# Operating Systems

## CPU Scheduling

Carlos Alberto Llano R.

1 de octubre de 2015

# Contents

# Basic concepts

When a computer is multiprogrammed, it frequently has multiple processes or threads competing for the CPU at the same time. This situation occurs whenever two or more of them are simultaneously in the ready state.

In a single-processor system, only one process can run at a time.

Others must wait until the CPU is free and can be rescheduled.

The objective of multiprogramming is to have some process running at all times, to maximize CPU utilization.

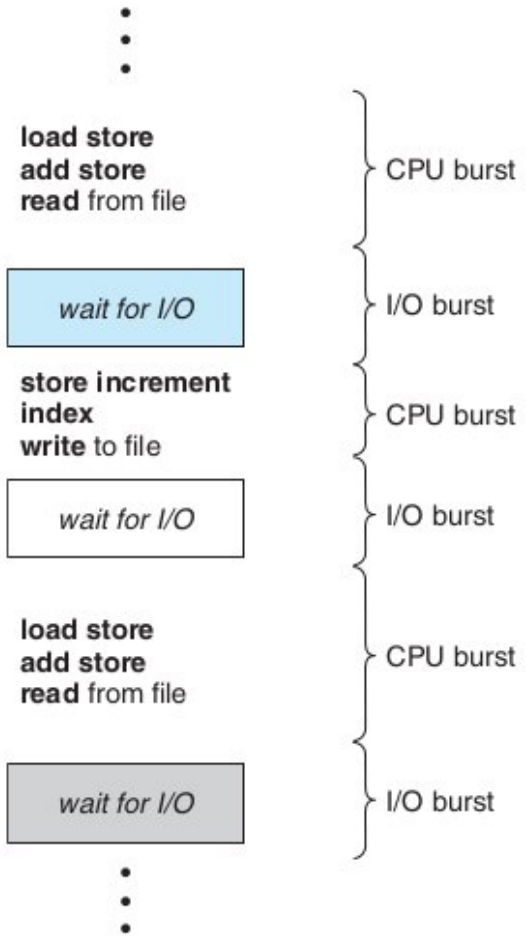
# Basic concepts

The key to multiprograming is scheduling.

The part of the operating system that makes the choice is called the scheduler, and the algorithm it uses is called the scheduling algorithm.

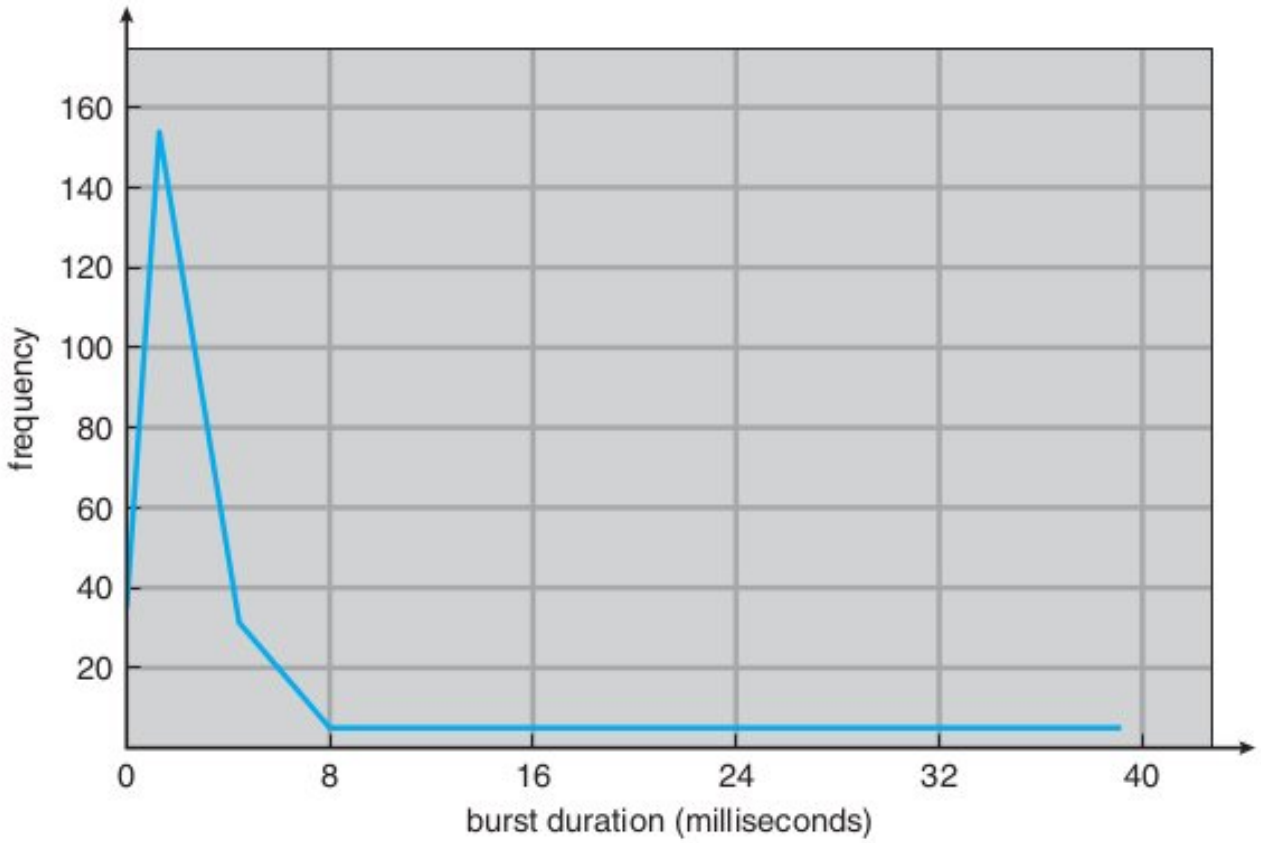
# CPU – I/O Burst Cycle

Process execution consists of a cycle of CPU execution and I/O wait.



