Unit-II Scheduling: Part-II

- The Multi-Level Feedback Queue, The Priority Boost, Attempt, Better Accounting,

- Multiprocessor Scheduling, Synchronization, Cache Affinity, Single-Queue Scheduling, Multi-Queue Scheduling, Linux Multiprocessor Schedulers

Multilevel Feedback Queue

- It tries to address two problems:
- First: It will optimize turnaround time.
- Second: It will minimize response time.
  - RR reduce response time but higher turn around time

Types of Multi Level Scheduling

- Multi Level Queue Scheduling
- Multi Level Feedback Queue Scheduling

Multi-Level Queue

- Processes are classified into different groups.
  - Like Interactive Processes and batch processes.
- These types of processes have different response time requirement (So need different scheduling algorithms)
- Interactive processes have high priority over batch processes.
- A multi-level queue scheduling algorithm partitions the ready queue into several separate queues.

MLQ Concept:

- Consider an example of simple multilevel queue scheduling algorithm with four queues
  - System Processes
  - Interactive processes
  - Batch processes
  - Application processes
- Each queue has priority over lower priority queue.
- No process in the batch queue could run unless the queues of system processes and interactive processes were empty.
- If interactive processes entered in the ready queue while batch process was running, the batch process will be pre-empted.

Example: MLQ

Process	Queue No	AT	BT	TAT	WT
P1	1	0	4	6	2
P2	1	0	3	7	4
P3	2	0	8	20	12
P4	1	10	5	5	0

- Q1 uses RR (TQ=2)
- Q2 uses FCFS
- Priority of Q1 is greater than Q2
- Queue No: Queues of the process.

Multi Level Queue Scheduling Disadvantages:

- Starvation:
  - Once the processes are assigned to queue, it can not switch between the queues.
    - If very few batch processes and very high system processes. Priority of system process is higher

- IDEA:
  - Separate processes with different CPU-burst characteristics.
  - If the process uses too much CPU time, it will be moved to the low priority queue.
  - Process that waits too long in low priority queue, may be moved to the high priority queue: it prevents starvation

- The key to MLFQ scheduling therefore lies is : how the scheduler sets priorities.

- Rather than giving a fixed priority to each job, MLFQ varies the priority of a job based on its observed behaviour.

  - Example:
    - For interactive system processes MLFQ will keep processes priority high
    - If a job uses the CPU intensively for long periods of time, MLFQ will reduce its priority.

  - In this way, MLFQ will try to learn about processes as they run, and thus use the history of the job to predict its future behaviour.

MLFQ

- It has number of distinct queues.
- Each queue has different priority.
- MLFQ uses the priority to decide which job to run at a given time.
  - Job with higher priority is chosen to run

MLFQ: Rules

- Two basic Rules:
  - Rule 1: If Priority(A) > Priority(B), A runs (B doesn’t).
  - Rule 2: If Priority(A) = Priority(B), A & B run in round-robin fashion using the time slice (quantum length) of the given queue.

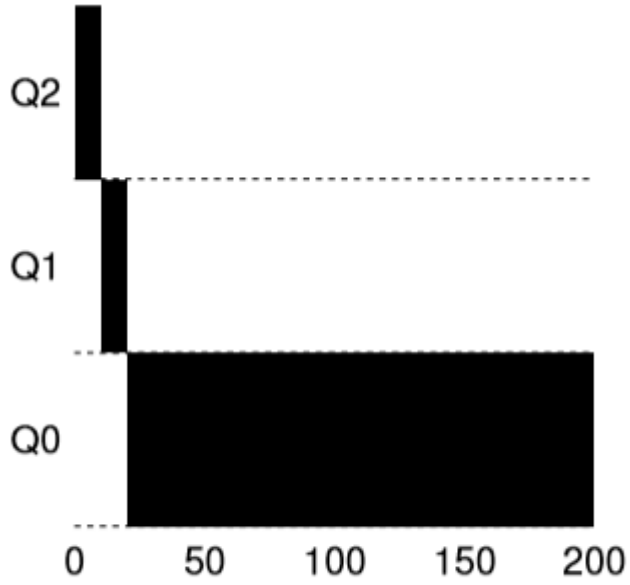
How to change Priority:

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

How to change Priority:

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.



