RUTGERS UNIVERSITY
Computer Sciences Department

CS 416 Operating Systems Design

Sudarsun Kannan

CPU Virtualization: Scheduling (continued…)

Questions answered in this lecture:

How I/O Scheduling works?

What are different scheduling policies, such as:

MLFQ?

What type of workload performs well with each scheduler?

Disclaimer: Materials reused, modified from OSTEP book and lectures of Prof. Andrea and Remzi Arpaci-Dusseau and Prof. Yojip Won

Announcements

- Reading: Today cover Chapters 8-10

Preemptive Scheduling

Prev schedulers:

- FIFO and SJF are non-preemptive
- Only schedule new job when previous job voluntarily relinquishes CPU (performs I/O or exits)

New scheduler:

- Preemptive: Potentially schedule different job at any point by taking CPU away from running job
- STCF (Shortest Time-to-Completion First)
- Always run job that will complete the quickest

Scheduling Basics

Workloads:

arrival_time

run_time

Schedulers:

FIFO

SJF

STCF

RR

Metrics:

turnaround_time

response_time

Workload Assumptions

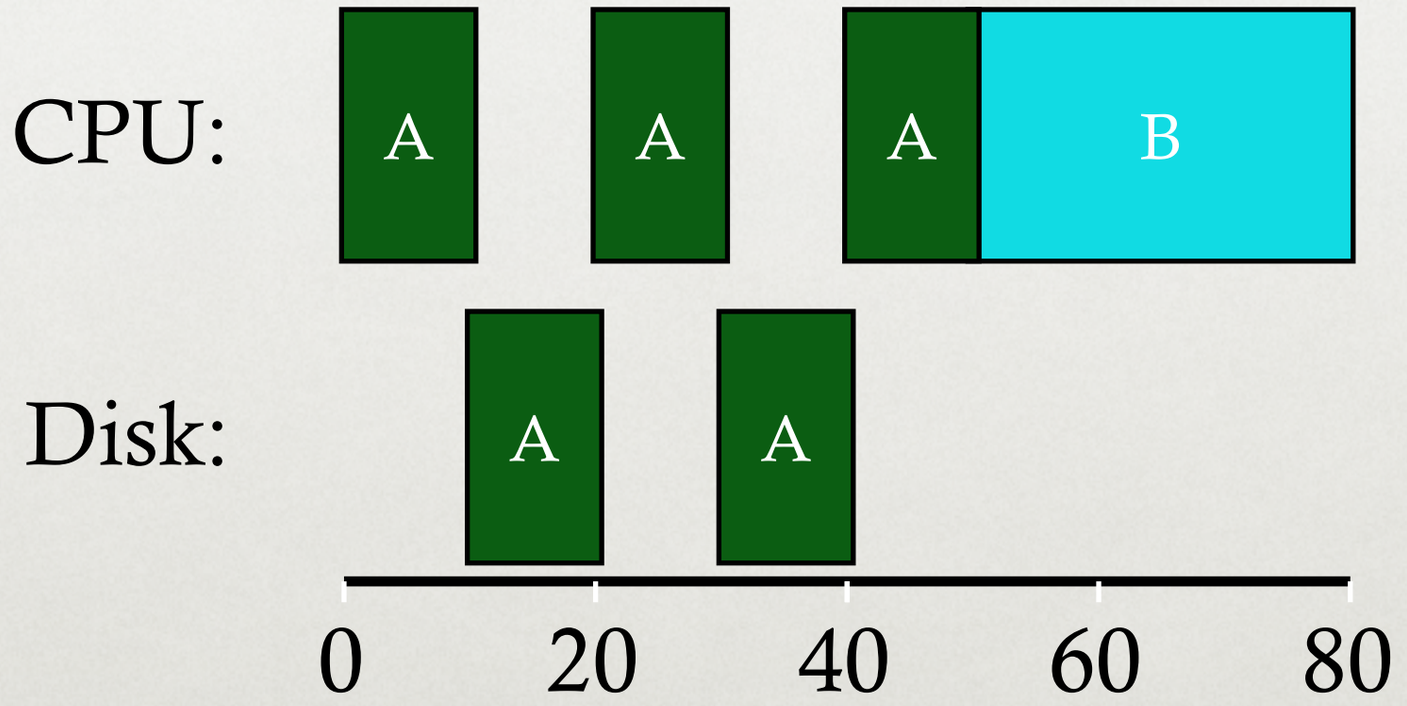
~~1. Each job runs for the same amount of time~~

~~2. All jobs arrive at the same time~~

~~3. All jobs only use the CPU (no I/O)~~

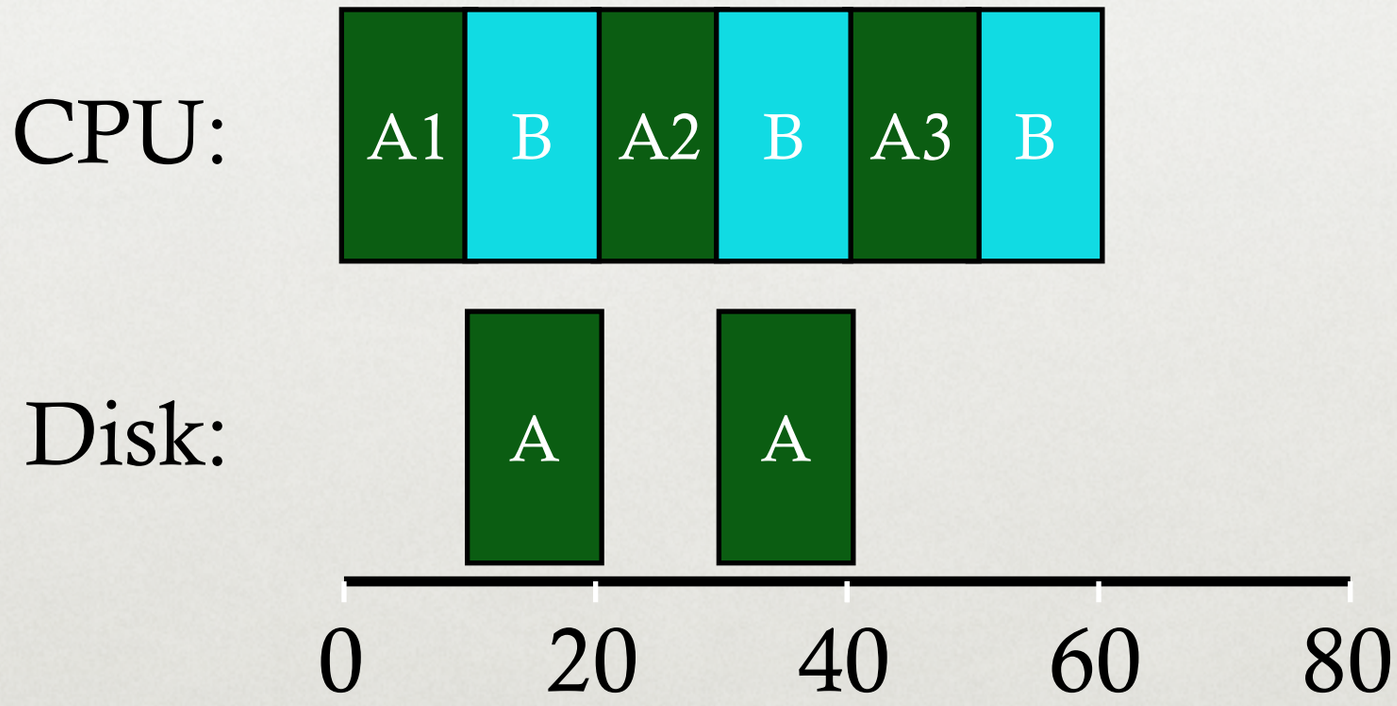
4. The run-time of each job is known

Not I/O Aware



Don’t let Job A hold on to CPU while blocked waiting for disk

I/O Aware (Overlap)



Treat Job A as 3 separate CPU bursts
When Job A completes I/O, another Job A is ready