# CPU – I/O Burst Cycle

Although they vary greatly from process to process and from computer to computer, they tend to have a frequency curve similar.



# CPU Scheduler

Whenever the CPU becomes idle, the operating system must select one of the processes in the ready queue to be executed.

The selection process is carried out by the short-term scheduler or CPU scheduler.

The scheduler selects a process from the processes in memory that are ready to execute and allocates the CPU to that process.

# When to Schedule

1. When a process switches from the running state to the waiting state (for example, as the result of an I/O request or an invocation of wait() for the termination of a child process)
2. When a process switches from the running state to the ready state (for example, when an interrupt occurs)
3. When a process switches from the waiting state to the ready state (for example, at completion of I/O)
4. When a process terminates

# Scheduling algorithms

Scheduling algorithms can be divided into two categories with respect to how they deal with clock interrupts.

A nonpreemptive or cooperative scheduling algorithm (1 and 4)

Once the CPU has been allocated to a process, the process keeps the CPU until it releases the CPU either by terminating or by switching to the waiting state. This scheduling method was used by Microsoft Windows 3.x.

# Scheduling algorithms

**a preemptive scheduling algorithm** picks a process and lets it run for a maximum of some fixed time. Windows 95 introduced preemptive scheduling, and all subsequent versions of Windows operating systems have used preemptive scheduling. The Mac OS X operating system for the Macintosh also uses preemptive scheduling

# Scheduling Criteria - CPU Utilization

**CPU utilization** We want to keep the CPU as busy as possible. Conceptually, CPU utilization can range from 0 to 100 percent. In a real system, it should range from 40 percent (for a lightly loaded system) to 90 percent (for a heavily loaded system).

# Scheduling Criteria - Throughput

**Throughput** One measure of work is the number of processes that are completed per time unit, called throughput. For long processes, this rate may be one process per hour; for short transactions, it may be ten processes per second.

# Scheduling Criteria - Turnaround time

**Turnaround time** Turnaround time is the sum of the periods spent waiting to get into memory, waiting in the ready queue, executing on the CPU, and doing I/O.

# Scheduling Criteria - Waiting time

**Waiting time** Waiting time is the sum of the periods spent waiting in the ready queue.

# Scheduling Criteria - Response time

**Response time** is the time it takes to start responding, not the time it takes to output the response. The turnaround time is generally limited by the speed of the output device.

# Workload Assumptions

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

# Workload Assumptions - turnaround time

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

$$T_{arrival} = 0$$

$$T_{turnaround} = T_{completion}$$

# First In, First Out (FIFO)

The most basic algorithm we can implement is known as First In, First Out (FIFO) scheduling or sometimes First Come, First Served (FCFS).

The implementation of the FCFS policy is easily managed with a FIFO queue.

# First In, First Out (FIFO)

When a process enters the ready queue, its PCB is linked onto the tail of the queue.

When the CPU is free, it is allocated to the process at the head of the queue.

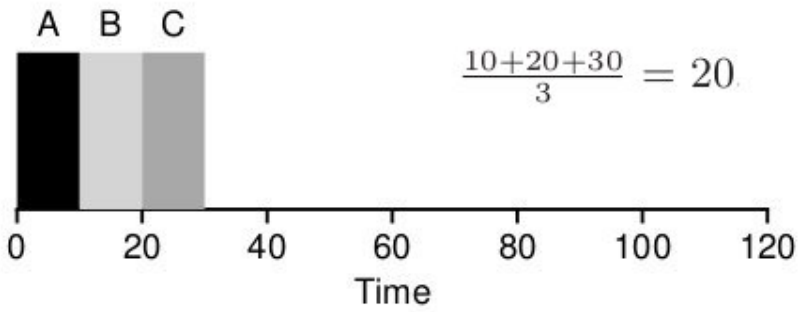
The running process is then removed from the queue.

# First In, First Out (FIFO)

Imagine three jobs arrive in the system, A, B, and C, at roughly the same time.

A arrived just a hair before B which arrived just a hair before C.

Assume also that each job runs for 10 seconds.

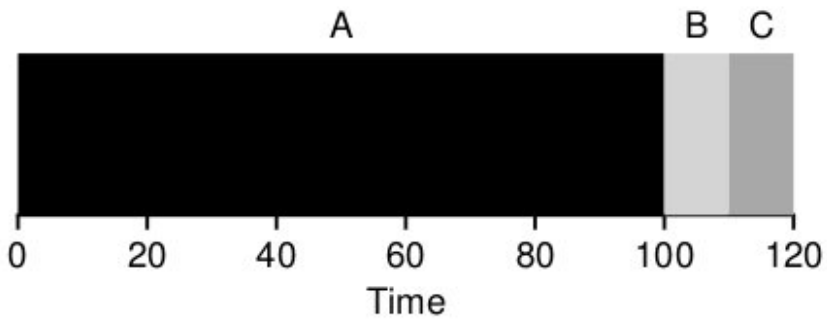


# First In, First Out (FIFO) - Convoy effect

What kind of workload could you construct to make FIFO perform poorly?

# First In, First Out (FIFO) - Convoy effect

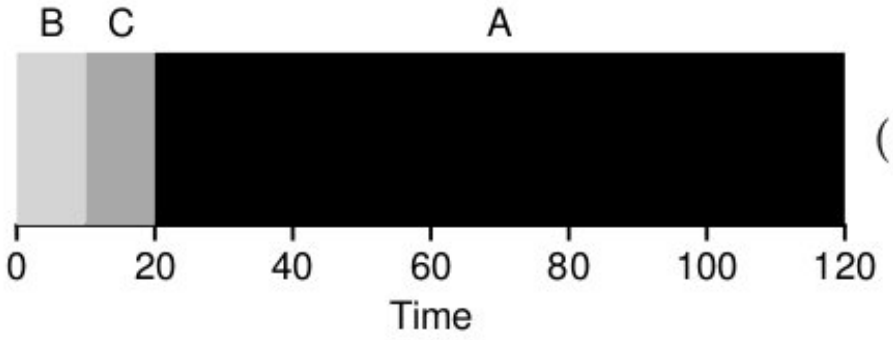
What kind of workload could you construct to make FIFO perform poorly?




