Lecture 8: Virtualizing CPU - Scheduling

The Course Organization (Bottom-up)

Concurrency
- Race Conditions, Lock/Semaphore
- Thread

Virtualization
- Memory Management — HW#4
- Process and CPU Scheduling — HW#3

Persistence
- IO Devices and Storage
- File System

HW#2 & Project

System Calls (User-level Programming) — HW#1

Course Overview

Part I. 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

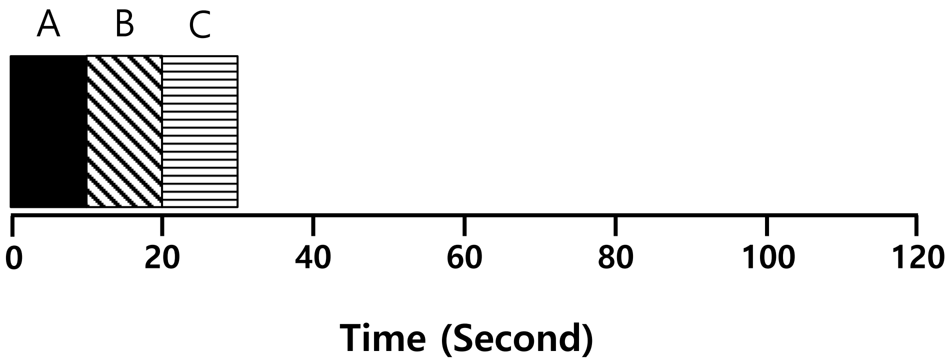
- Performance metric: Turnaround time
  - The time at which **the job completes** minus the time at which **the job arrived** in the system.

$$\boxed{T_{turnaround} = T_{completion} - T_{arrival}}$$

- Another metric is fairness.
  - Performance and fairness are often at odds in scheduling.

First In, First Out (FIFO)

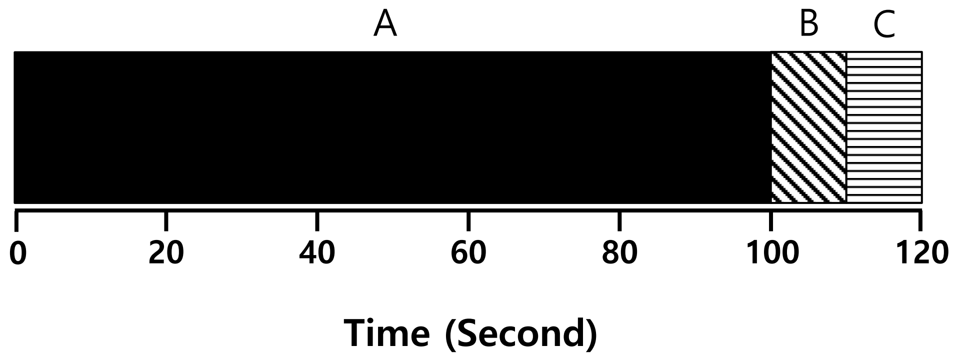
- First Come, First Served (FCFS)
  - Very simple and easy to implement
- Example:
  - A arrived just before B which arrived just before C.
  - Each job runs for 10 seconds.



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

Why FIFO is not that great? – Convoy effect

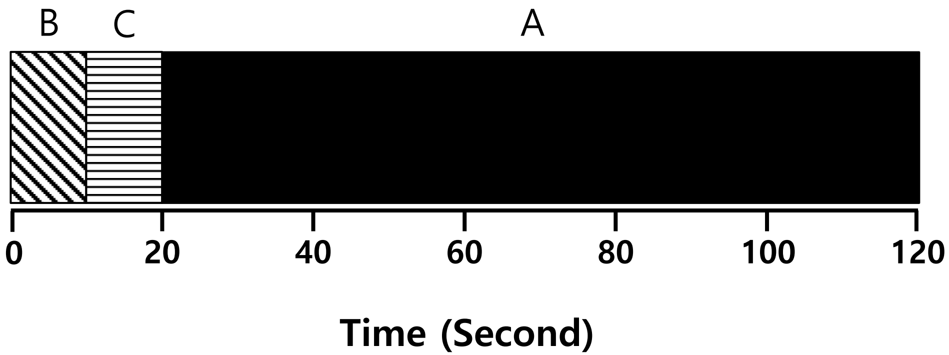
- Let's relax assumption 1: Each job **no longer** runs for the same amount of time.
- Example:
  - A arrived just before B which arrived just before C.
  - A runs for 100 seconds, B and C run for 10 each.



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

Shortest Job First (SJF)

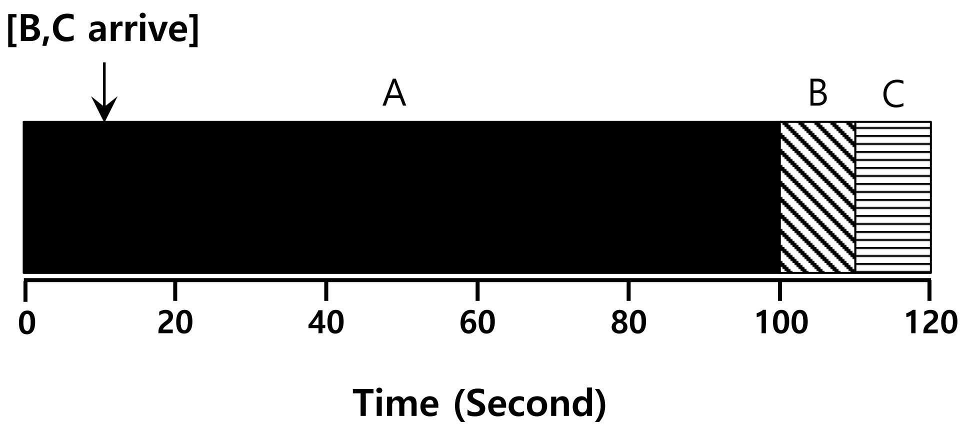
- Run the shortest job first, then the next shortest, and so on
  - Non-preemptive scheduler
- Example:
  - A arrived just before B which arrived just before C.
  - A runs for 100 seconds, B and C run for 10 each.