Each CPU burst is shorter than Job B, so with SCTF,
Job A preempts Job B

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 (no I/O)~~

~~4. The run-time of each job is known~~

(need smarter, fancier scheduler)

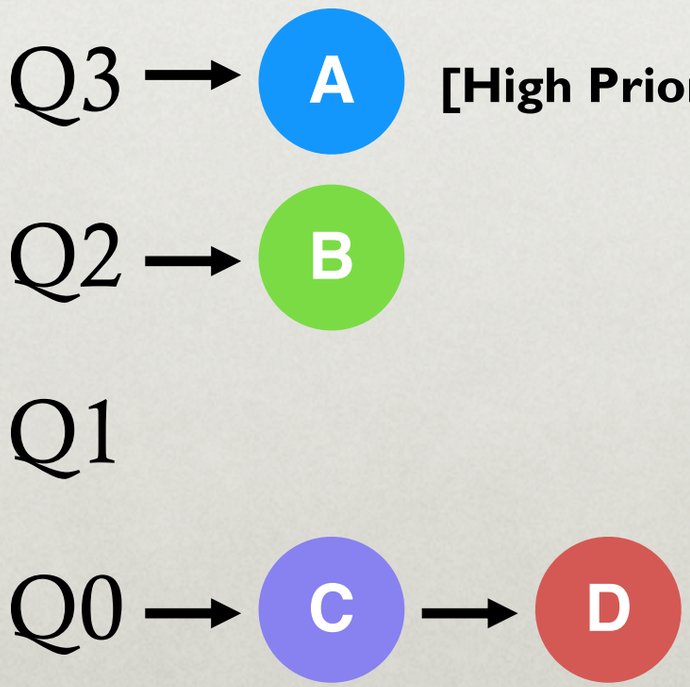
MLFQ
(Multi-Level Feedback Queue)

- Goal: general-purpose scheduling
- Must support two job types with distinct goals
  - "interactive" programs care about response time
  - "batch" programs care about turnaround time
- Approach: multiple levels of round-robin
  - each level has higher priority than lower levels and preempts them
- MLFQ has a number of distinct queues.
- Each queues is assigned a different priority level.

Priorities

Rule 1: If priority(A) > Priority(B), A runs

Rule 2: If priority(A) == Priority(B), A & B run in RR



"Multi-level"

How to know how to set priority?

Approach 1: nice
Approach 2: history "feedback"

History

- Use past behavior of process to predict future behavior
  - Common technique in systems
- Processes alternate between I/O and CPU work
- Guess how CPU burst (job) will behave based on past CPU bursts (jobs) of this process

More MLFQ Rules

Rule 1: If priority(A) > Priority(B), A runs

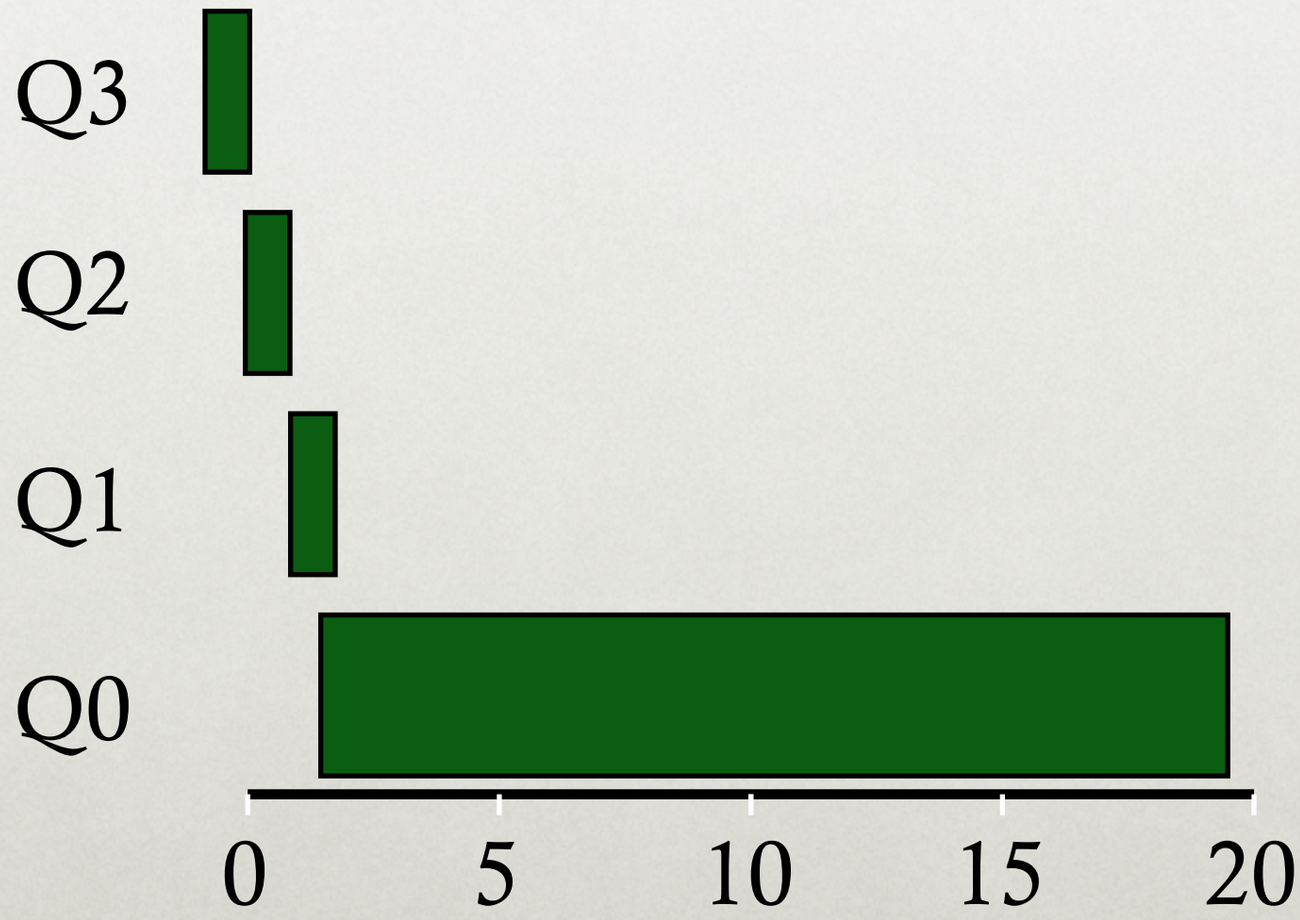
Rule 2: If priority(A) == Priority(B), A & B run in RR

More rules:

Rule 3: Processes start at top priority

Rule 4: If job uses whole slice, demote process (longer time slices at lower priorities)

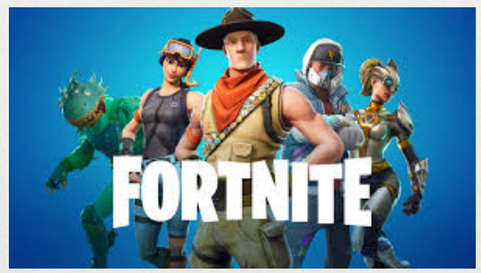
One Long Job (Example)