Long-running Job Over Time

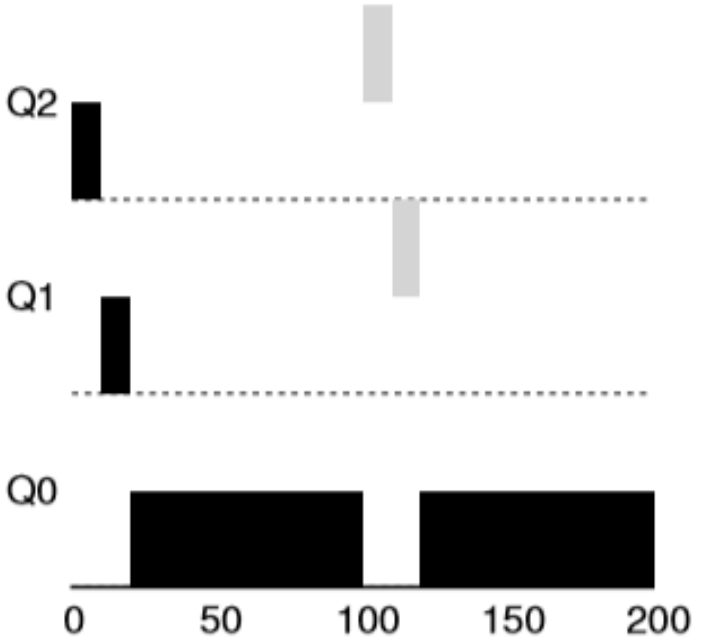
How to change Priority:

Example 2: Along came a short job

- How MLFQ tries to approximate SJF.
- In this example, there are two jobs:
  - A: long-running job (black),
  - B: short-running job (gray), arrives at T =100, running time 20ms
- Assume A has been running for some time, and then B arrives. What will happen?

- A is running a long in the lowest-priority queue.

- B is inserted into the highest queue; as its run-time is short, B completes before reaching the bottom queue.



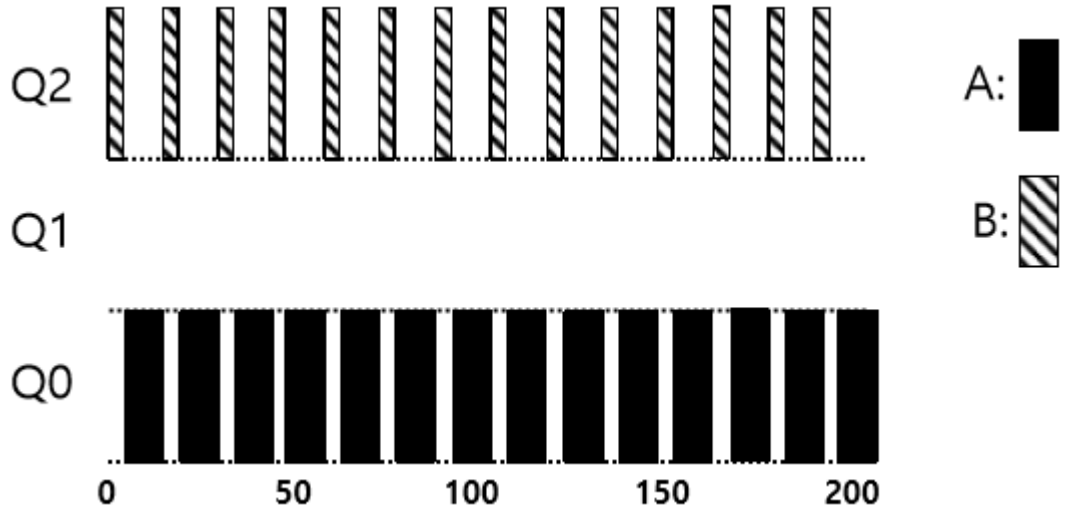
Along Came An Interactive Job

- According to Rule 4B:

- If there are two many jobs in interactive systems, it will consume all CPU time and long running job will never receive CPU-Starvation.

Better Accounting

- Assume:
  - 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
- 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).