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

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



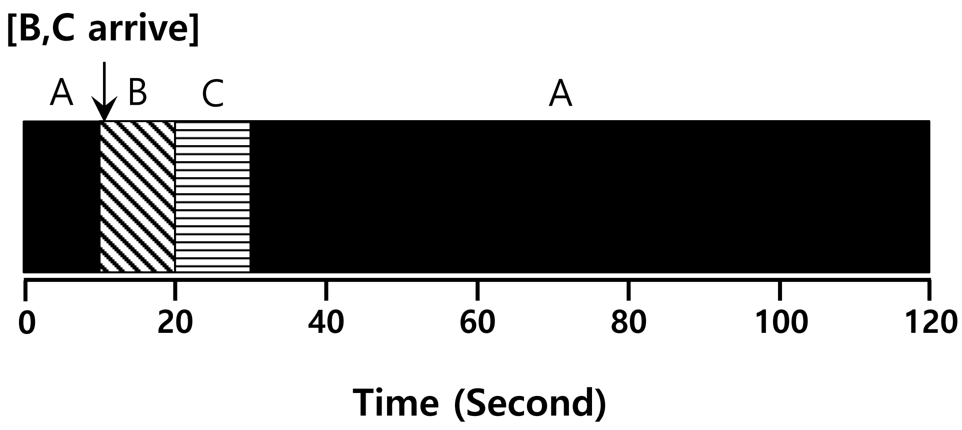
$$Average\ turnaround\ time = \frac{100 + (110 - 10) + (120 - 10)}{3} = 103.33\ sec$$

Shortest Time-to-Completion First (STCF)

- Add preemption to SJF
  - Also knows 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



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

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.

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

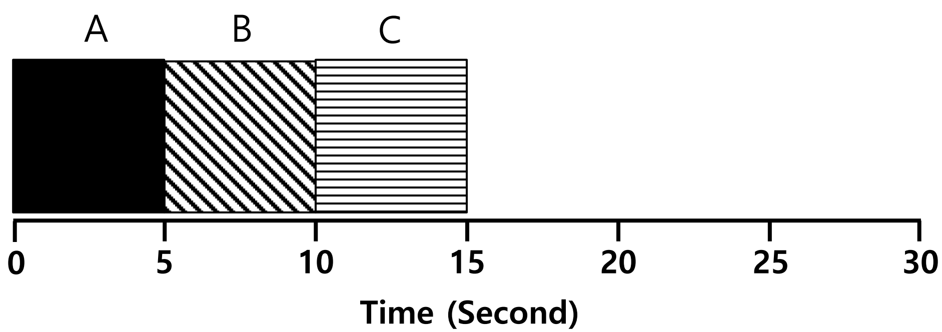
Round Robin (RR) Scheduling

- 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.

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

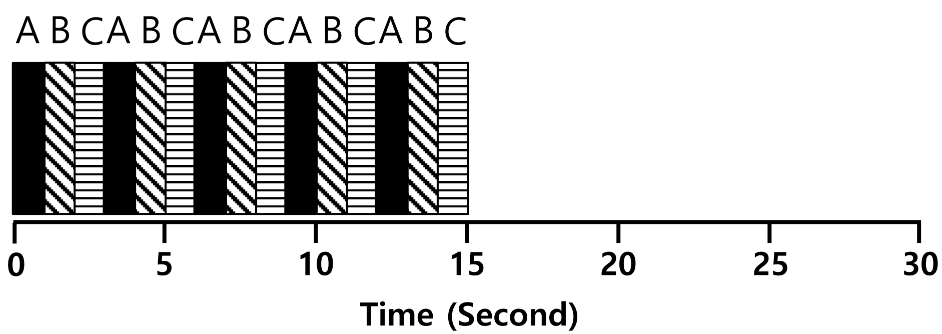
RR Scheduling Example

- 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 time slice
  - Better response time
  - The cost of context switching will dominate overall performance.

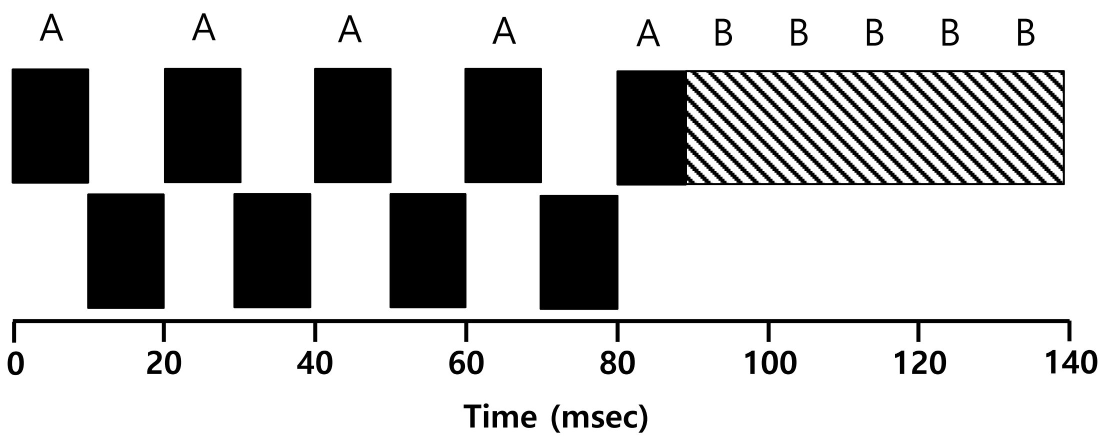
- The longer time slice
  - Amortize the cost of switching
  - Worse response time

Deciding on the length of the time slice presents a trade-off to a system designer

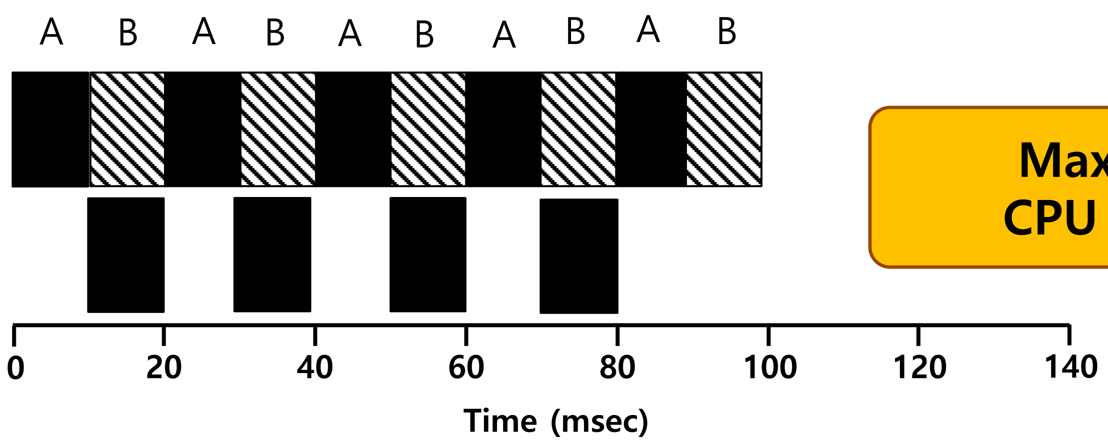
Incorporating I/O

- 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.