A four-queue scheduler with time slice 10ms

Long batch job – DNA analysis

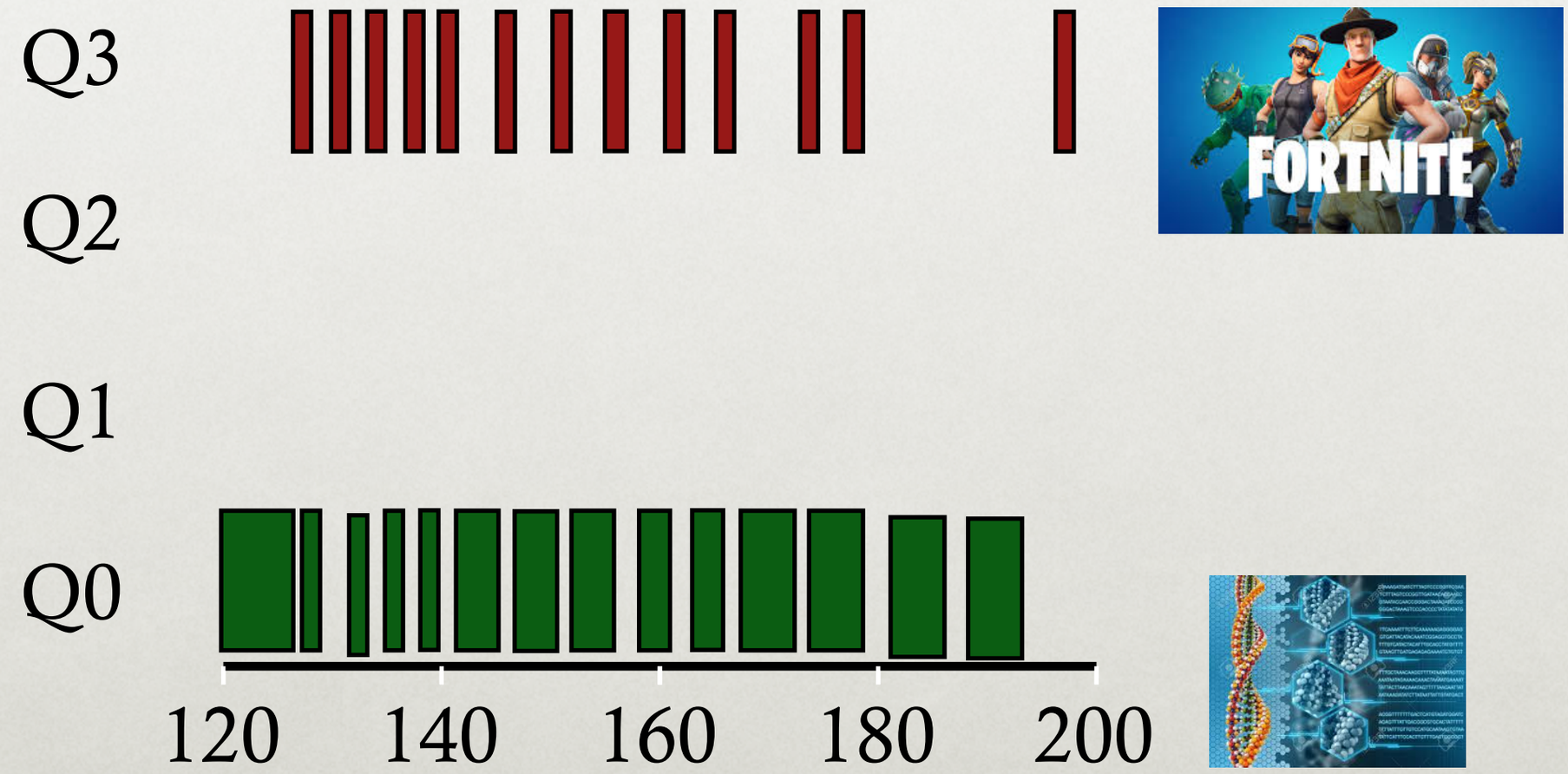
An Interactive Process Joins



Interactive job performs quick operation and does an I/O

Interactive process never uses entire time slice, so never demoted

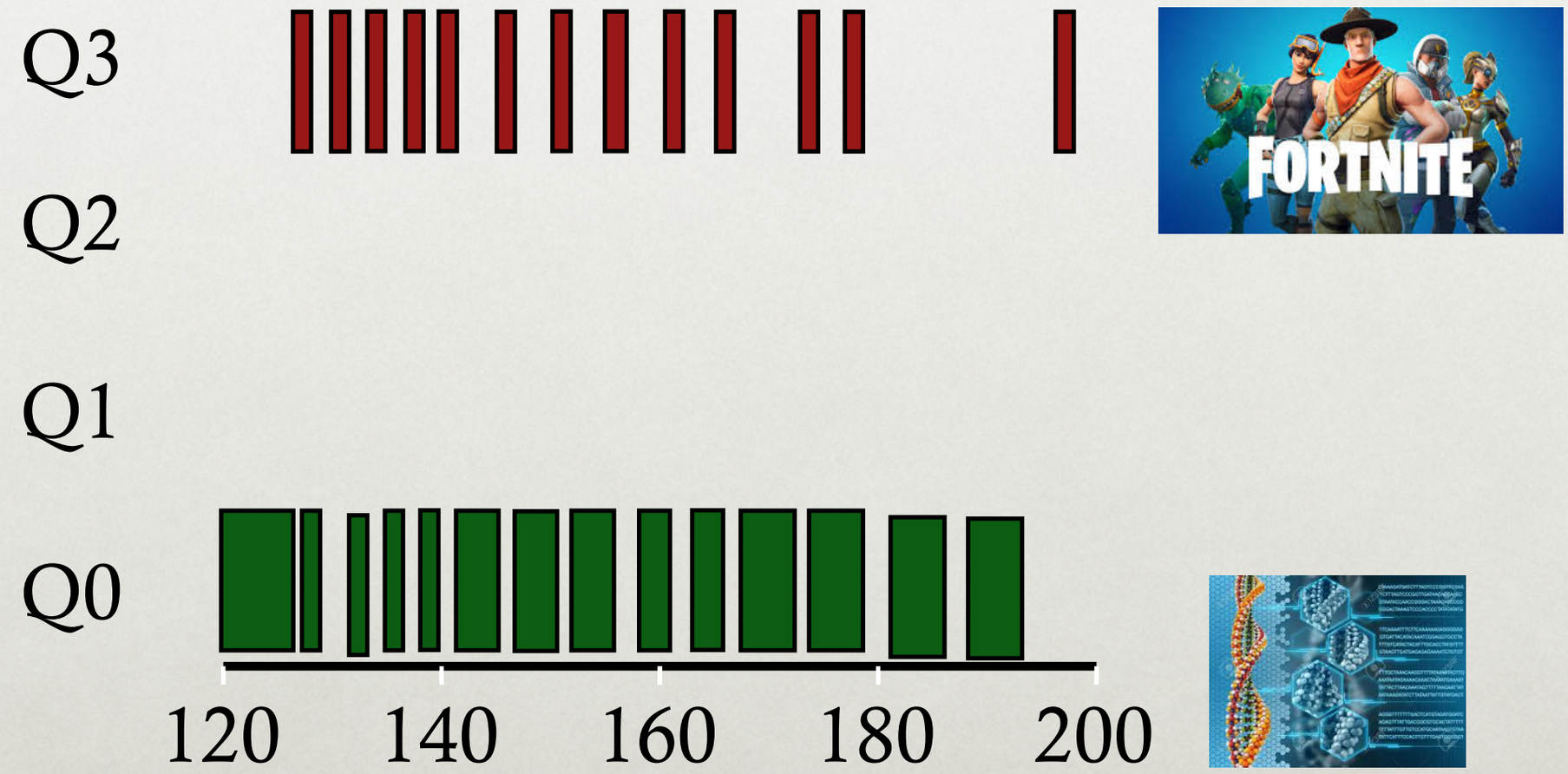
Problems with MLFQ?



Problems

- unforgiving + starvation
- gaming the system

Problems with MLFQ?



