Advanced CPU Scheduling

ICS332

Operating Systems

Henri Casanova (henric@hawaii.edu)

CPU Scheduling in the Real World

- The previous set of lecture notes goes over the basics of scheduling
- The punch line was: use RR with a good time quantum
- Unfortunately, things are not as simple
- Let look at 3 more advanced scheduling topics:

1. Multi-Level Feedback Queue (OSTEP Chapter 8)
   - Approach used in most real-world OSes, including Windows
2. Multi-Processor Scheduling
   - We'll only skim the surface here (OSTEP chapter 10 if you want more details)
3. What Linux does
   - We'll only skim the surface here as Linux scheduling has a long/complex history with many different approaches

The Time Quantum Conundrum

- We don't want to use too-small a RR time quantum
  - Context-switch overhead would reduce CPU (useful) utilization, which ends up harming CPU-intensive jobs
- But then the time quantum is not small!
- This means that "interactive" jobs can appear less interactive (when there are many jobs)
  - e.g., I type in my text editor, and there is a noticeable lag before it responds
  - Interactive jobs typically have just occasional small CPU bursts and many I/O bursts
- We have a conundrum:
  - Interactive jobs want a small time quantum
  - CPU-bound want what a large time quantum

Multi-Level Feedback Queue (MLFQ)

- You have noted that everything seems to work fine on your machine
  - You can run a bunch of apps, and still type in your text editor without experiencing lags
- This means that OSes do something to keep interactive jobs happy
- One such “something” is Multi-Level Feedback Queue (MLFQ)
- **The goal:** Make sure interactive jobs never get stuck in the Ready Queue due to CPU-bound jobs
  - Interactive jobs should get the CPU as soon as they want it every time

MLFQ Basic Idea #1: Priorities

- To make sure that interactive jobs are not stuck behind CPU-bound jobs the solution is to use priority levels, and use one Round-Robin Ready Queue per level

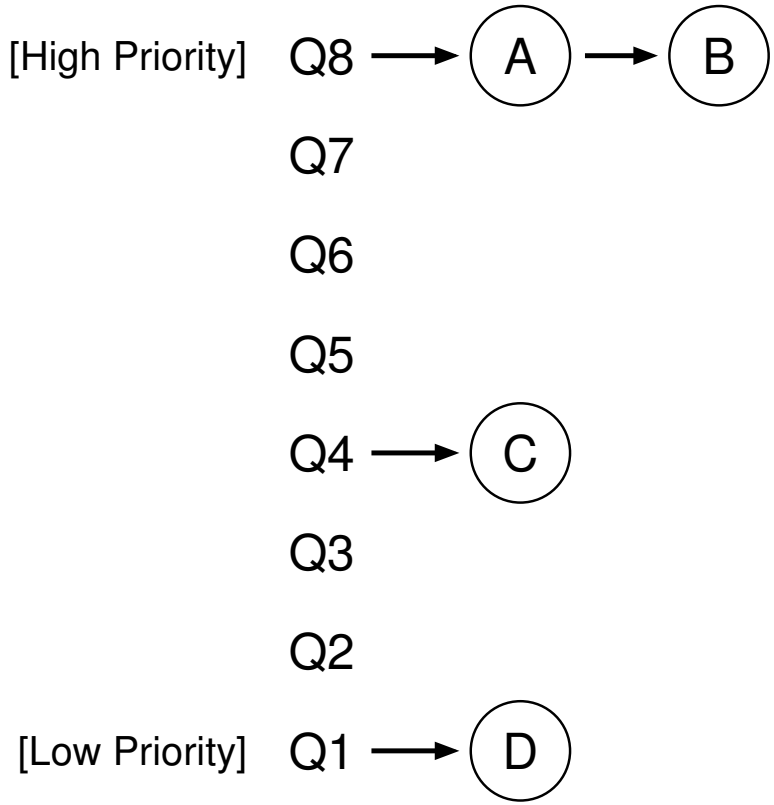


Figure 8.1: **MLFQ Example**

MLFQ Basic Idea #1: Priorities

- Simple Rules:
  - If Priority(A) > Priority(B) then A runs and B doesn't
    - A higher-priority job ready to run always preempts a lower-priority job
  - If Priority(A) == Priority(B) then A and B run in Round-Robin
- Ideally, we want interactive jobs in high-priority queues
- So that on the "rare" occasion they need the CPU they get it quickly
- Remember that once a job does I/O it is no longer in any Ready Queue
- We also want jobs to be demoted/promoted to lower/higher queues when they stop/start being less/more interactive