$(\frac{100+110+120}{3} = 110)$

# Shortest-Job-First Scheduling (SJF)

The name should be easy to remember because it describes the policy quite completely: it runs the shortest job first, then the next shortest, and so on.




$(\frac{10+20+120}{3} = 50)$

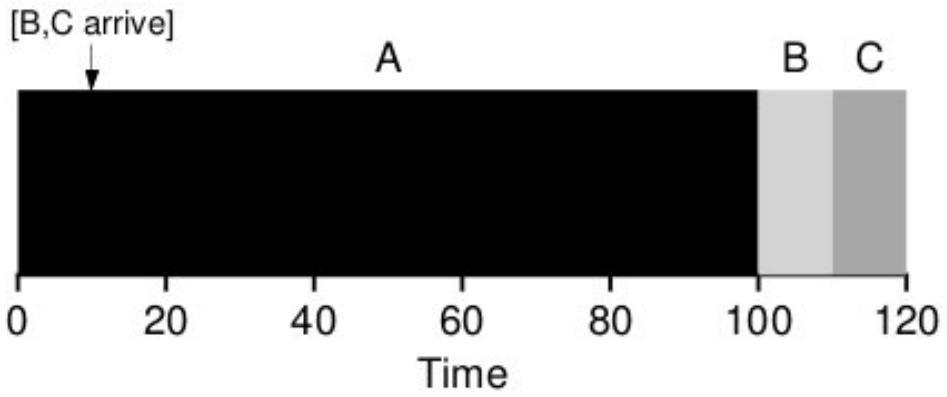
# Shortest-Job-First Scheduling (SJF)

Now assume that jobs can arrive at any time instead of all at once.

What problems does this lead to?

Pause to think...

# Shortest-Job-First Scheduling (SJF)




$$\left(\frac{100+(110-10)+(120-10)}{3}\right)$$

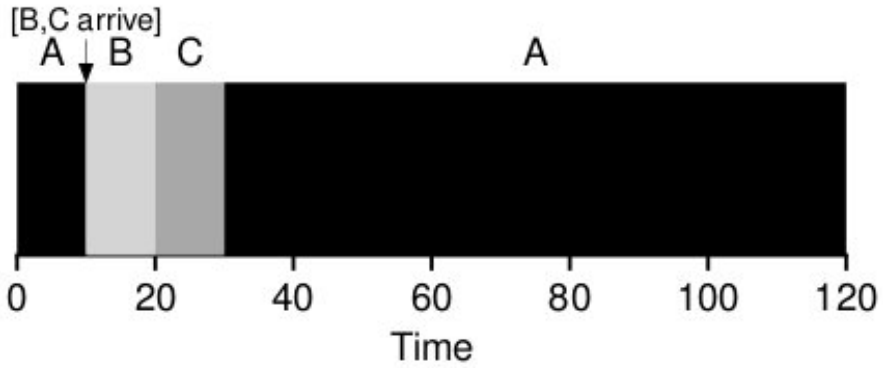
103.33 seconds

Non-preemptive scheduler.

# Shortest Time-to-Completion First (STCF)

or Preemptive Shortest Job First (PSJF) scheduler.

We need to relax assumption 3 (that jobs must run to completion)




$$\left(\frac{(120-0)+(20-10)+(30-10)}{3}\right)$$

50 seconds

# Priority Scheduling

The SJF algorithm is a special case of the general priority scheduling algorithm.

A priority is associated with each process, and the CPU is allocated to the process with the highest priority. Equal-priority processes are scheduled in FCFS order.

Priorities are generally indicated by some fixed range of numbers, such as 0 to 7 or 0 to 4,095. There is no general agreement on whether 0 is the highest or lowest priority.

# Priority Scheduling

| Process | Burst Time | Priority |
|---|---|---|
| $P_1$ | 10 | 3 |
| $P_2$ | 1 | 1 |
| $P_3$ | 2 | 4 |
| $P_4$ | 1 | 5 |
| $P_5$ | 5 | 2 |


8.2 milliseconds.

# Priority Scheduling

Priorities can be defined either internally or externally.

Internally defined priorities use some measurable quantity or quantities to compute the priority of a process. For example:

- time limits
- memory requirements
- the number of open files
- the ratio of average I/O

# Priority Scheduling

External priorities are set by criteria outside the operating system. For example:

- the importance of the process
- the type and amount of funds being paid for computer use
- the department sponsoring the work
- and other, often political, factors

# Priority Scheduling

A major problem with priority scheduling algorithms is indefinite blocking, or starvation.

A process that is ready to run but waiting for the CPU can be considered blocked.

A priority scheduling algorithm can leave some low priority processes waiting indefinitely.

Rumor has it that when they shut down the IBM 7094 at MIT in 1973, they found a low-priority process that had been submitted in 1967 and had not yet been run.

# Priority Scheduling - Solution

Pause to think...

# Priority Scheduling - Solution

Pause to think... aging

Aging involves gradually increasing the priority of processes that wait in the system for a long time. For example:

if priorities range from 127 (low) to 0 (high), we could increase the priority of a waiting process by 1 every 15 minutes. Eventually, even a process with an initial priority of 127 would have the highest priority in the system and would be executed.

# A New Metric: Response Time

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.

Response time is defined as the time from when the job arrives in a system to the first time it is scheduled.