The Priority Boost

- **Starvation: A process in the lower priority queue can suffer from starvation due to some short processes taking all the CPU time.**
- **The simple idea is to periodically boost the priority of all the jobs in system.**
- There are many ways to achieve this, like throw them all in the topmost queue; hence, a new rule:
  - Rule 5: After some time period S, move all the jobs in the system to the topmost queue.
- New rule solves two problems at once.
  - First, processes are guaranteed not to starve: by sitting in the top queue, a job will share the CPU with other high-priority jobs in a round-robin fashion, and thus eventually receive service.
  - Second, if a CPU-bound job has become interactive, the scheduler treats it properly once it has received the priority boost.

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 round-robin fashion using the time slice (quantum length) of the given queue.
- Rule 3: When a job enters the system, it is placed at the highest priority (the topmost queue).
- 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 period S, move all the jobs in the system to the topmost queue

MLFQ: Examples:

Process	Arrival Time	Burst Time
P1	0	53
P2	0	17
P3	0	68
P4	0	24

- Three Queues
- Priority of Queues are :
  - Q1 : Highest Priority
  - Q2:
  - Q3 :Lowest Priority
- Q1 Uses RR (TQ=17)
- Q2 Uses RR (TQ=25)
- Q3 uses FCFS

Find Average TAT and WT.

Process	Arrival Time	Burst Time	TAT	WT
P1	0	53	136	83
P2	0	17	34	17
P3	0	68	162	94
P4	0	24	125	101

- Average TAT=114.25
- Average WT=73.75

- **Example –**
  Consider a system which has a CPU bound process, which requires the burst time of 40 seconds. The multilevel Feed Back Queue scheduling algorithm is used and the queue time quantum ‘2’ seconds and in each level it is incremented by ‘5’ seconds. Then how many times the process will be interrupted and on which queue the process will terminate the execution?

- **Solution –**
  Process P needs 40 Seconds for total execution.
  At Queue 1 it is executed for 2 seconds and then interrupted and shifted to queue 2.
  At Queue 2 it is executed for 7 seconds and then interrupted and shifted to queue 3.
  At Queue 3 it is executed for 12 seconds and then interrupted and shifted to queue 4.
  At Queue 4 it is executed for 17 seconds and then interrupted and shifted to queue 5.
  At Queue 5 it executes for 2 seconds and then it completes.
  Hence the process is interrupted 4 times and completes on queue 5.

- MLFQ is interesting for the following reason:
  - instead of demanding a priori knowledge of the nature of a job, it observes the execution of a job and prioritizes it accordingly.
  - In this way, it manages to achieve the best of both worlds:
    - it can deliver excellent overall performance (similar to SJF/STCF) for short-running interactive jobs, and is fair and makes progress for long-running CPU-intensive workloads.
    - For this reason, many systems, including BSD UNIX derivatives, Solaris, and Windows NT and subsequent Windows operating systems use a form of MLFQ as their base scheduler.

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?

Multiprocessor Scheduling

HOW TO SCHEDULE JOBS ON MULTIPLE CPUS

- How should the OS schedule jobs on multiple CPUs?
- What new problems arise?
- Do the same old techniques work, or are new ideas required?

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.**

Processor Affinity/ CPU Pining/ Cache Affinity

- **Processor affinity**, or **CPU pinning** or "cache affinity", enables the binding and unbinding of a process or a thread to a CPU or a range of CPUs, so that the process or thread will execute only on the designated CPU or CPUs rather than any CPU

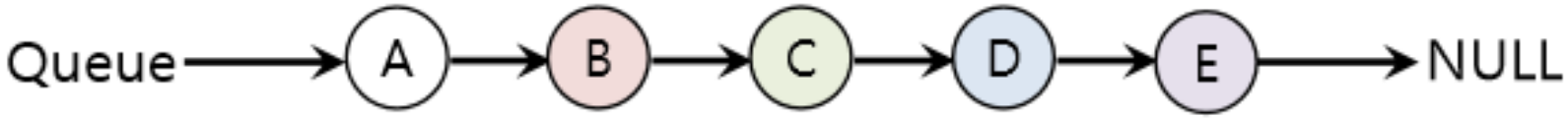
Single Queue Multiprocessor Scheduling (SQMS)

- Put all jobs that need to be scheduled into a single queue.
  - Each CPU simply picks the next job from the globally shared queue.
  - Disadvantages:
    - Lack of Scalability.
    - To ensure that the scheduler works correctly on multiple CPUs, developers will have to insert some form of locking into the code. (CPU 0 is accessing the queue, no other CPU can access it) – not good idea- scheduler need to be highly efficient.