- **Big Question:** How do we decide job priorities??
  - It's not like programs say "Hey OS, just to let you know, I am about to become interactive!"
  - Besides they'd be lying all the time to get more CPU

- **We need a way to automatically detect job interactiveness**

MLFQ Basic Idea #1: Priorities

- When a job first shows up, we put it in the highest-priority queue
  - We don't know anything about it, we conservatively assume it will be interactive
- We now need a way to demote non-interactive jobs!
- **Key Insight:**
  - Interactive jobs do not use their time quanta fully because they always have short CPU bursts
  - CPU-intensive, non-interactive jobs use their time quanta fully because they always have long CPU bursts
- The OS has information about whether each job uses its time quantum fully or not!
  - Either a job places an I/O syscall before the "time quantum expired" timer goes off (time quantum was not fully used)
  - Either the "time quantum expired" timer goes off and the job is still doing its fetch-decode-execute cycle (time quantum was fully used)

- So now we have a **simple strategy**:
  - If a job uses its full time quantum, it is demoted!
  - If a job does not use its full time quantum, then it's not demoted

Simple 2-job Example

- Say we have an interactive job and a CPU-bound job
- At the beginning both are in the high-priority queue
- As soon as the CPU-bound job completes its first time quantum, it is demoted to a lower-priority queue
- At that point, the CPU-bound job can only run whenever the interactive job is doing I/O
- The interactive job runs as if it was alone on the machine

- See Figure 8.4 and its description in OSTEP

Problem: Starvation

- A clear problem with what we have so far is that a CPU-bound job may never run
- This can happen if we have too many interactive / I/O-bound jobs
- Example:
  - A CPU-bound job
  - Two I/O-bound jobs that use 1/2 of their time quantum for each CPU burst
  - The two I/O-bound jobs are in perfect synchrony: when one finishes its time quantum the other is always ready to start its time quantum
  - In this situation, after its initial demotion, the CPU-bound job will never run
  - This is starvation
- See Figure 8.5 (left side) and its description in OSTEP

Other Problems: Gaming the System

- Gaming the system:
  - A more insidious problem is that a very smart user could game the scheduler
  - If I know the time quantum duration, I can have my program do some fast, useless I/O operation, right before my time quantum expires
  - As a result, my program never uses its time quantum fully, and remains at the highest priority always!
  - Basically I am masquerading as an interactive job
- Jobs that change behavior
  - Say a job is first CPU-intensive, so it's demoted to the lowest priority queue
  - At some point later it becomes interactive
  - At that point, it will appear very unresponsive to the user
- **Bottom-line:** We need to treat CPU-bound jobs a bit better

Solution: Priority Boost

- A simple approach: every S seconds, move all jobs back to the highest priority queue, and let them trickle back down
- This is called a Priority Boost
- See Figure 8.5 (right side) and its description in OSTEP
- Note that we haven't fixed the "gaming the system" problem (see OSTEP Section 8.4)
  - By the way, on UNIX-like systems you can be `nice` (let's look at the man page)
- An immediate question: "what's a good value of S? "
  - If S is too big, then CPU-bound jobs will be unhappy
  - If S is too small, then interactive jobs may lag
- Each time we add a parameter to a strategy, we raise the question of "what's a good value?"
  - Some call these "Voodoo constants" because picking good values is a dark art
  - See the Ousterhout's Law insert in OSTEP

MLFQ Parameters

- We have defined the general MLFQ approach
- In practice we have many "voodoo" constants
  - The number of priority levels
  - The time quantum duration for the Ready Queue at each priority level
  - The duration S after which all jobs experience a priority boost
- It's all easy to implement based on a configuration, but the question is: what's a good configuration?
- People have experimented over the years and a good rule of thumb is: larger time quanta for lower-priority ready queues