# Round-Robin Scheduling

The round-robin (RR) scheduling algorithm is designed especially for time sharing systems.

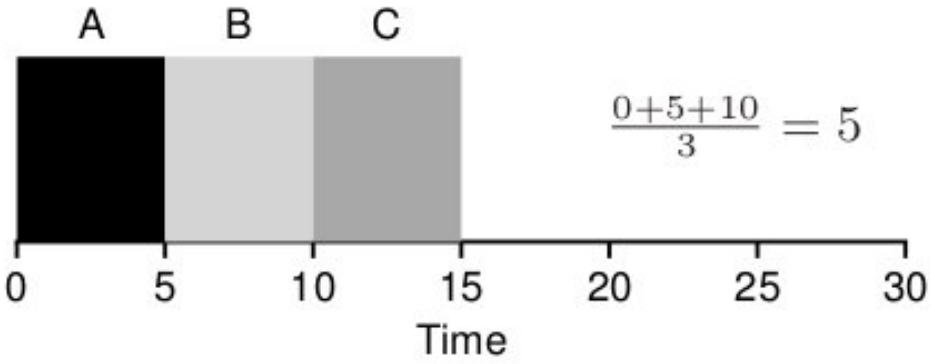
It is similar to FCFS scheduling, but preemption is added to enable the system to switch between processes.

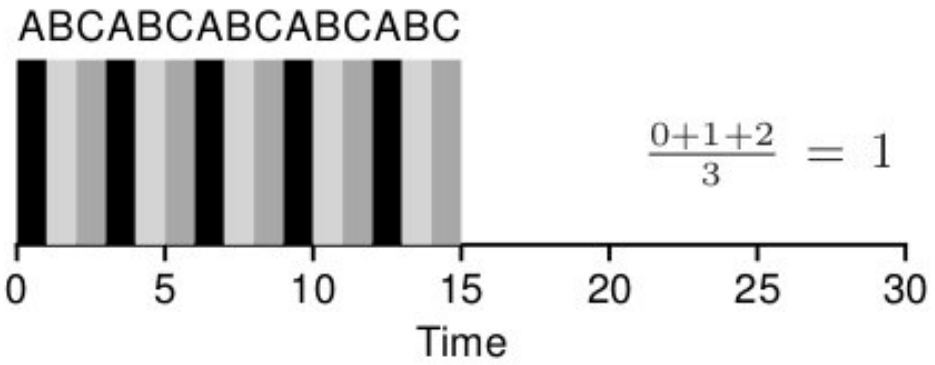
A small unit of time, called a time quantum or time slice (10 to 100 milliseconds in length).

the time slice must be a multiple of the timer-interrupt period, if the timer interrupts every 10 milliseconds, the time slice could be 10, 20, or any other multiple of 10 ms.

# Round-Robin Scheduling - Example

Assume three jobs A, B, and C arrive at the same time in the system, and that they each wish to run for 5 seconds.



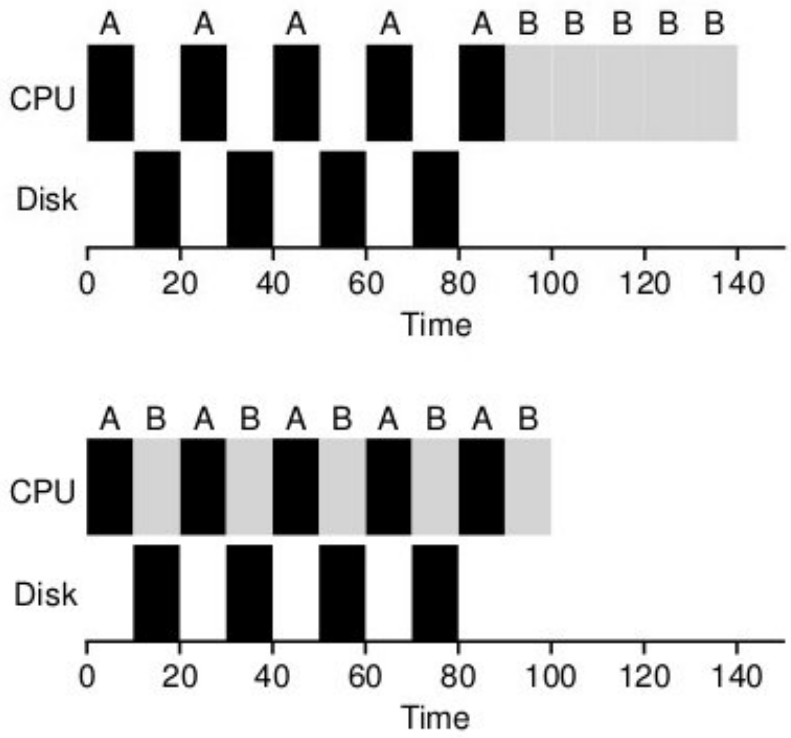
# Round-Robin Scheduling time slice

The length of the time slice is critical for RR. 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.

# Incorporating I/O - Overlap

Assume we have two jobs, A and B, which each need 50 ms of CPU time.

A runs for 10 ms and then issues an I/O request (I/O = 10ms), B simply uses the CPU for 50 ms and performs no I/O.



# The Multi-Level Feedback Queue

The fundamental problem MLFQ tries to address:

- Optimize turnaround time
- To make a system feel responsive to interactive users

Round Robin reduce response time but terrible for turnaround time.