      Reduces Performance
    - Cache affinity

Cache Affinity:

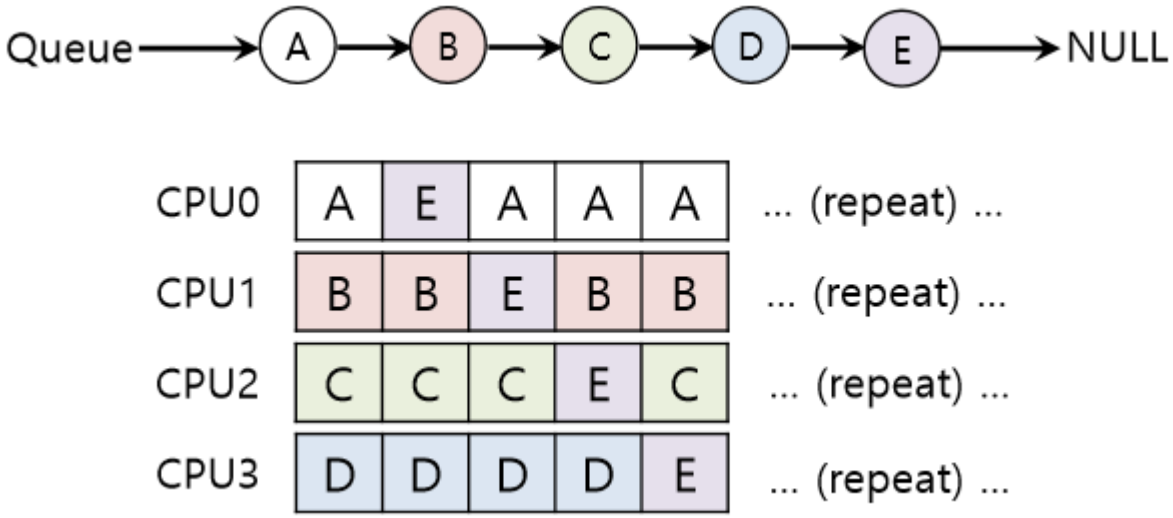
Example:



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) ...

- To handle this problem, most SQMS schedulers include some kind of affinity mechanisms, so that process will continue to run on the same CPU if possible



- 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

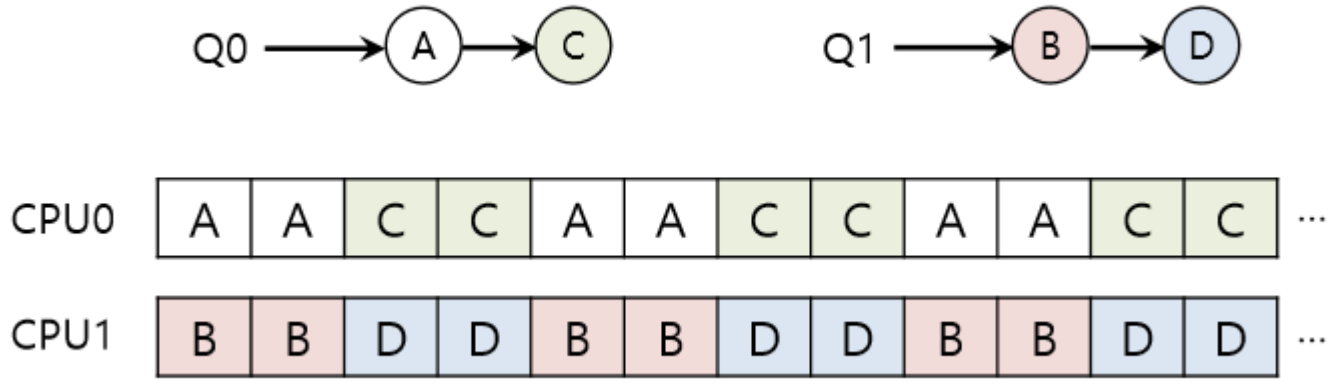
Multi-Queue Multiprocessor Scheduling (MQMS)

- 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, according to some heuristic (e.g., random, or picking one with fewer jobs than others)
  - Then it is scheduled essentially independently, thus avoiding the problems of information sharing and synchronization found in the single-queue approach

MQMS Example:

- Assume a system with two CPUs (labelled CPU 0 and CPU 1).