Problem: Low priority job may never get scheduled

Periodically boost priority of all jobs (or all jobs that haven't been scheduled)

Lottery Scheduling

Goal: proportional (fair) share

Fair-share scheduler

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

Approach:

- give processes lottery tickets
- whoever wins runs
- higher priority => more tickets

Amazingly simple to implement

Lottery Scheduling

- Tickets
  - Represent the share of a resource that a process should receive
  - 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

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

Lottery Code

```
int counter = 0;
int winner = getrandom(0, totaltickets);
node_t *current = head;
while (current) {
        counter += current->tickets;
        if (counter > winner) break;
        current = current->next;
}
// current is the winner
```

Lottery example

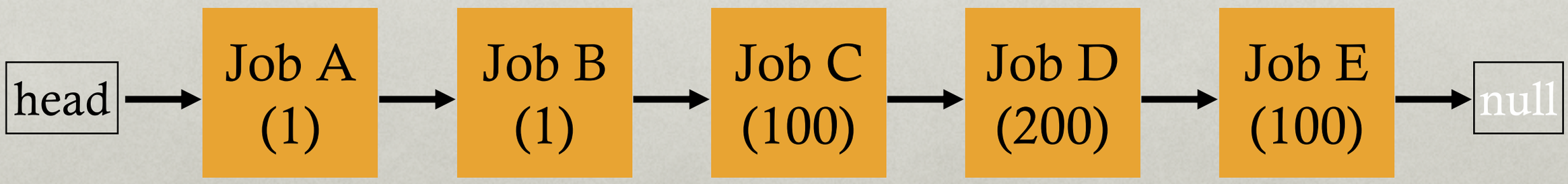
```
int counter = 0;
int winner = getrandom(0, totaltickets);
node_t *current = head;
while(current) {
    counter += current->tickets;
    if (counter > winner) break;
    current = current->next;
}
// current gets to run
```

Who runs if winner is:

50

350

0



Other Lottery Ideas

Ticket Transfers

Ticket Currencies

Ticket Inflation

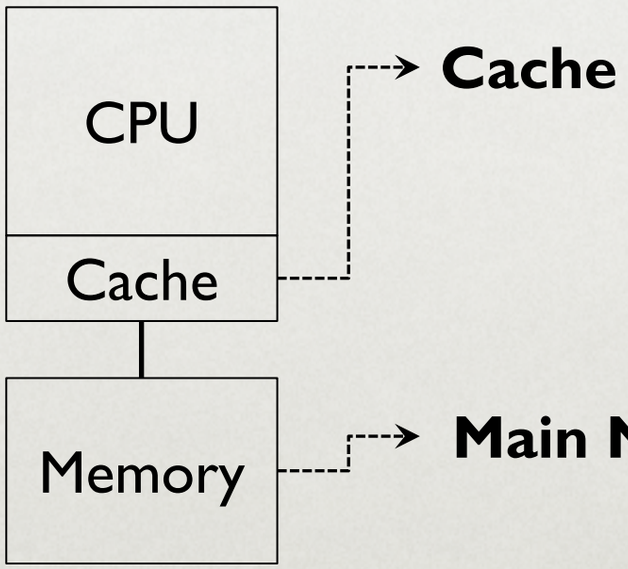
(read more in OSTEP)

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.
  → Rewrite application to run in parallel, using **threads**.

How to schedule jobs on Multiple CPUs?

Single CPU with cache



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

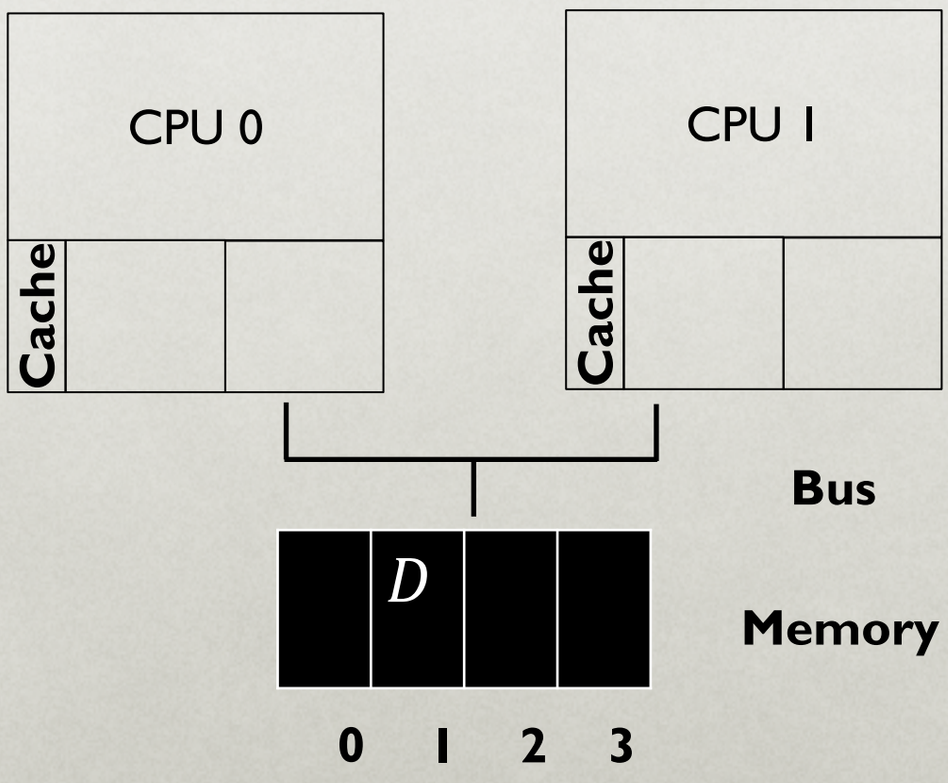
- Holds all of the data
- Access to main memory is slower than cache.