Part II: Scheduling: The Multi-Level Feedback Queue

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.

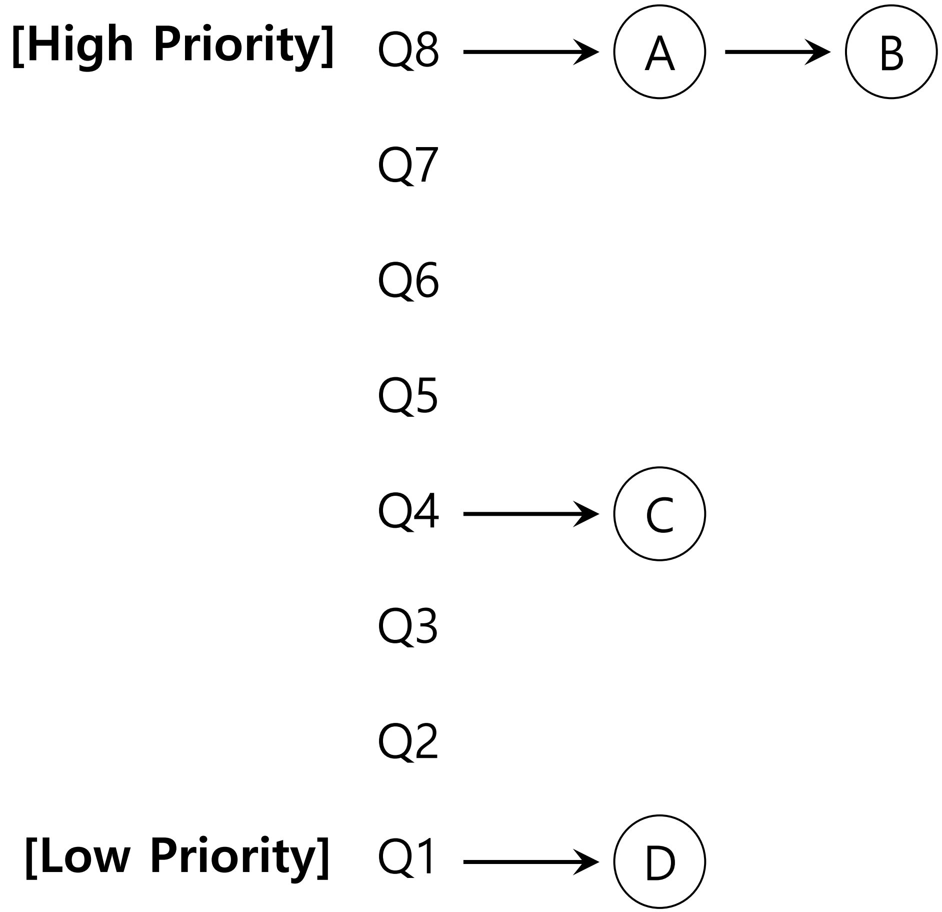
- A job that is ready to run is on a single queue.
  - A job **on a higher queue** is chosen to run.
  - Use round-robin scheduling among jobs in the same queue

Rule 1: If Priority(A) > Priority(B), A runs (B doesn't).
Rule 2: If Priority(A) = 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