- Jobs in the system: A, B, C, and D.
- Given that each CPU has a scheduling queue.
- The OS has to decide into which queue to place each job. It might do something like this (with RR)



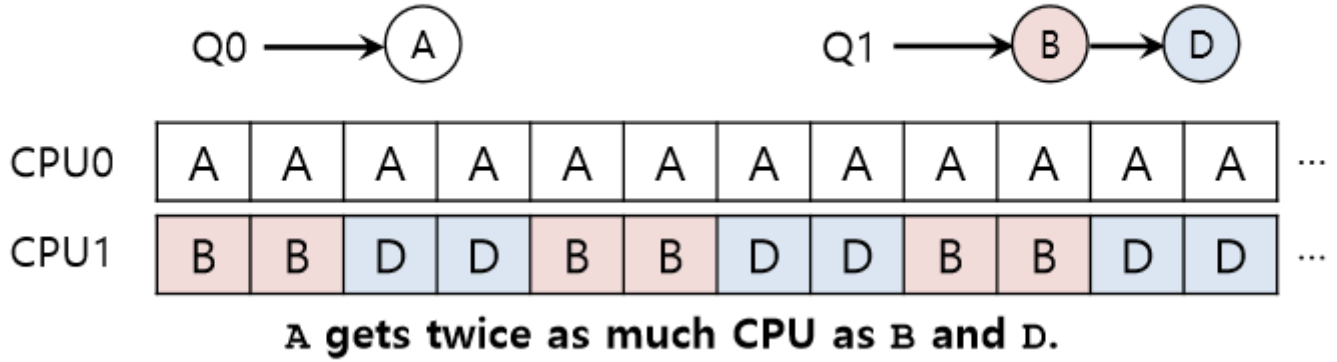
Advantages

- More scalable (As the number of CPUs grows, so too does the number of queues)

- Provides cache affinity.
  - jobs stay on the same CPU and thus reap the advantage of reusing cached contents therein.

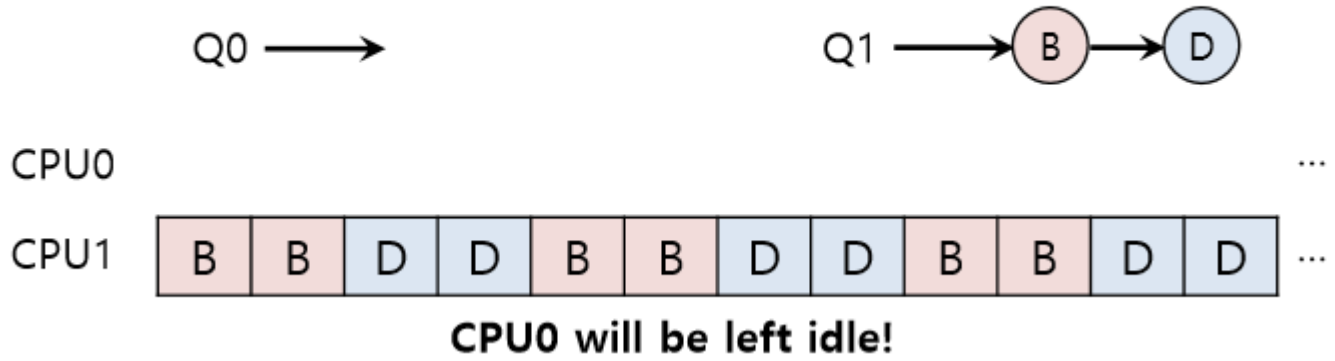
Disadvantages: Load Balancing

After job C in Q0 finishes:



A gets twice as much CPU as B and D.