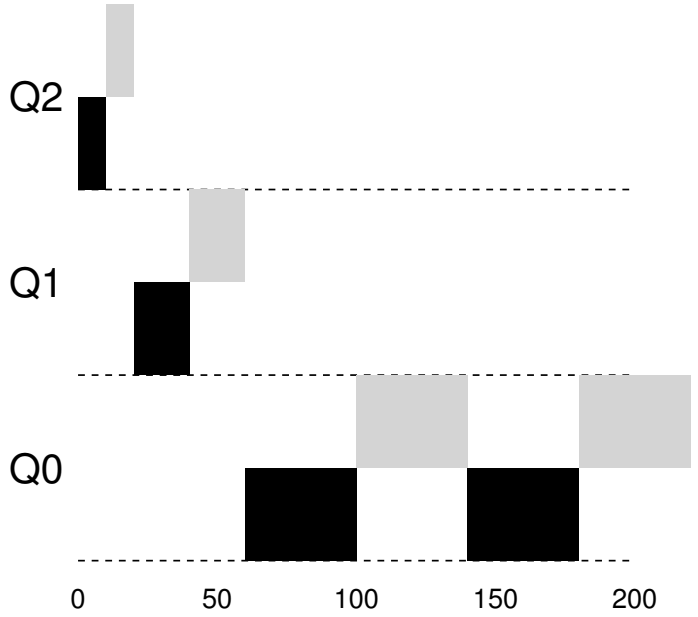


Figure 8.7: **Lower Priority, Longer Quanta**

MLFQ in Real OSes

- The basic MLFQ idea is used in Solaris, FreeBSD (and thus MacOS), and Windows
- The Solaris OS (Sun Microsystems) scheduler is basically what we just described
  - A config file describes the queue configurations
  - Raises the question of how often the default values are actually tuned
  - In your career you'll encounter systems with many configuration parameters, and setting these parameters correctly is a challenge and often not done
  - Not letting the number of parameters grow too large when designing a system is always a good idea, but not easy
- In other OSes the basic approach is MLFQ, but there are a bunch of added bells and whistles
  - The variant of MLFQ in the FreeBSD OS does some accounting: priority is tuned based on how much CPU a job has used
  - Windows uses a mix of MLFQ and other scheduling approaches

In-Class Exercise

- Consider the following jobs:
  - A: CPU burst time 1ms, I/O burst time 5ms (Disk)
  - B: CPU burst time 4ms, I/O burst time 2ms (NIC)
  - C: CPU burst time ∞ ms
- At time t = 0, all are in the ready queue (in the A, B, C order), and all begin execution with a CPU burst
- The OS uses a 2-queue MLFQ: 5ms quantum for top queue, 20ms quantum for bottom queue
- There is no priority boost
- Plot the CPU utilization time-line for 34 ms
- Use letter A / B / C for 1ms of execution of job A / B / C
- Use letter I for 1ms of idle time
- Example: **`AA B III CCC AAAA B II B`**
- Same question assuming one priority boost at time 17

Solution

- A: 1ms / 5ms; B: 4ms / 2ms; C: ∞ ms
- Top queue: 5ms; Bottom queue: 20ms

CPU **A**

A's 1st time quantum, which is not fully utilized

Disk

NIC

Solution

- A: 1ms / 5ms; B: 4ms / 2ms; C: ∞ ms
- Top queue: 5ms; Bottom queue: 20ms

CPU **ABBBB**

B's 1st time quantum, which is not fully utilized

Disk **AAAAA**

NIC

Solution

- A: 1ms / 5ms; B: 4ms / 2ms; C: ∞ ms
- Top queue: 5ms; Bottom queue: 20ms

CPU **ABBBBCCCCC**

Disk **AAAAA**

NIC **BB**

C's 1st time quantum, which is fully utilized, and so C is **demoted** to the lower queue

Solution

- A: 1ms / 5ms; B: 4ms / 2ms; C: ∞ ms
- Top queue: 5ms; Bottom queue: 20ms

CPU **ABBBBCCCCCA**

A's 2nd time quantum

Disk **AAAAA**

NIC **BB**

Solution

- A: 1ms / 5ms; B: 4ms / 2ms; C: ∞ ms
- Top queue: 5ms; Bottom queue: 20ms

CPU **ABBBBCCCCCCABBBB**

B's 2nd time quantum

Disk **AAAAA** **AAAAA**

NIC **BB**

Solution

- A: 1ms / 5ms; B: 4ms / 2ms; C: ∞ ms
- Top queue: 5ms; Bottom queue: 20ms

CPU	**ABBBBCCCCCABBBBC**		
Disk	**AAAAA**	**AAAAA**	
NIC	**BB**	**BB**	

C's 2nd time quantum, which is only 1ms because A is ready and of a higher priority!!