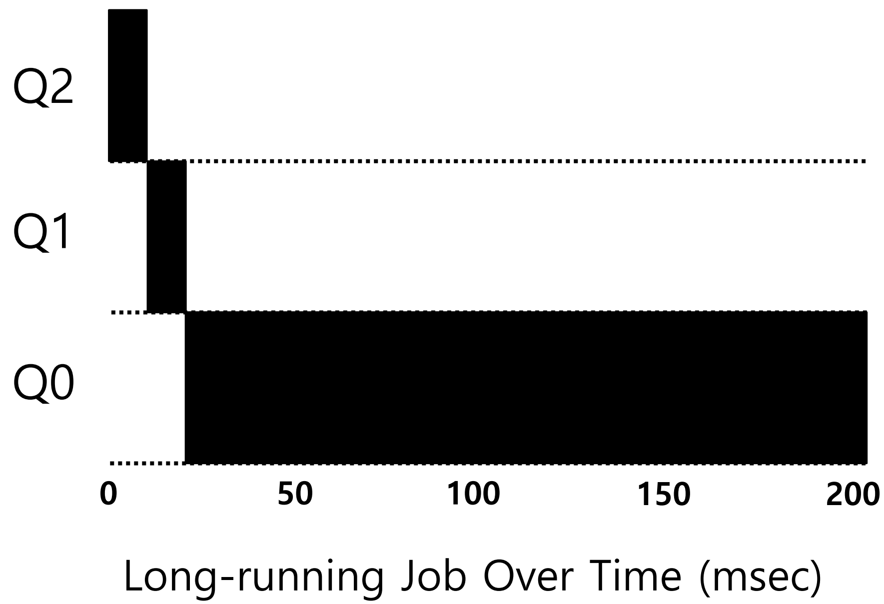
MLFQ: How to Change Priority

- MLFQ priority adjustment 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

- A three-queue scheduler with time slice 10ms

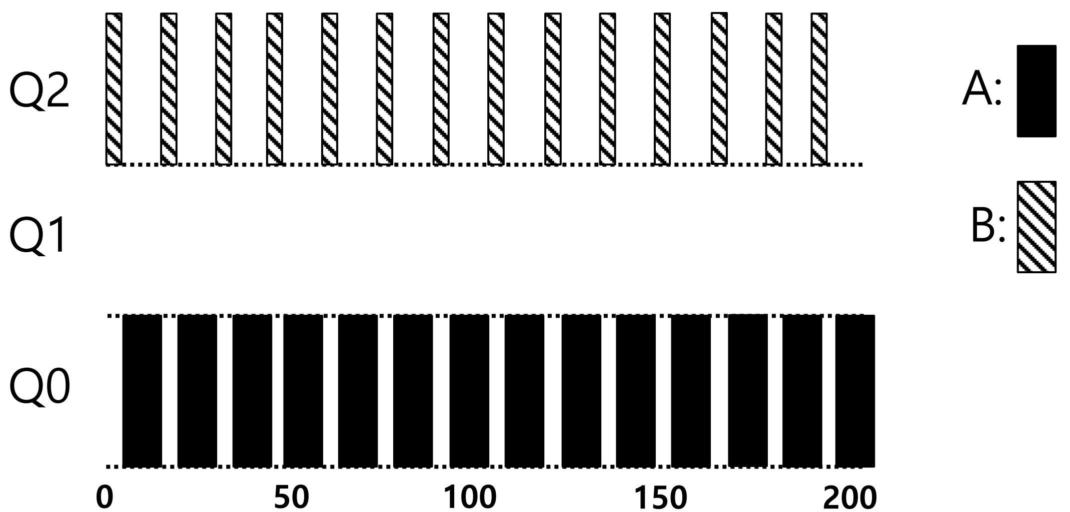


Example 2: Along Came a Short Job

- 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 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

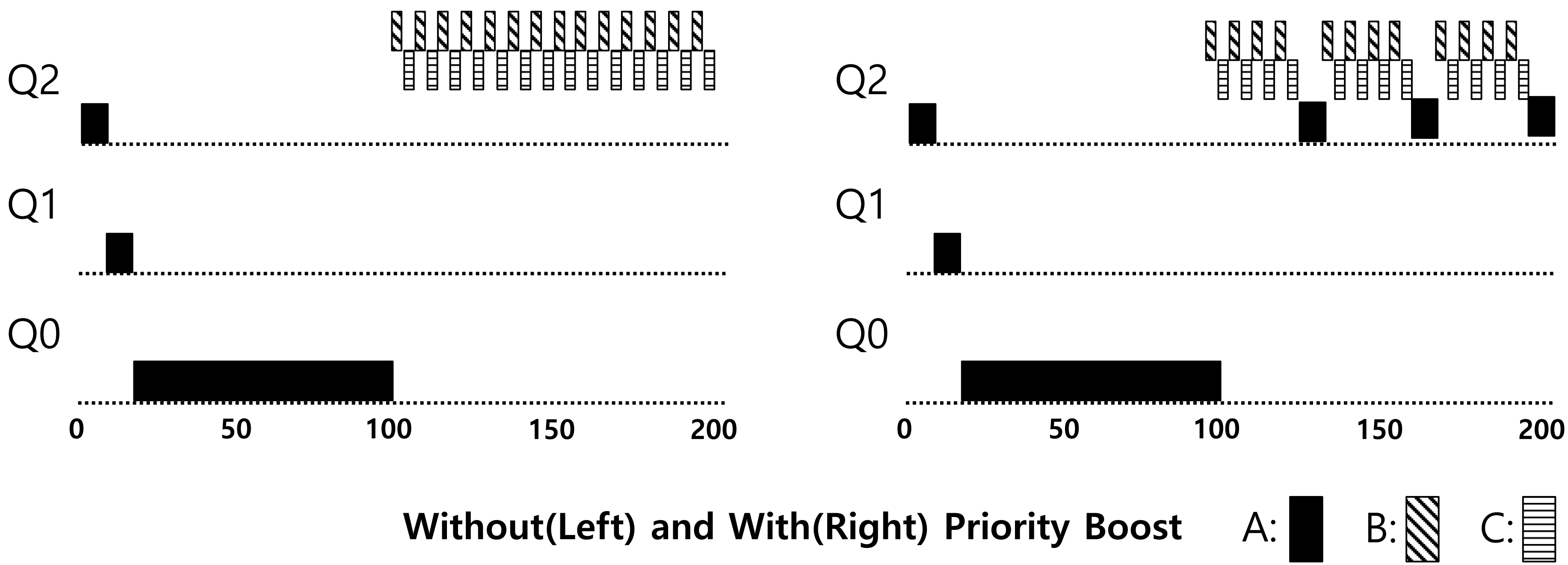
- Starvation
  - If there are “too many” interactive jobs in the system.
  - Long-running jobs will never receive any CPU time.

- Game the scheduler
  - After running 99% of a time slice, issue an I/O operation.
  - The job gains a higher percentage of CPU time.

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

The Priority Boost

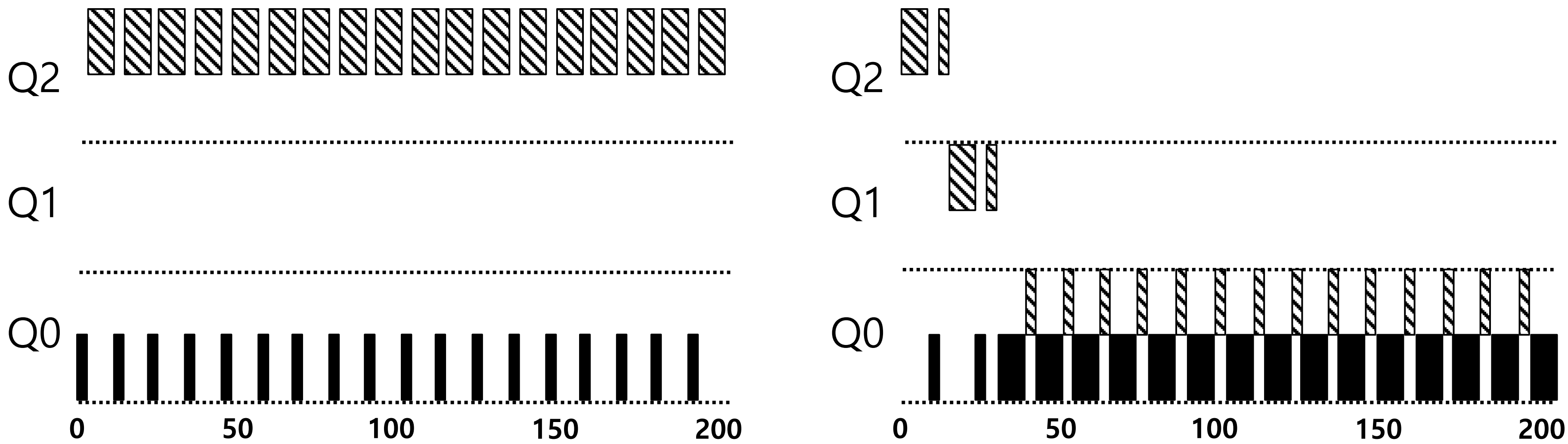
- **Rule 5:** After some time period S, move all the jobs in the system to the topmost queue.
  - Example:
    - A long-running job(A) with two short-running interactive job(B, C)