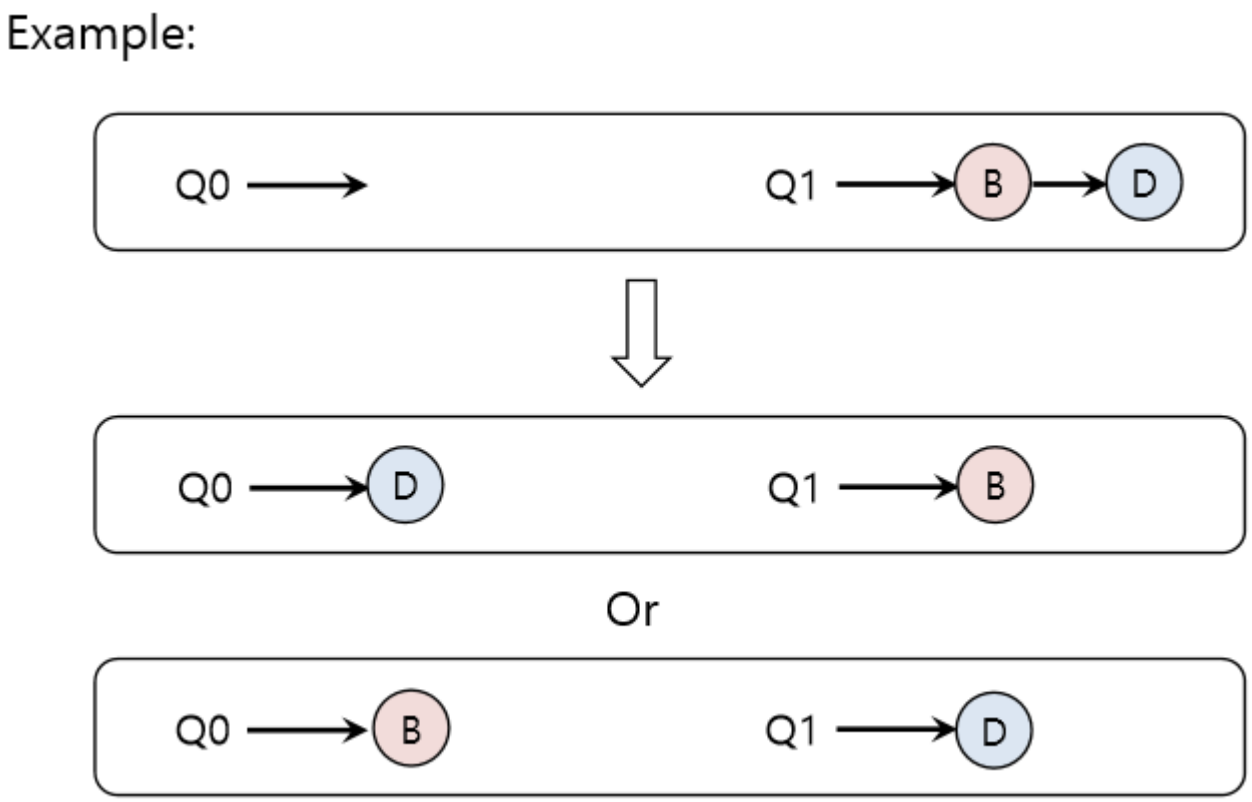
After job A in Q0 finishes:



CPU0 will be left idle!

How to deal with Load Imbalance?

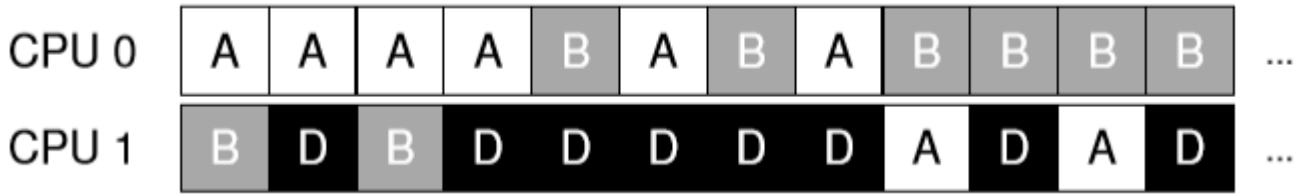
Migration of jobs:



- It is also complicated

- A more tricky case arises, where A was left alone on CPU 0 and B and D were alternating on CPU 1:

- In this case, a single migration does not solve the problem
- Continuous migration of one or more jobs is required.



Linux Multiprocessor Scheduler

- In Linux community, no common solution has approached to building a multiprocessor scheduler.
- Over time, three different solutions
  - O(1) Scheduler
  - The Completely Fair Scheduler (CFS)
  - BF Scheduler (BFS)
- O(1) and CFS uses multiple queues
- BFS uses single queue
- O(1) is a priority based scheduler (similar to MLFQ)