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

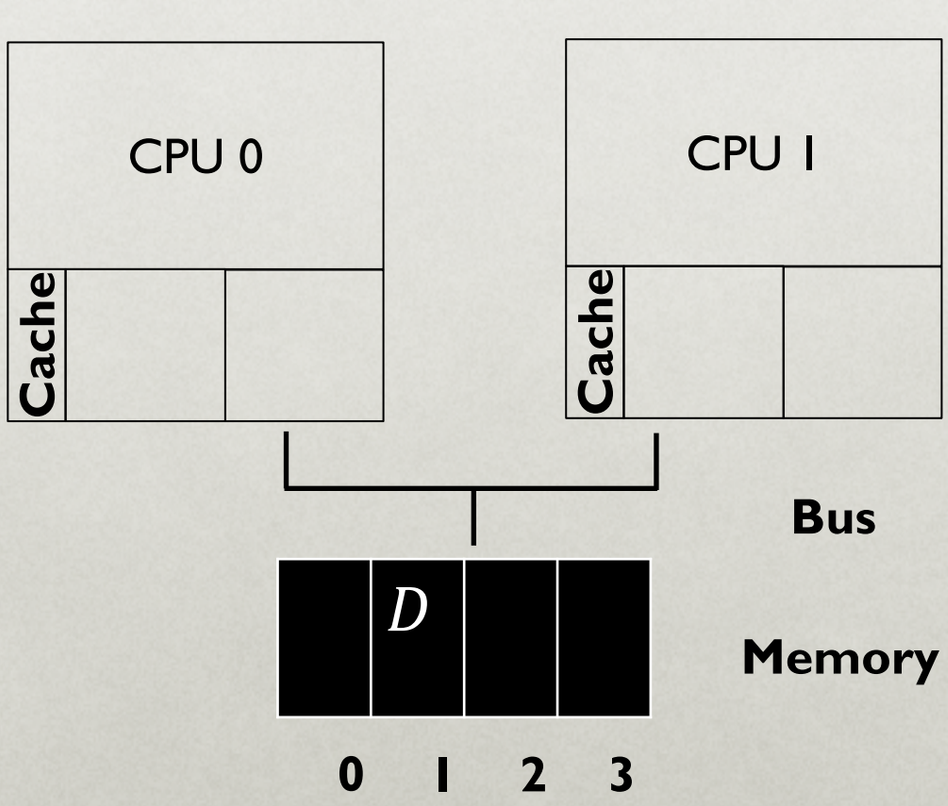


Initial D value is 0

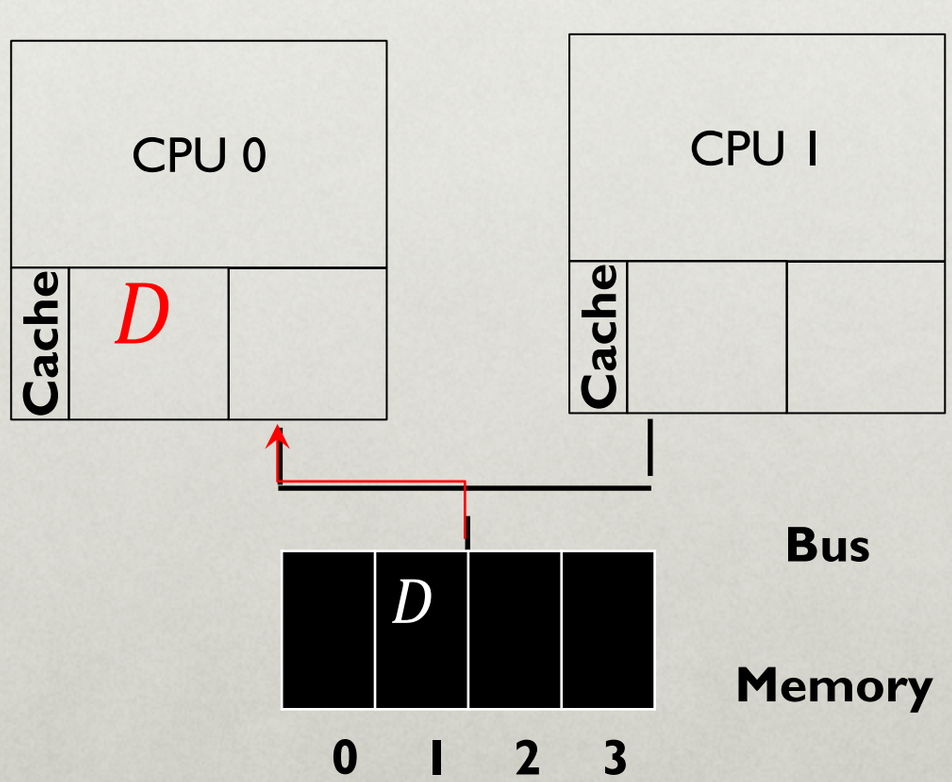
Cache Coherence

- Consistency of shared resource data stored in multiple caches.

0. Two CPUs with caches sharing memory



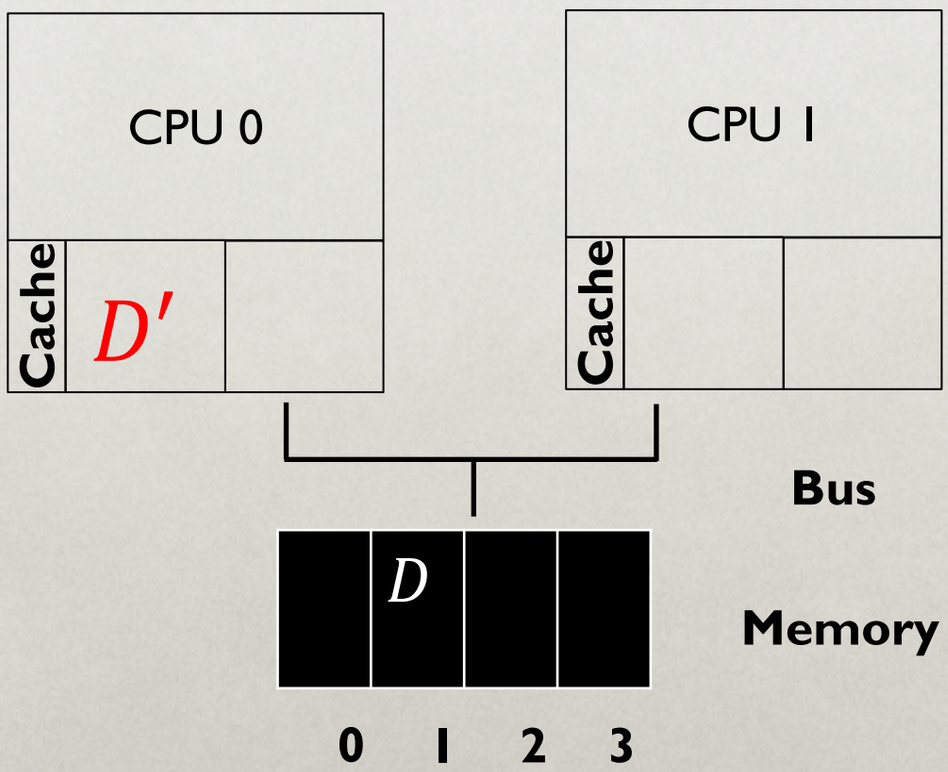
1. CPU0 reads a data at address 1.



Cache Coherence

- Consistency of shared resource data stored in multiple caches.

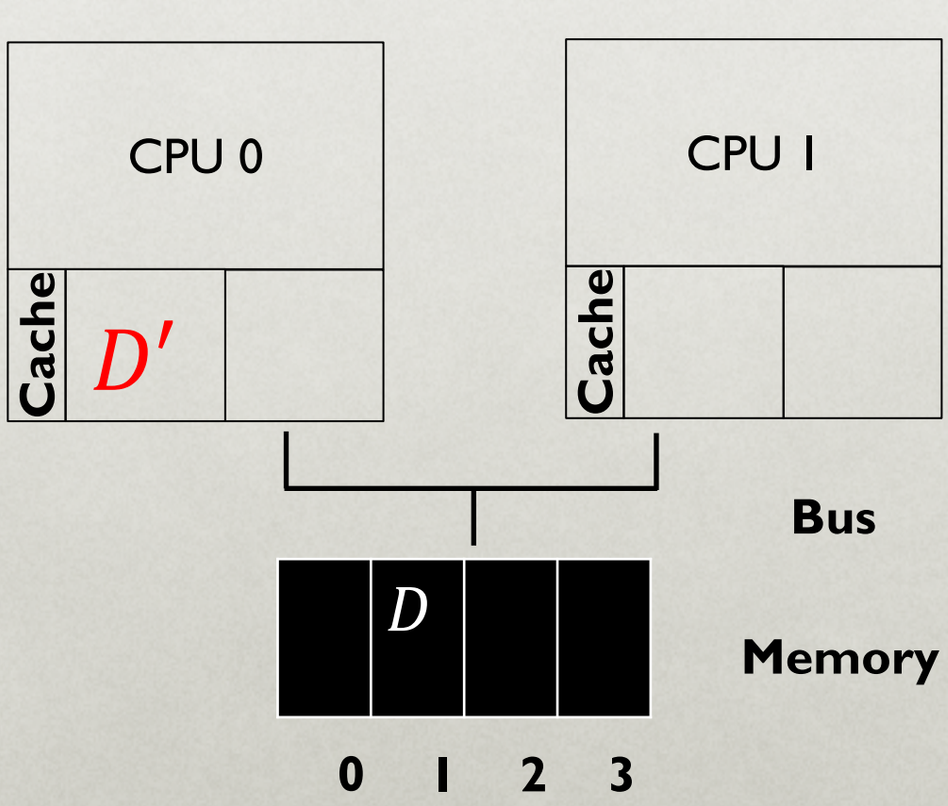
2. D is updated and CPU1 is scheduled.



