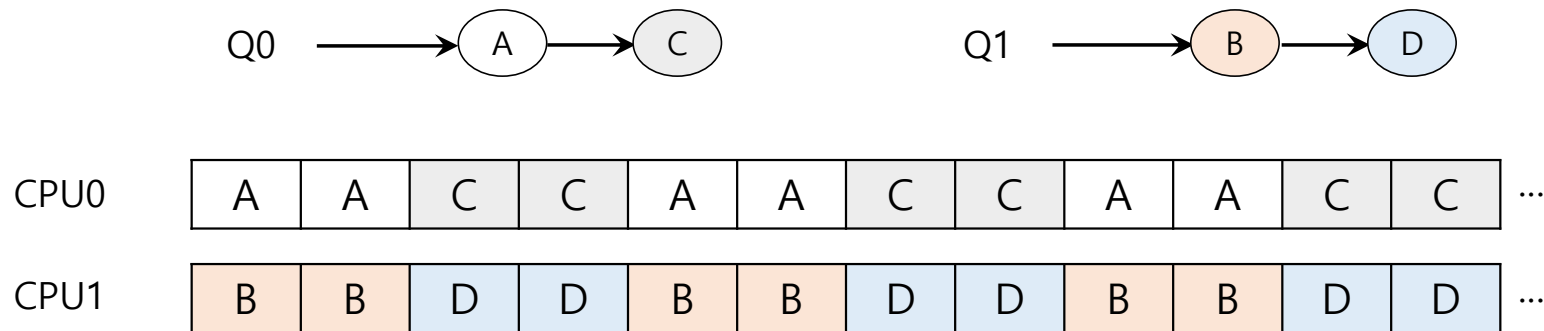


# Multi-queue Multiprocessor Scheduling (MQMS)

- MQMS consists of **multiple scheduling queues**.
  - Each queue will follow a particular scheduling discipline.
  - When a job enters the system, it is placed on **exactly one** scheduling queue.
  - Avoid the problems of information sharing and synchronization.

# MQMS Example

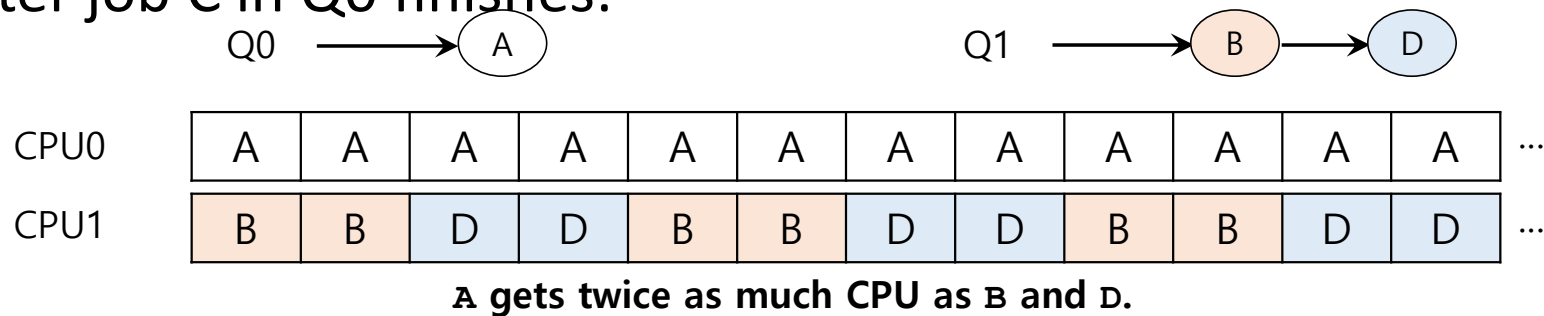
- With **round robin**, the system might produce a schedule that looks like this:



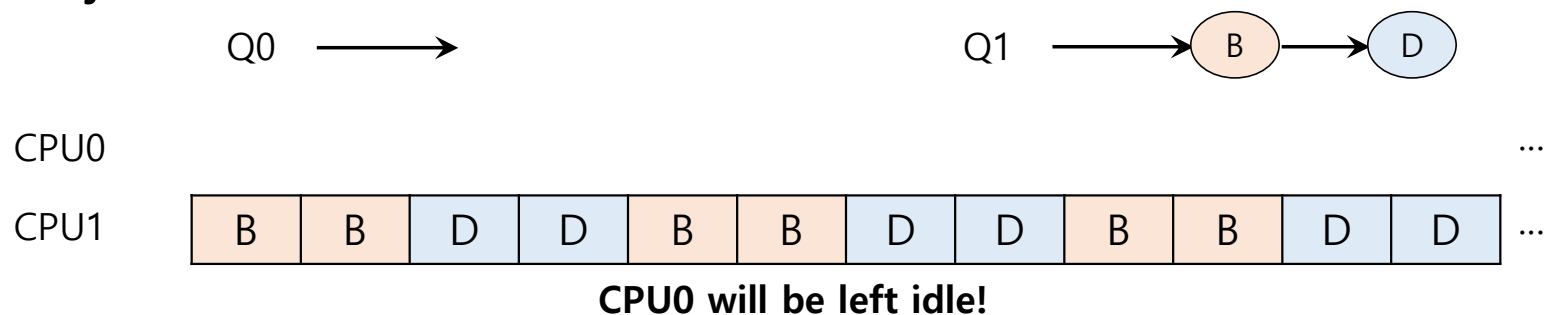
**MQMS provides more scalability and cache affinity.**

# Load Imbalance issue of MQMS

- After job C in Q0 finishes:



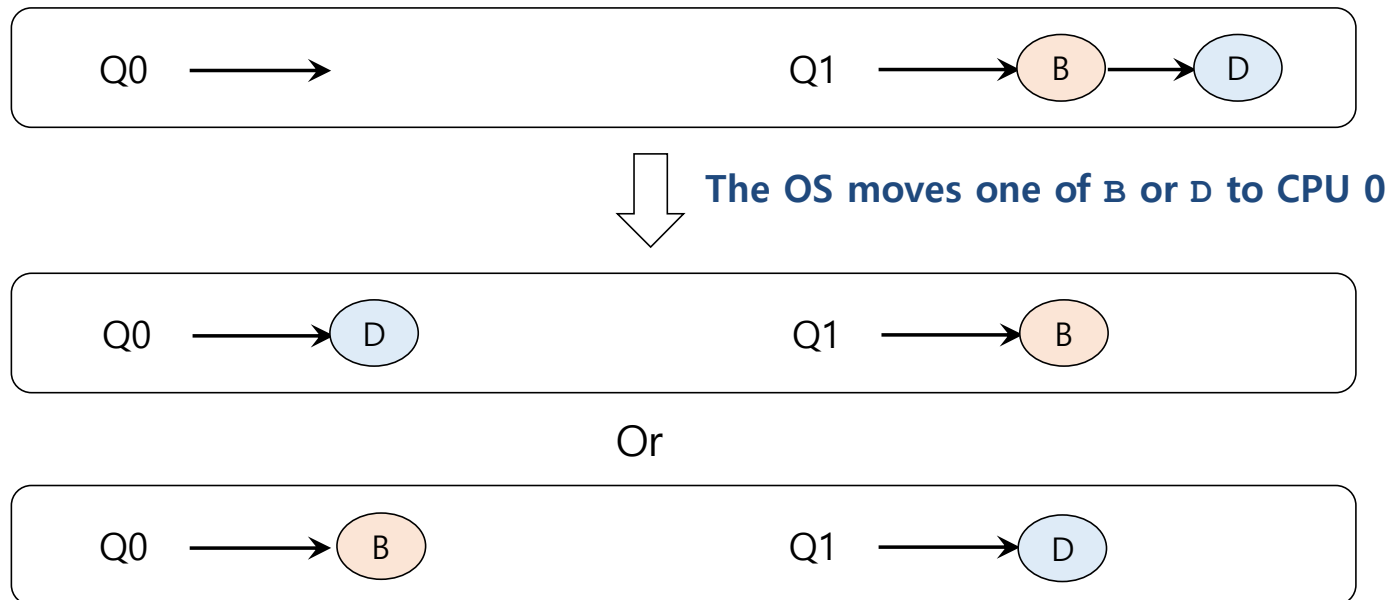
- After job A in Q0 finishes:



# How to deal with load imbalance?

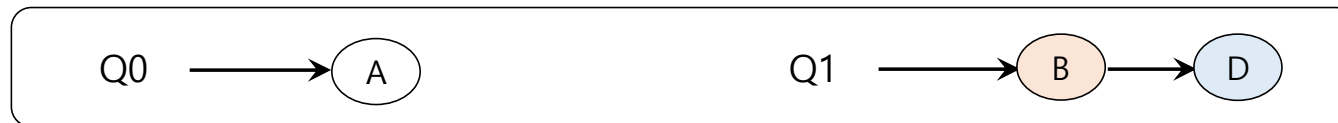
- The answer is to move jobs (**Migration**).

- Example:



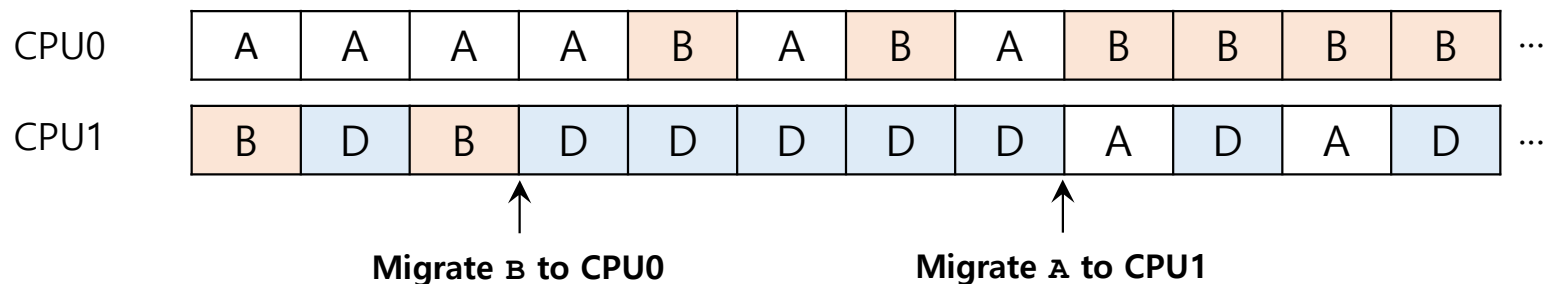
# How to deal with load imbalance? (Cont.)

- A more tricky case:



- A possible migration pattern:

- Keep switching jobs





# Work Stealing

- Move jobs between queues
  - Implementation:
    - A source queue that is low on jobs is picked.
    - The source queue occasionally peeks at another target queue.
    - If the target queue is more full than the source queue, the source will “**steal**” one or more jobs from the target queue.
  - Cons:
    - *High overhead* and trouble *scaling*



# Linux Multiprocessor Schedulers

- $O(1)$ 
  - A Priority-based scheduler
  - Use Multiple queues
  - Change a process's priority over time
  - Schedule those with highest priority
  - Interactivity is a particular focus
- Completely Fair Scheduler (CFS)
  - Deterministic proportional-share approach
  - Multiple queues