*How can we design a scheduler that both minimizes response time for interactive jobs while also minimizing turnaround time without a priori knowledge of job length?*

**The multi-level feedback queue is an example of a system that learns from the past to predict the future.**

# MLFQ: Basic Rules

MLFQ:

- has a number of distinct queues, each assigned a different priority level.
- uses priorities to decide which job should run at a given time.
- more than one job may be on a given queue, and thus have the same priority then Round Robin.
- the key is how the scheduler sets priorities.
- varies the priority of a job based on its observed behavior.
  - If a job repeatedly relinquishes the CPU while waiting for I/O then will keep its priority high
  - If a job use the CPU intensively for long periods of time then will reduce its priority

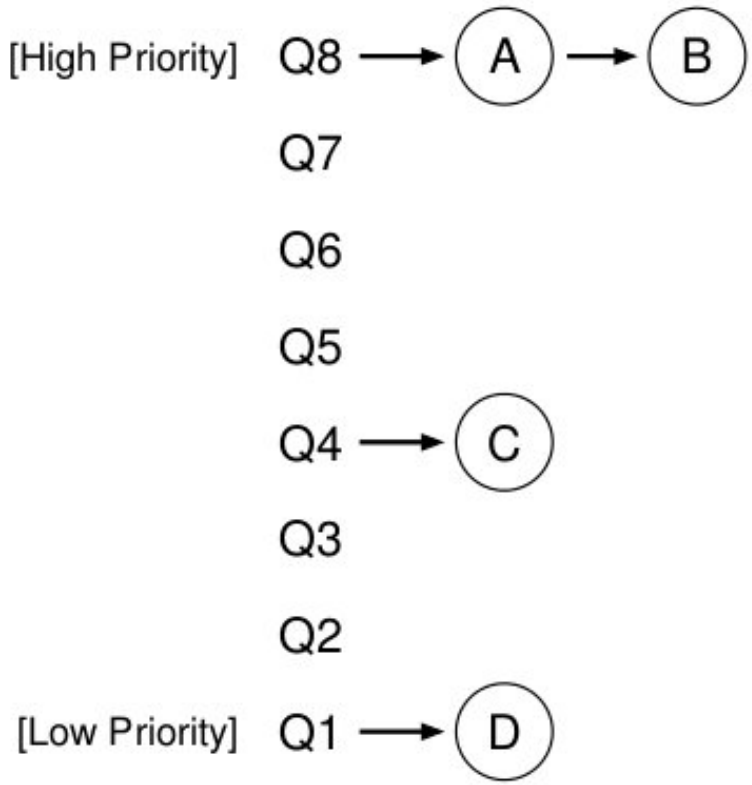
# MLFQ: Basic Rules

MLFQ will try to learn about processes as they run, and thus use the history of the job to predict its future behavior.

The first two basic rules for MLFQ:

- Rule 1: If Priority(A) $>$ Priority(B), A runs (B doesn't).
- Rule 2: If Priority(A) $=$ Priority(B), A & B run in RR.

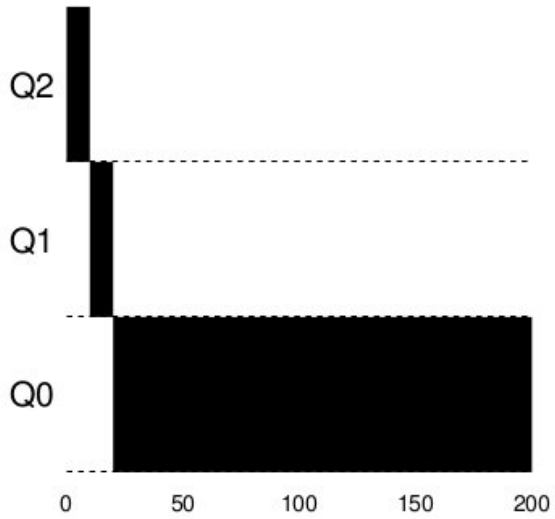
# MLFQ: Basic Rules



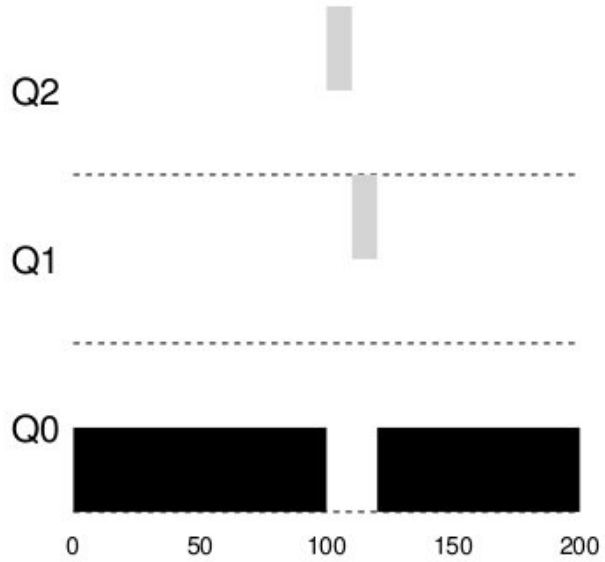
# MLFQ - How To Change Priority - Rules

- 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.

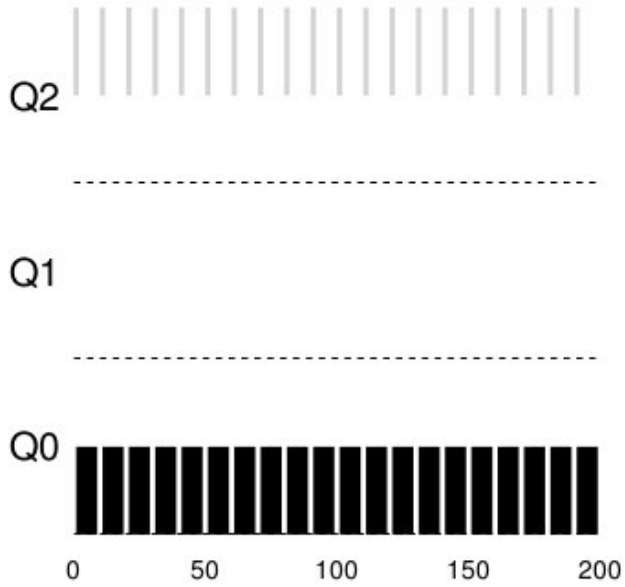
# MLFQ - How To Change Priority - Example 1



# MLFQ - How To Change Priority - Example 2



# MLFQ - How To Change Priority - Example 3



# Problems With the Current MLFQ

The approach we have developed thus far contains serious flaws.