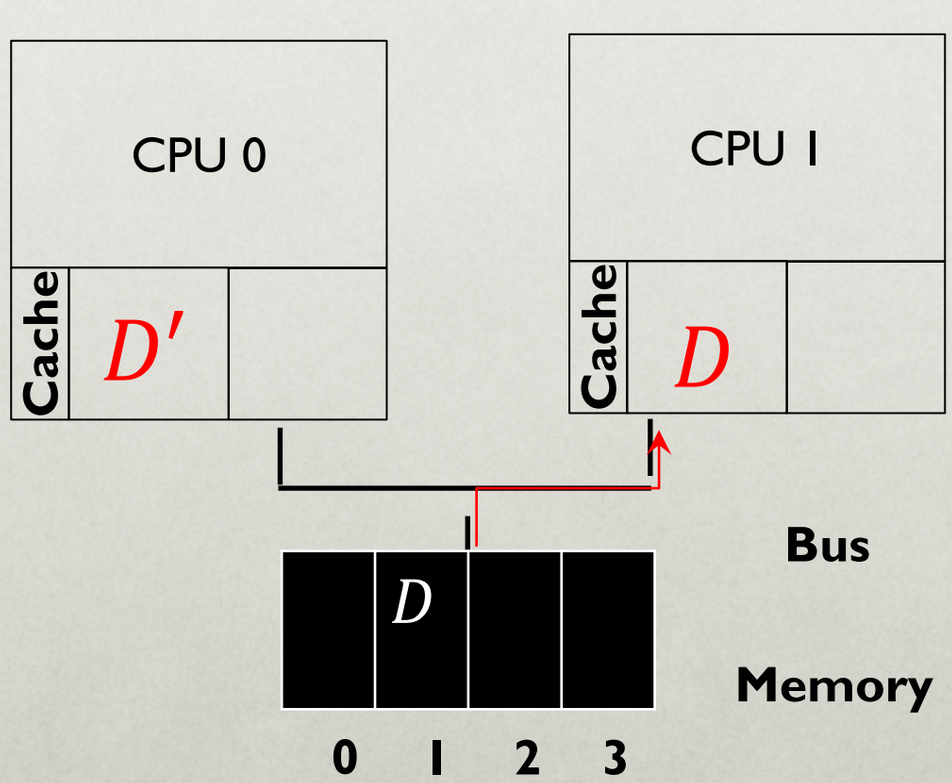
Cache Coherence

- Consistency of shared resource data stored in multiple caches.

2. D is updated and CPU1 is scheduled.



3. CPU1 re-reads the value at address A



Data Inconsistency Problem!

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

Don’t forget synchronization

```
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      }
```

Don’t forget synchronization

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

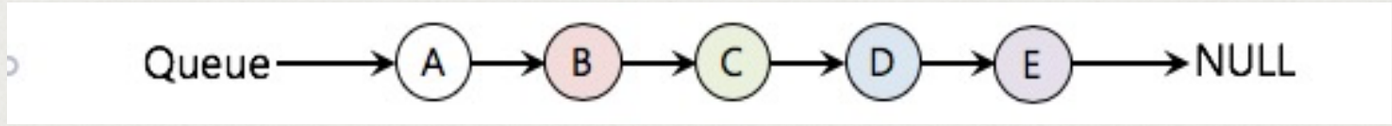
Cache Affinity

- 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.

Cache Affinity

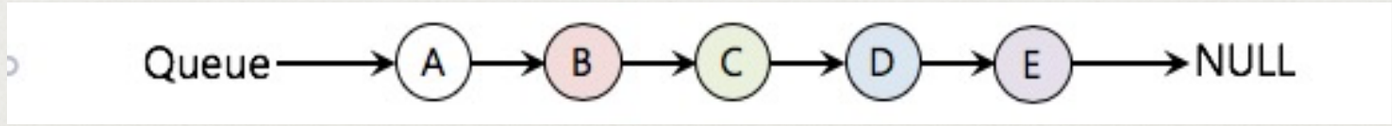
- Put all jobs that need to be scheduled into a single queue.



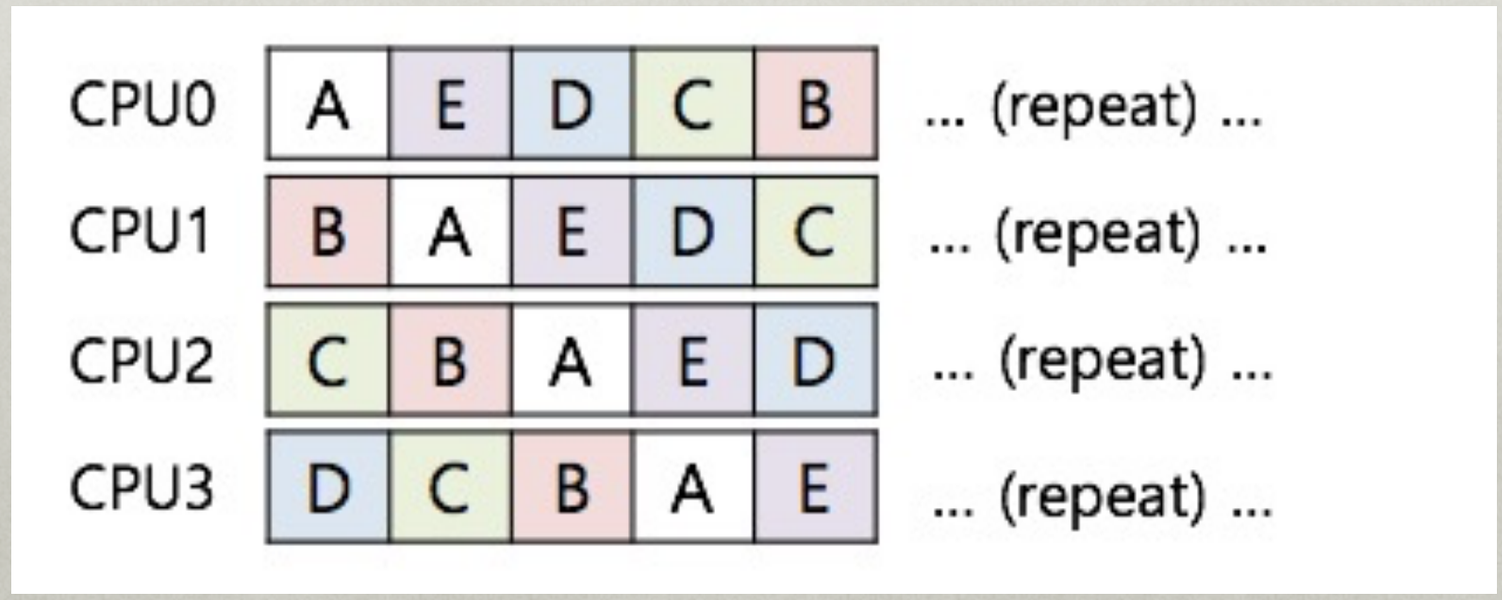
  - Each CPU simply picks the next job from the globally shared queue.
  - Cons:
    - Some form of **locking** have to be inserted → Lack of scalability
    - Cache affinity

Cache Affinity

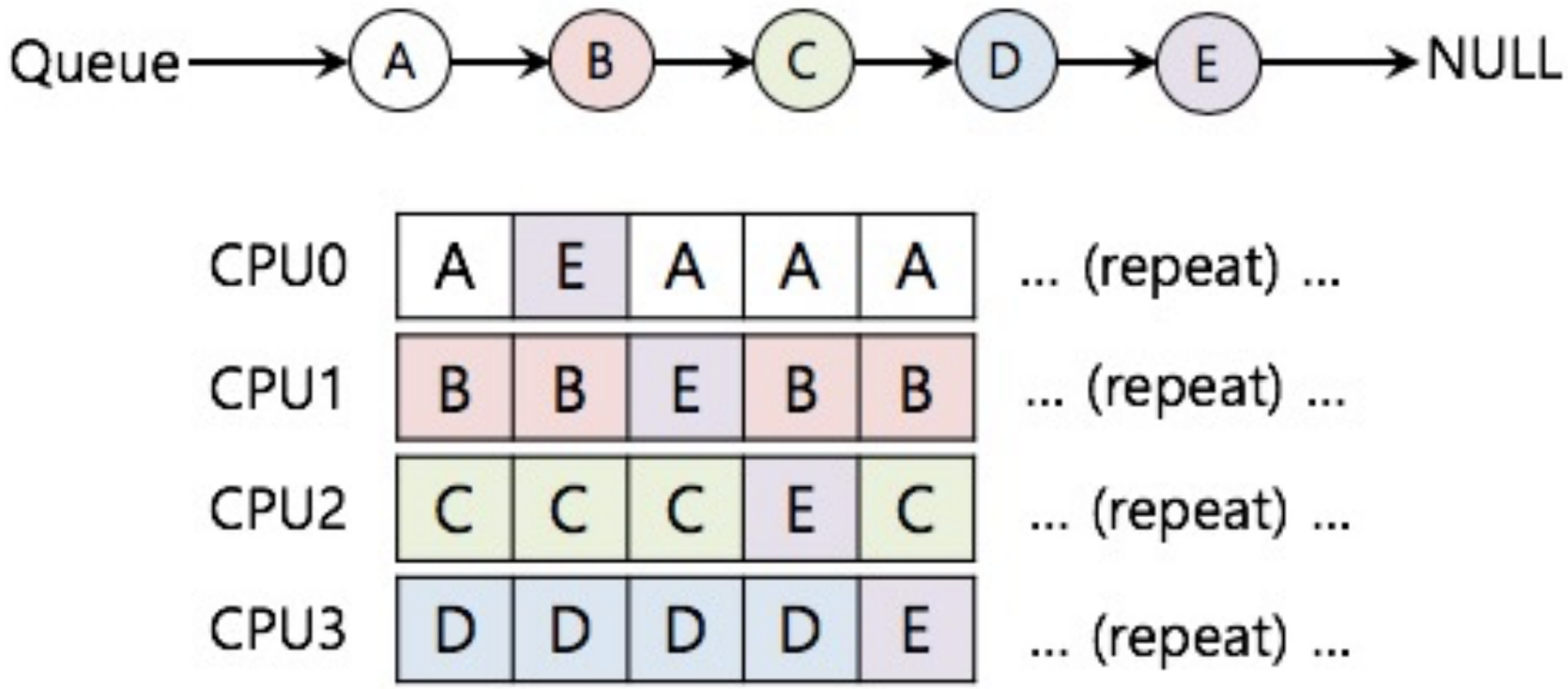
- Put all jobs that need to be scheduled into a single queue.