Without(Left) and With(Right) Priority Boost

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).

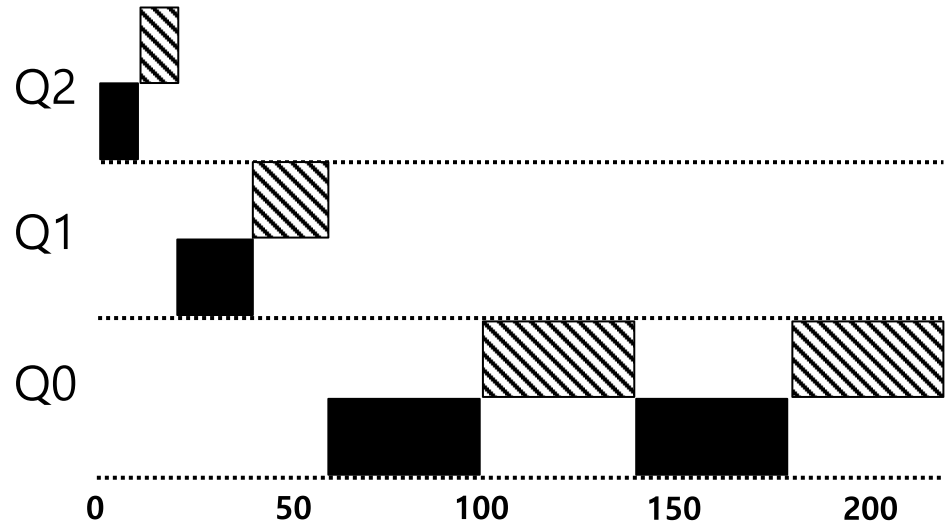


Without(Left) and With(Right) Gaming Tolerance

Tuning MLFQ And Other Issues

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 Priority(A) > Priority(B), A runs (B doesn't).
  - **Rule 2:** If Priority(A) = 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.

Part III: Scheduling: Proportional Share

Proportional Share Scheduler

- Fair-share scheduler
  - Guarantee that each job obtain *a certain percentage* of CPU time.
  - Not optimized for turnaround or response time

Basic Concept

- Tickets
  - Represent the share of a resource that a process should receive
  - <u>The percent of tickets</u> 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

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

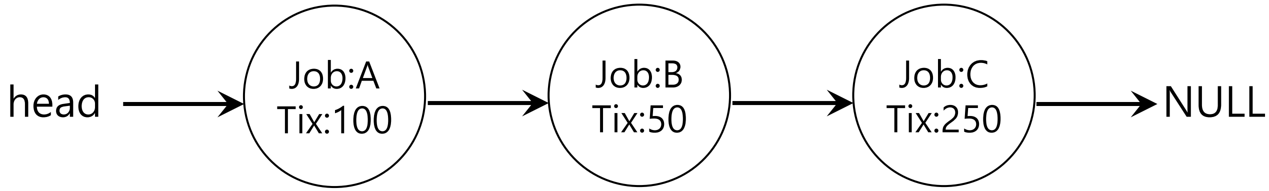
User B → *10* (B's currency) to B1 → *100* (global currency)

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:



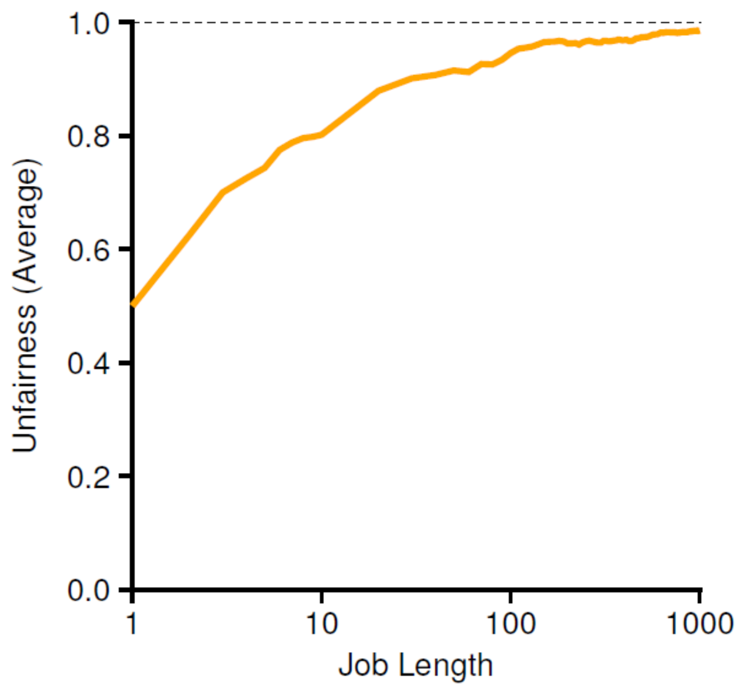

```
// counter: used to track if we've found the winner yet
int counter = 0;

// winner: use some call to a random number generator to
// get a value, between 0 and the total # of tickets
int winner = getrandom(0, totaltickets);

// current: use this to walk through the list of jobs
node_t *current = head;

// loop until the sum of ticket values is > the winner
while (current) {
        counter = counter + current->tickets;
        if (counter > winner)
                break; // found the winner
        current = current->next;
}
// 'current' is the winner: schedule it...
```

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.