Can you think of any?

# Problems With the Current MLFQ

The approach we have developed thus far contains serious flaws.

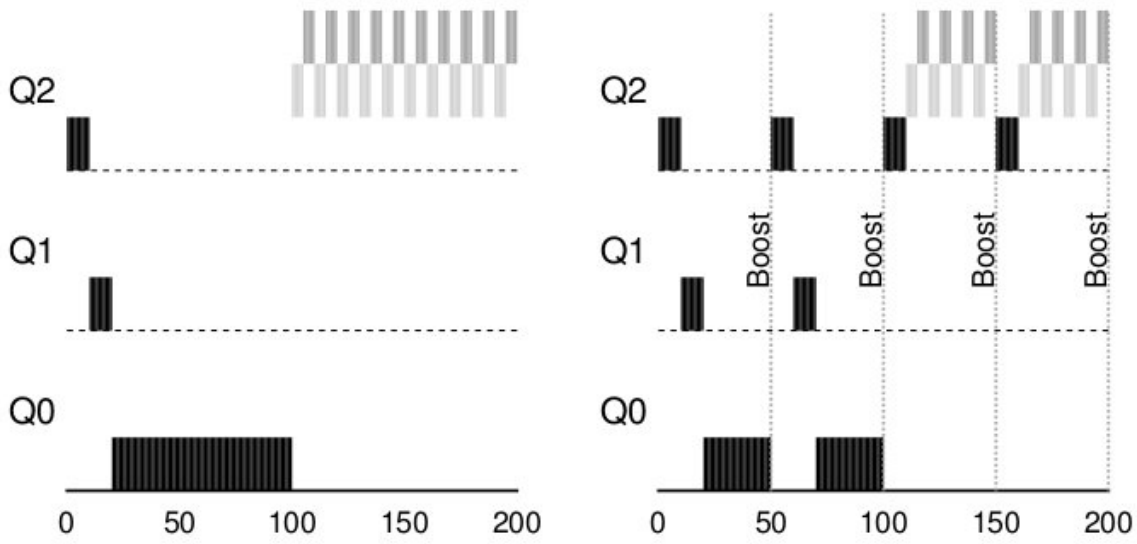
Can you think of any?

- 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.
- before the time slice is over, issue an I/O operation (to some file you don't care about) and thus relinquish the CPU; doing so allows you to remain in the same queue, and thus gain a higher percentage of CPU time.

# MLFQ - Rule5

throw them all in the topmost queue:

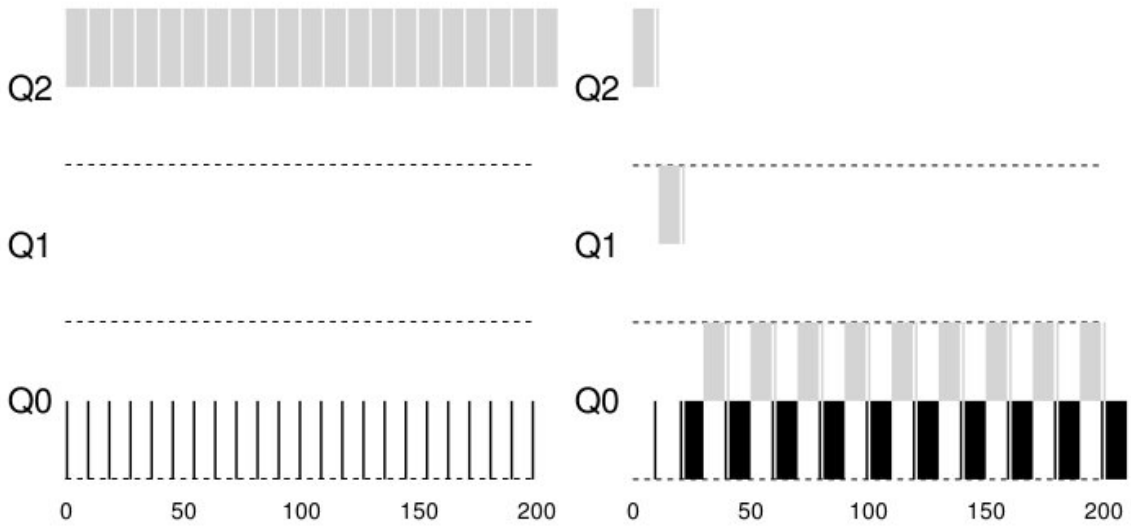
- Rule 5: After some time period S, move all the jobs in the system to the topmost queue.



# MLFQ - Rule4

Additionally other solution that is added here is to perform better accounting of CPU time at each level of the MLFQ.

- 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).



# Tunning MLFQ - Big question

- 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?

There are no easy answers to these questions.

# Tunning MLFQ - Solaris

- Solaris is particularly easy to configure
- set of tables that determine exactly how the priority of a process is altered through out its lifetime.
- how long each time slice is
- how often to boost the priority of a job
- Default values for the table are 60 queues,
- time-slice lengths from 20 milliseconds (highest priority) to a few hundred milliseconds (lowest),
- priorities boosted around every 1 second or so.

# Tunning MLFQ - FreeBSD

uses a formula to calculate the current priority level of a job, basing it on how much CPU the process has used.

# Lottery scheduling

The basic idea is to give processes lottery tickets for various system resources, such as CPU time.

Processes that should run more often should be given more chances to win the lottery.

Whenever a scheduling decision has to be made, a lottery ticket is chosen at random, and the process holding that ticket gets the resource.