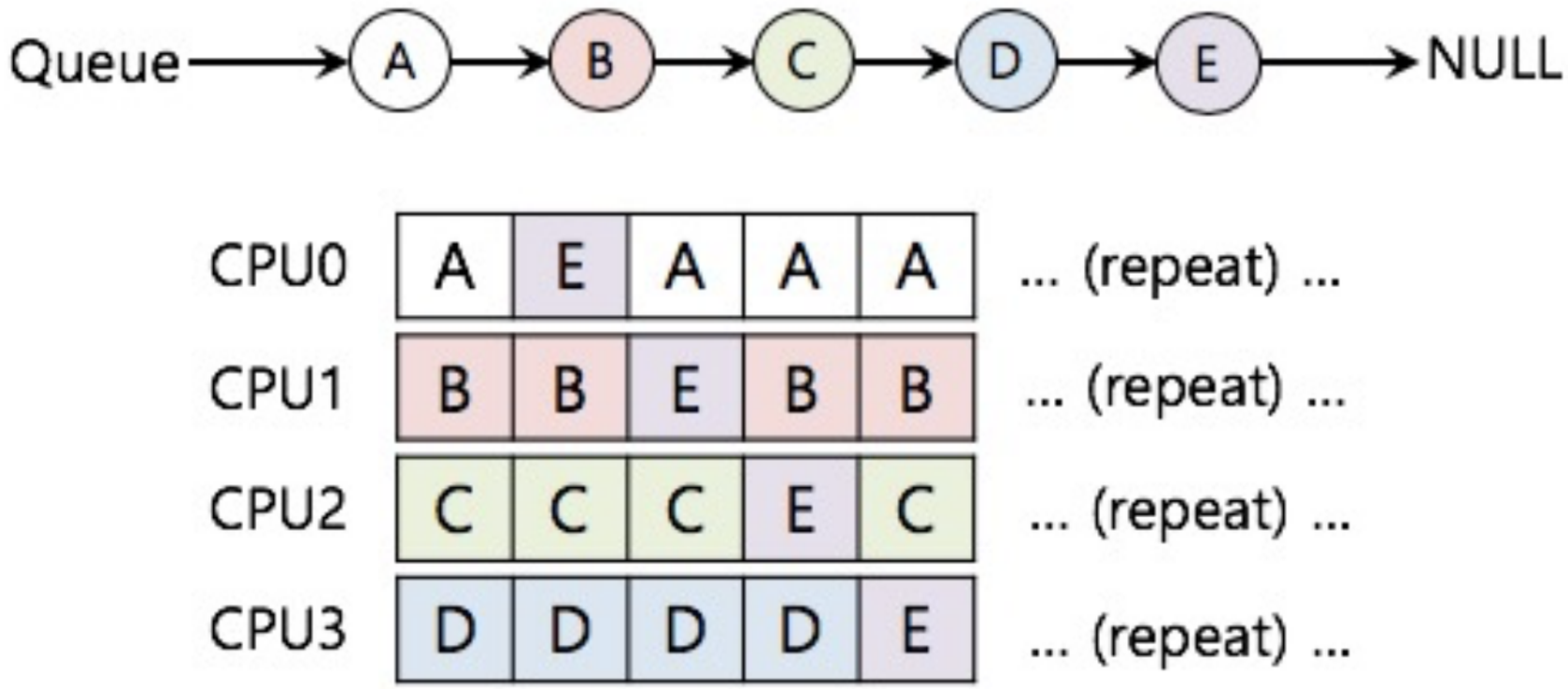
  - Each CPU simply picks the next job from the globally shared queue.
  - Cons:
    - Some form of **locking** have to be inserted → Lack of scalability
    - Cache affinity?



Scheduling Cache Affinity



Scheduling Cache Affinity



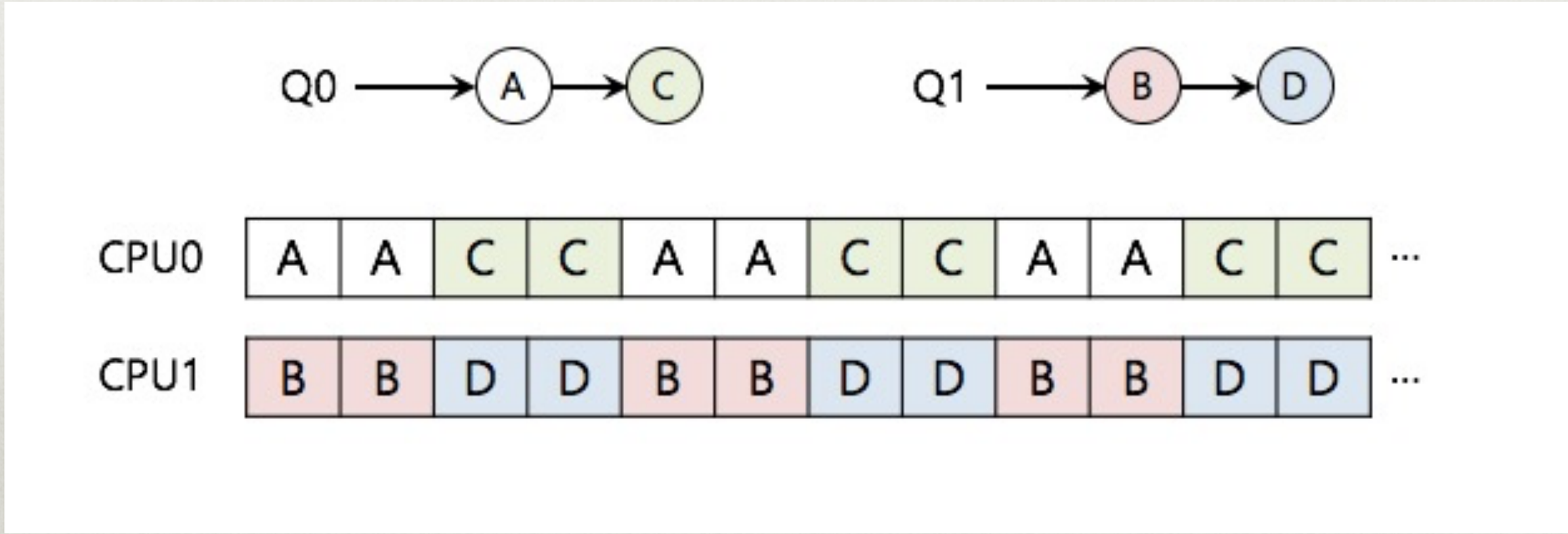
- Preserving affinity for most
  - Jobs A through D are not moved across processors.
  - Only job **E** Migrating from CPU to CPU.
- Implementing such a scheme can be **complex**.