Stride Scheduling

- Stride of each process
  - (A large number) / (the 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, increment a counter(=pass value) for it by its stride.
  - Pick the process to run that has the lowest pass value

```
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	…

If new job enters with pass value 0,
It will monopolize the CPU!

Part IV. Multiprocessor Scheduling

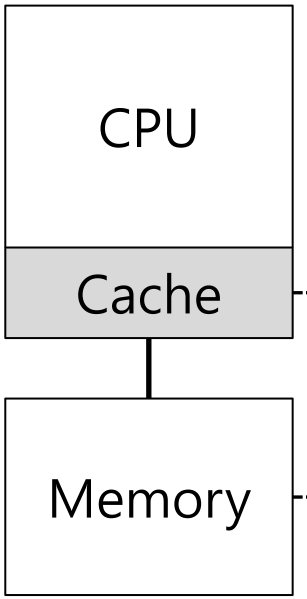
Multiprocessor Scheduling

- 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



Cache

- Small, fast memories
- Hold copies of popular data that is found in the main memory.
- Utilize *temporal* and *spatial* locality

Main Memory

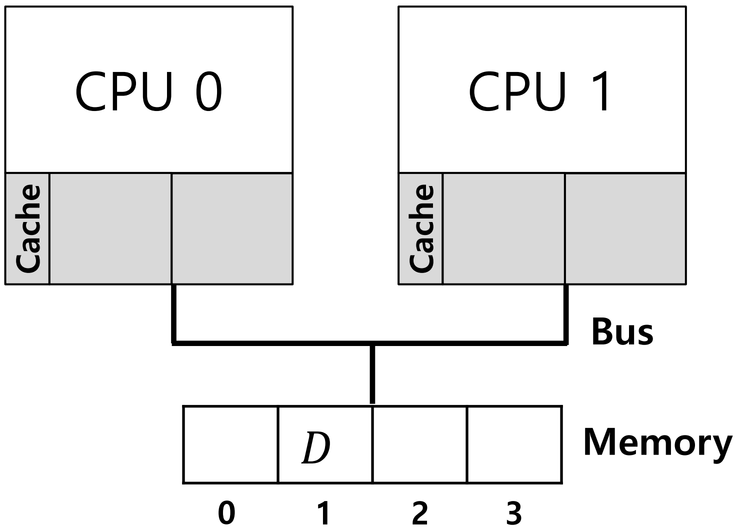
- Holds all of the data
- Access to main memory is slower than cache.

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

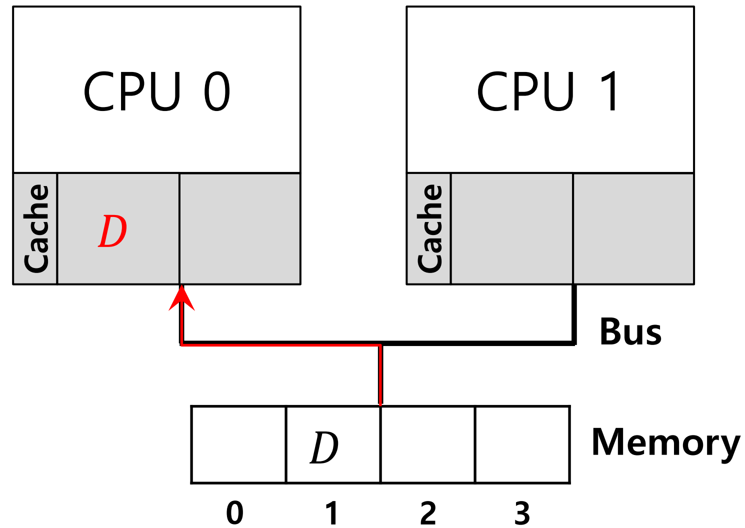
Cache coherence

- Consistency of shared resource data stored in multiple caches.

0. Two CPUs with caches sharing memory

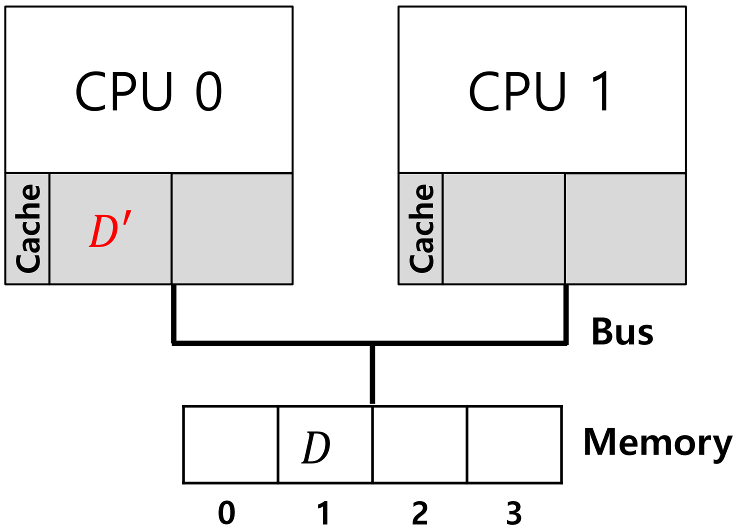


1. CPU0 reads a data at address 1.

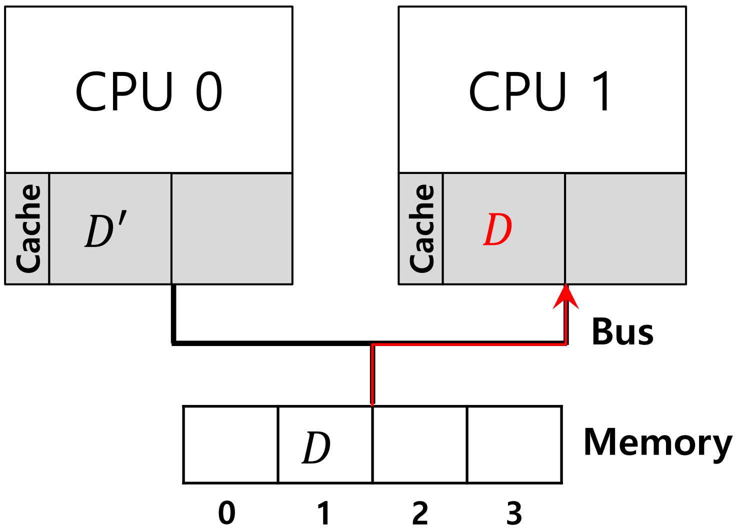


Cache coherence (Cont.)

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 <u>invalidate</u> its copy or <u>update</u> it.

Don’t forget synchronization

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