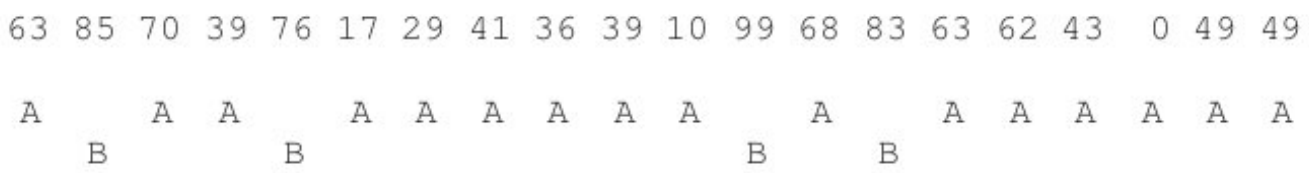
When applied to CPU scheduling, the system might hold a lottery 50 times a second, with each winner getting 20 msec of CPU time as a prize.

# Lottery scheduling - Example

Two processes, A and B, and further that A has 75 tickets while B has only 25. Thus, what we would like is forA to receive 75 % of the CPU and B the remaining 25 %.

The scheduler must know how many total tickets there are (in our example, there are 100).

Assuming A holds tickets 0 through 74 and B 75 through 99, the winning ticket simply determines whether A or B runs. The scheduler then loads the state of that winning process and runs it.



# Lottery scheduling - Ticket Mechanisms

**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.

# Lottery scheduling - Ticket Mechanisms

Example:

- Assume users A and B have each been given 100 tickets.
- User A is running two jobs, A1 and A2, and gives them each 500 tickets (out of 1000 total) in User A's own currency.
- User B is running only 1 job and gives it 10 tickets (out of 10 total).
- The system will convert A1's and A2's allocation from 500 each in A's currency to 50 each in the global currency.
- The lottery will then be held over the global ticket currency (200 total) to determine which job runs.

# Lottery scheduling - Ticket Mechanisms

```
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)
```

# Lottery scheduling - Ticket Mechanisms

**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

# Lottery scheduling - Ticket Mechanisms

```
// 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...
```

# Lottery scheduling - Implementation

You need is a good random number generator to pick the winning ticket, a data structure to track the processes of the system (e.g., a list), and the total number of tickets.

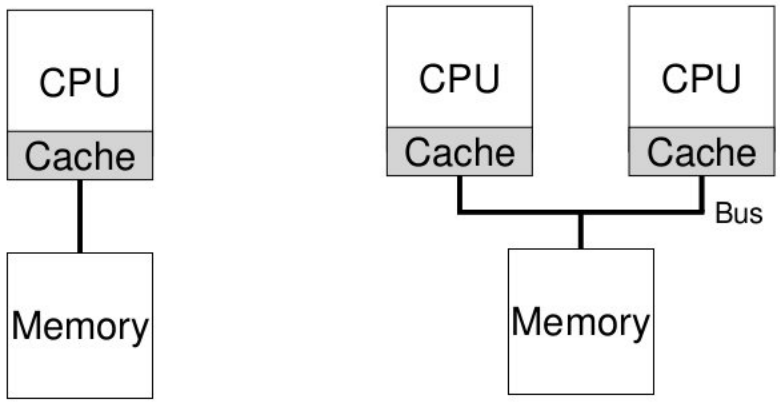
To make this process most efficient, it might generally be best to organize the list in sorted order, from the highest number of tickets to the lowest.

# Lottery scheduling - How To Assign Tickets?

The system behaves is strongly dependent on how tickets are allocated.

The "ticket-assignment"problem remains open.

# Multiprocessor Architecture




Cache coherence is the consistency of shared resource data that ends up stored in multiple local caches.

bus snooping each cache pays attention to memory updates

cache affinity to run it on the same CPU

# Single-Queue Scheduling - SQMS

The most basic approach is to simply reuse the basic framework for single processor scheduling, by putting all jobs that need to be scheduled into a single queue.

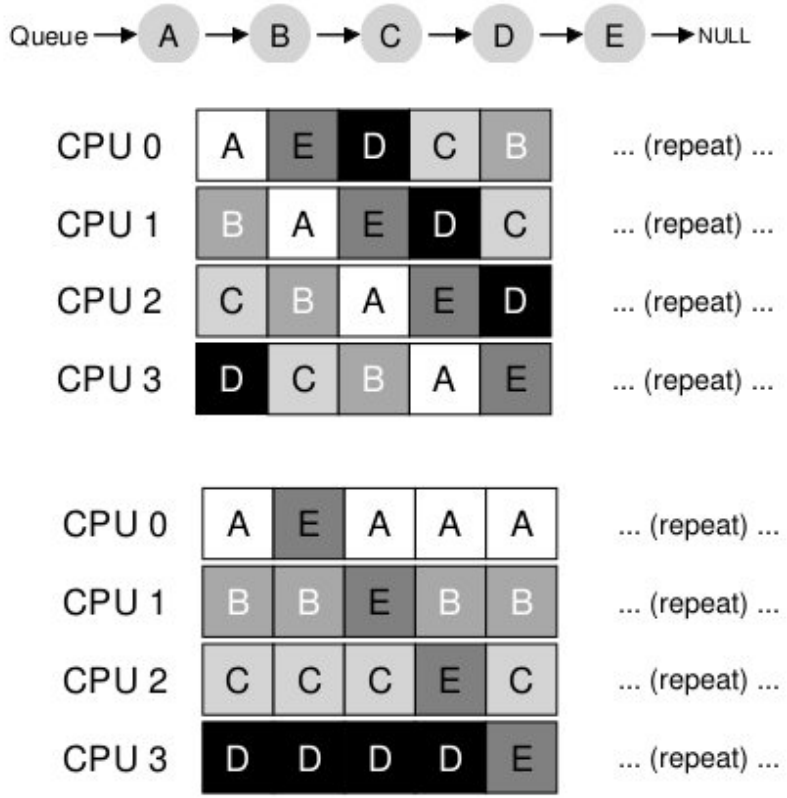
it does not require much work to take an existing policy that picks the best job to run next and adapt it to work on more than one CPU

# Single-Queue Scheduling - SQMS - Issues

Issues:

- To ensure the scheduler works correctly on multiple CPUs, the developers will have inserted some form of locking into the code
- cache affinity

# Single-Queue Scheduling - SQMS - Issues



# Multi-Queue Scheduling - MQMS

MQMS consists of multiple scheduling queues.