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.

Multi-queue Multiprocessor Scheduling (MQMS)

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



MQMS provides more scalability and cache affinity.

Completely Fair Scheduling (CFS)

CFS changes or removes time-slice allotment

- Do away with timeslices completely
- Assign each process a proportion of the processor
- Yields constant fairness but a variable switching rate

CFS is based on a simple concept

Model scheduling as if the system had an ideal, perfectly multitasking processor

- Each process receives 1/n of the processor's time, where n is the number of runnable processes
- Would be scheduled for infinitely small durations
- In any measurable period, all n processes would run for the same amount of time

Completely Fair Scheduling (CFS)

CFS changes or removes time-slice allotment

- Do away with timeslices completely
- Assign each process a proportion of the processor
- Yields constant fairness but a variable switching rate

Assume there are two processes

- Unix model - run one process for 5ms and another for 5ms
- Each process receives 100% of the processor
- IDEAL System: Perfect multitasking processor
  - Run both processes simultaneously for 10 milliseconds
  - Each at 50% power - called perfect multitasking

Ideal Multiprocessor possible?

Why not?

Completely Fair Scheduling (CFS)

Rank processes based on their worth and need for processor time

Processes with a higher priority run before those with a lower priority

Linux has two priority ranges

- Nice value: ranges from -20 to +19 (default is 0)
- High values of nice means lower priority
- Real-time priority: ranges from 0 to 99
- Higher values mean higher priority
- Real-time processes always executes before standard (nice) processes

ps ax -eo pid,ni,rtprio,cmd

Linux CFS

Rank processes based on their worth and need for processor time

Processes with a higher priority run before those with a lower priority

Linux has two priority ranges

- Nice value: ranges from -20 to +19 (default is 0)
- High values of nice means lower priority
- Real-time priority: ranges from 0 to 99
- Higher values mean higher priority
- Real-time processes always executes before standard (nice) processes

ps ax -eo pid,ni,rtprio,cmd

Linux CFS

Assume the targeted latency is 20 milliseconds

If there are two runnable tasks at the same priority

Each will run for 10 milliseconds before preempting in favor of the other

If we have four tasks at the same priority

Each will run for 5 milliseconds

If there are 20 tasks, each will run for 1 millisecond

As the number of runnable tasks approaches infinity, the proportion of allotted processor and the assigned timeslice approaches zero

Problem? (Solution: Run any task for a minimum of 1millisecond)

Linux CFS

Common case of only a handful of runnable processes, CFS is perfectly fair

Consider the case of two runnable processes

Process1: Nice value 0

Process2: Nice value 5

Processes receive different proportions of the processor's time

The weights work out to about a 1/3 penalty for the nice-5 process.

If the target latency is again 20 milliseconds

The two processes will receive 15 milliseconds and 5 milliseconds each of processor time, respectively

Summary

Understand goals (metrics) and workload, then design scheduler around that

General purpose schedulers need to support processes with different goals

Past behavior is good predictor of future behavior

Random algorithms (lottery scheduling) can be simple to implement, and avoid corner cases.

Important to consider caches in multiprocessor scheduling

* Materials modified and reused from OSTEP book and lectures of Prof. Andrea Arpaci-Dusseau and Prof. Yojip Won

Disclaimer: Materials reused, modified from OSTEP book and lectures of Prof. Andrea and Remzi Arpaci-Dusseau and Prof. Yojip Won

Completely Fair Scheduling (CFS)

Ideal Multiprocessor possible?

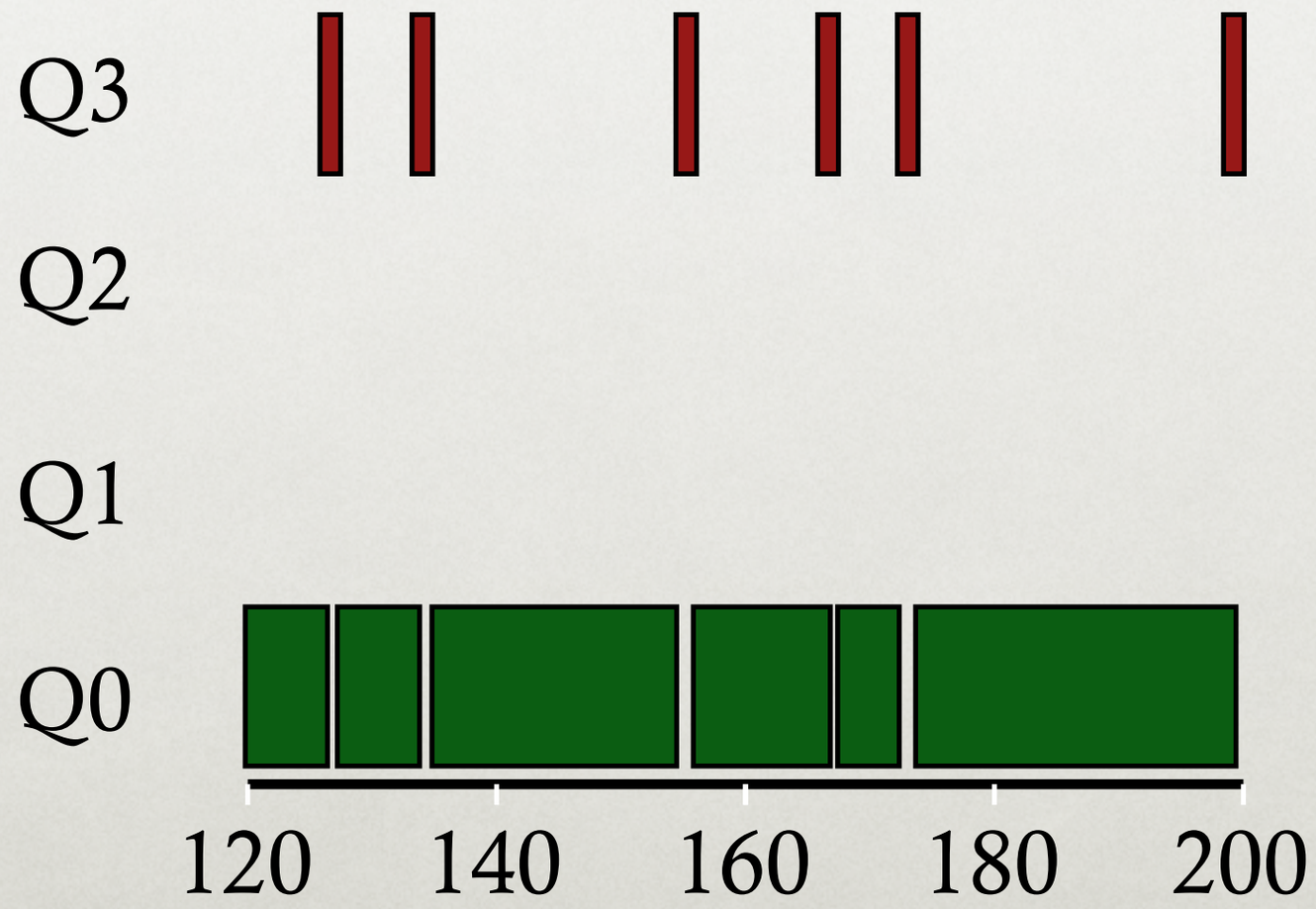
Impossible on a single processor to run multiple processes simultaneously

- Inefficient to run processes for infinitely small durations
- Switching cost to preempting one process for another
- Overhead of swapping one process for another and the effects on caches

CFS is mindful of the overhead and performance

- Runs each process for some amount of time, round-robin, selecting next the process that has run the least
- **Calculates how long a process should run as a function of the total number of runnable processes**

Prevent Gaming



Problem: High priority job could trick scheduler and get more CPU by performing I/O right before time-slice ends

Fix: Account for job's total run time at priority level (instead of just this time slice); downgrade when exceed threshold