```
typedef struct __Node_t {
        int value;
        struct __Node_t *next;
} Node_t;

int List_Pop() {
        Node_t *tmp = head;           // remember old head ...
        int value = head->value;      // ... and its value
        head = head->next;            // advance head to next pointer
        free(tmp);                    // free old head
        return value;                 // return value at head
}
```

Simple List Delete Code

Don’t forget synchronization (Cont.)

- Solution

```
pthread_mtuex_t m;
typedef struct __Node_t {
        int value;
        struct __Node_t *next;
} Node_t;

int List_Pop() {
        lock(&m)
        Node_t *tmp = head;             // remember old head ...
        int value = head->value;        // ... and its value
        head = head->next;              // advance head to next pointer
        free(tmp);                      // free old head
        unlock(&m)
        return value;                   // return value at head
}
```

Simple List Delete Code with lock

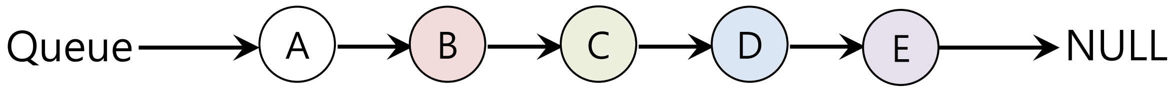
Cache Affinity

- 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)

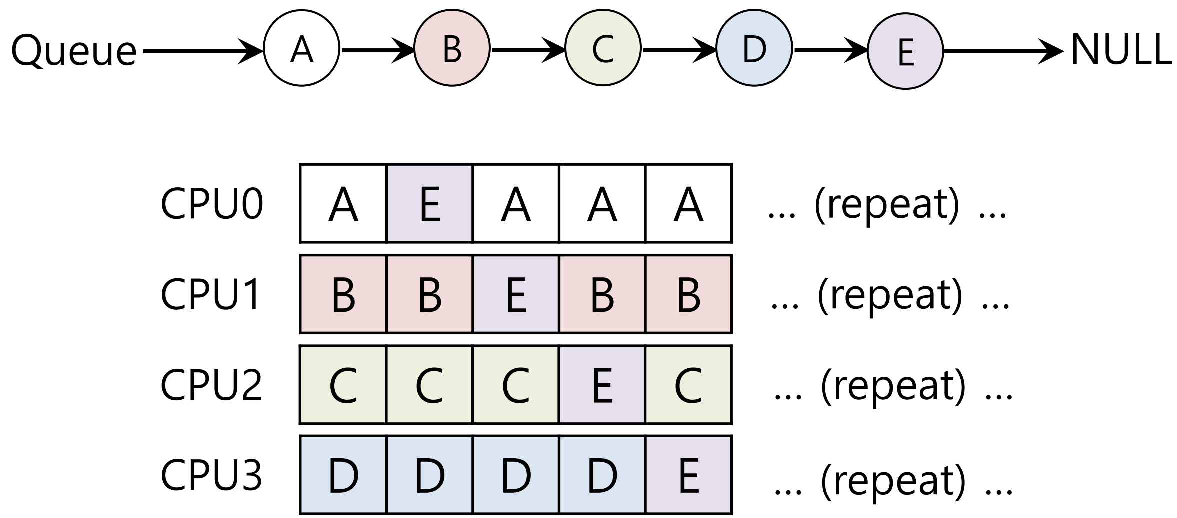
- Put all jobs that need to be scheduled into a single queue.
  - Each CPU simply picks the next job from the globally shared queue.
  - Cons:
    - Some form of **locking** have to be inserted → Lack of scalability
    - Cache affinity
    - Example:



    - Possible job scheduler across CPUs:

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

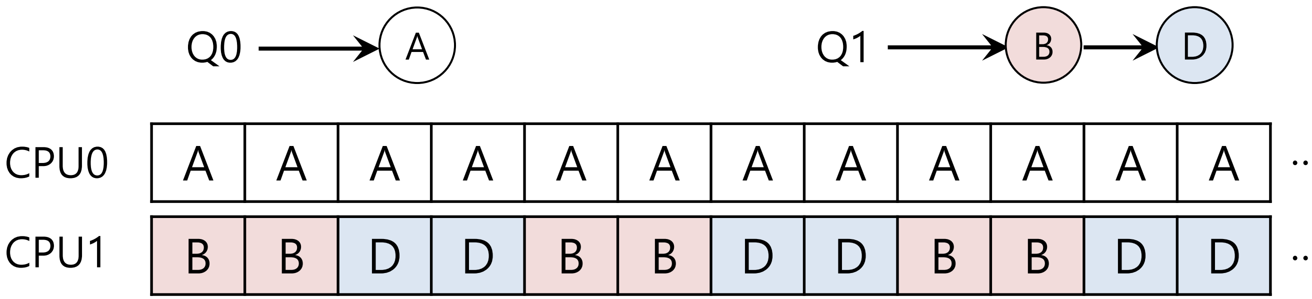
- With **round robin**, the system might produce a schedule that looks like this:

CPU0	A	A	C	C	A	A	C	C	A	A	C	C	…
CPU1	B	B	D	D	B	B	D	D	B	B	D	D	…

MQMS provides more scalability and cache affinity.

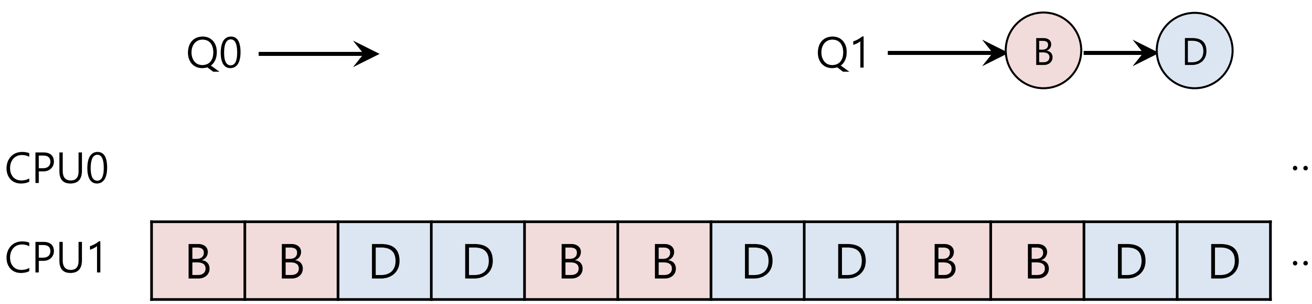
Load Imbalance issue of MQMS

- After job C in Q0 finishes:



A gets twice as much CPU as B and D.

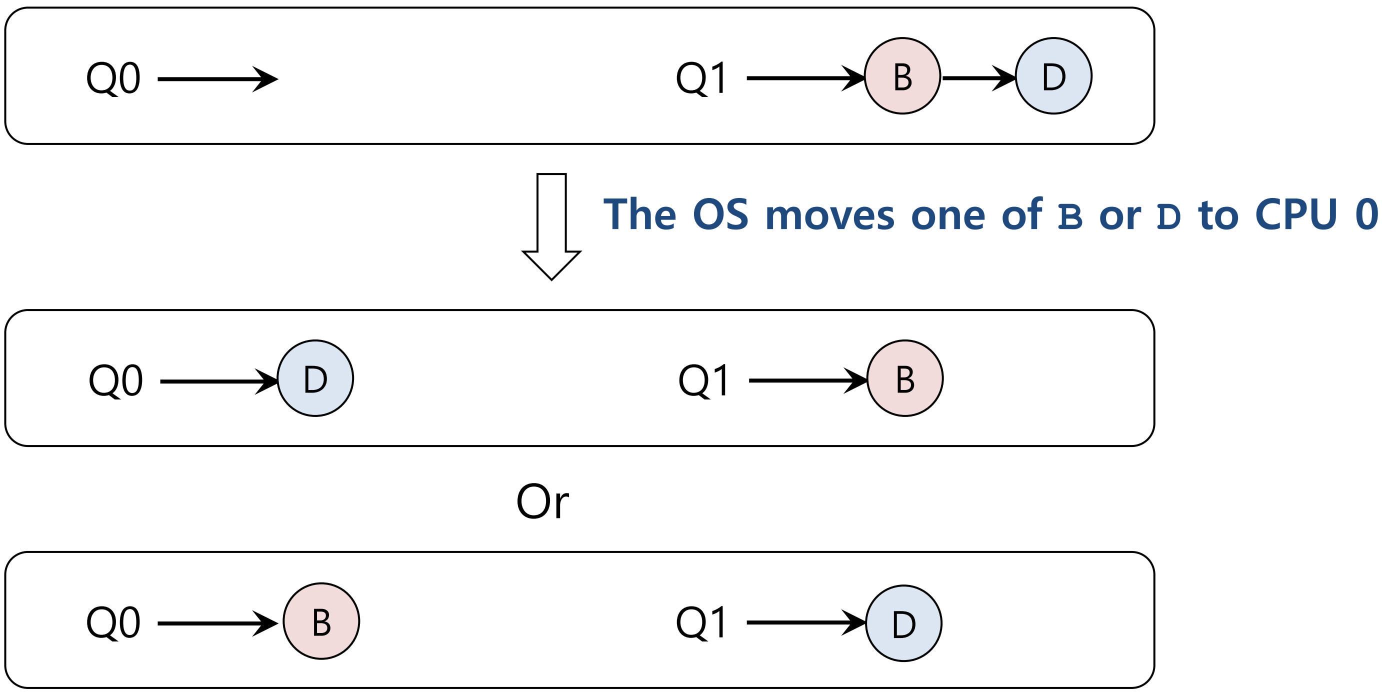
- After job A in Q0 finishes:



CPU0 will be left idle!

How to deal with load imbalance?

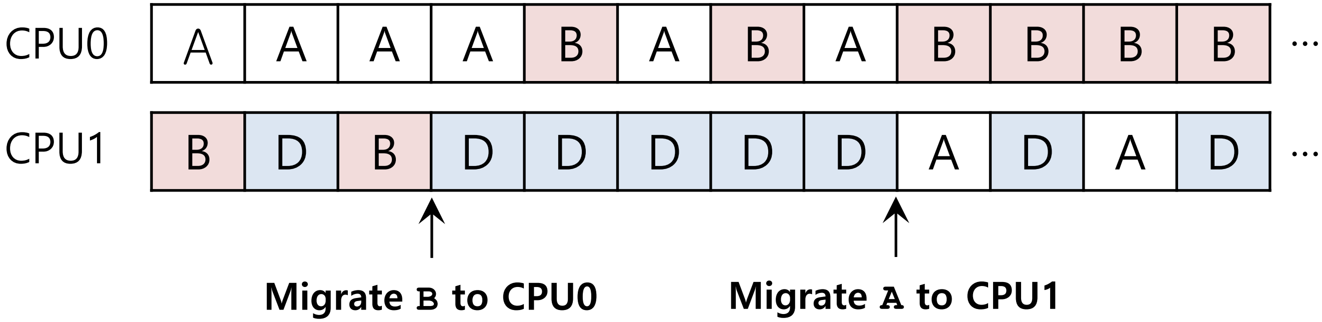
- The answer is to move jobs (**Migration**).
  - Example:



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 <u>low on jobs</u> is picked.
    - The source queue occasionally peeks at another target queue.
    - If the target queue is <u>more full than</u> 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

Linux Multiprocessor Schedulers (Cont.)

- BF Scheduler (BFS)
  - A single queue approach
  - Proportional-share
  - Based on Earliest Eligible Virtual Deadline First(EEVDF)

Summary

- CPU Scheduling
  - Scheduling metrics: Turnaround time, Response time; Fairness
  - Scheduling Strategy: FIFO, SJF, PSJF, RR, MLFQ
    - Simple: FIFO, SJF, PSJF, RR
    - MLFQ
    - Proportional Share
  - Multiprocessor scheduling
    - Cache coherence
    - Cache affinity
    - SQMS, MQMS, Load balance
- Next: Memory (Chapters 13, 15, 16, 17, 18, 19, 20, 21, 22, 23)