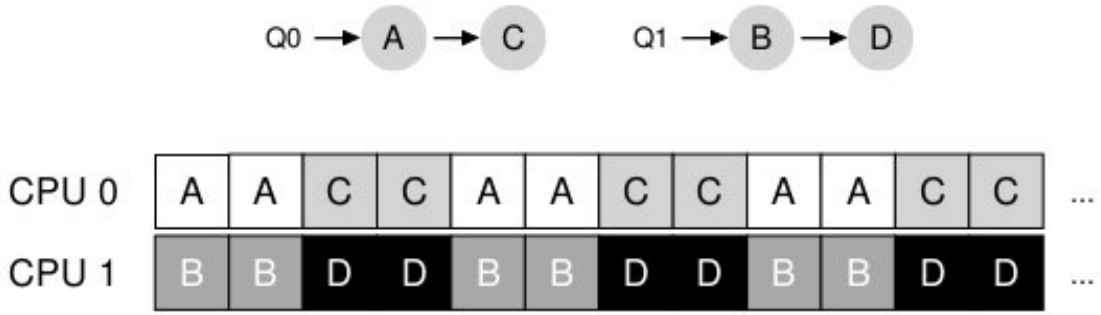
Each queue will likely follow a particular scheduling discipline, such as round robin, though of course any algorithm can be used.

When a job enters the system, it is placed on exactly one scheduling queue, according to some heuristic.

# Multi-Queue Scheduling - MQMS

Assume we have a system where there are just two CPUs (labeled CPU 0 and CPU 1), and some number of jobs enter the system:

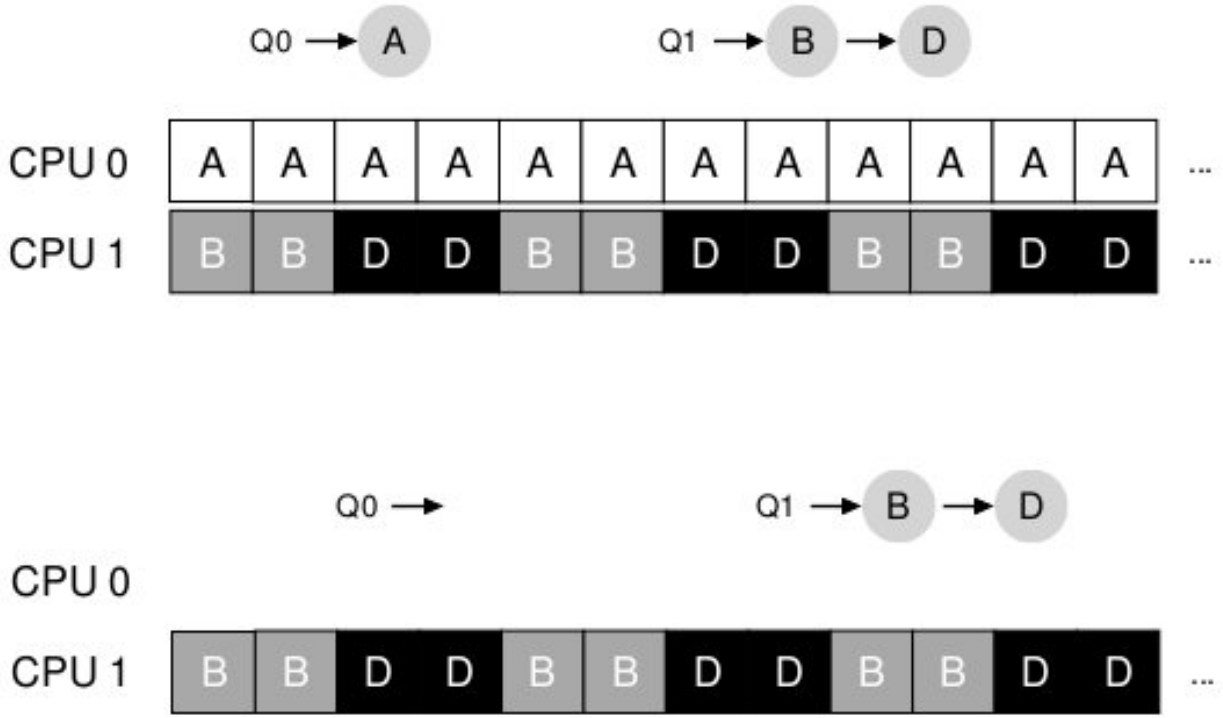
A, B, C, and D for example.




more scalable, cache contention, intrinsically provides cache affinity.

# Multi-Queue Scheduling - MQMS - Issues

load imbalance:



# Multi-Queue Scheduling - MQMS - Migration

To use a technique known as work stealing

A work-stealing approach, a (source) queue that is low on jobs will occasionally peek at another (target) queue, to see how full it is.

If the target queue is (notably) more full than the source queue, the source will "steal.ºne or more jobs from the target to help balance load.

# Linux Multiprocessor Schedulers

In the Linux community, no common solution has approached to building a multiprocessor scheduler.

Three different schedulers:

- the O(1) scheduler, (uses multiple queues and priority-based)
- the Completely Fair Scheduler (CFS), (uses multiple queues)
- the BF Scheduler (BFS)2 (uses a single queue)