Solution

- A: 1ms / 5ms; B: 4ms / 2ms; C: ∞ ms
- Top queue: 5ms; Bottom queue: 20ms

CPU **ABBBBCCCCCABBBBCA**

A's 3rd time quantum

Disk **AAAAA** **AAAAA**

NIC **BB** **BB**

Solution

- A: 1ms / 5ms; B: 4ms / 2ms; C: ∞ ms
- Top queue: 5ms; Bottom queue: 20ms

CPU **ABBBBCCCCCABBBBCABBBB**

B's 3rd time quantum

Disk **AAAAA** **AAAAA** **AAAAA**

NIC **BB** **BB**

Solution

- A: 1ms / 5ms; B: 4ms / 2ms; C: ∞ ms
- Top queue: 5ms; Bottom queue: 20ms

CPU **ABBBBCCCCCABBBBCABBBBC**

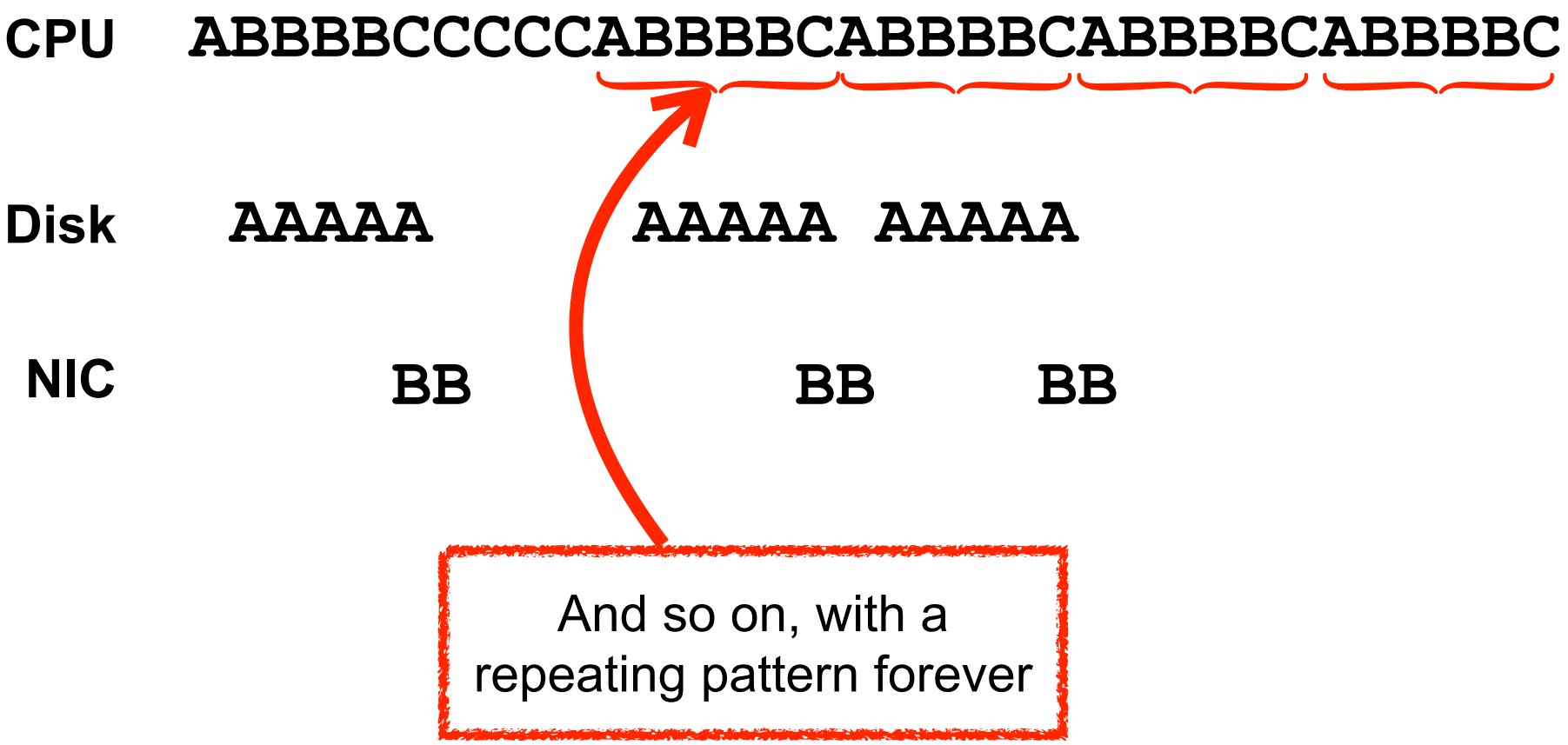
C's 3rd time quantum

Disk **AAAAA AAAAA AAAAA**

NIC **BB BB BB**

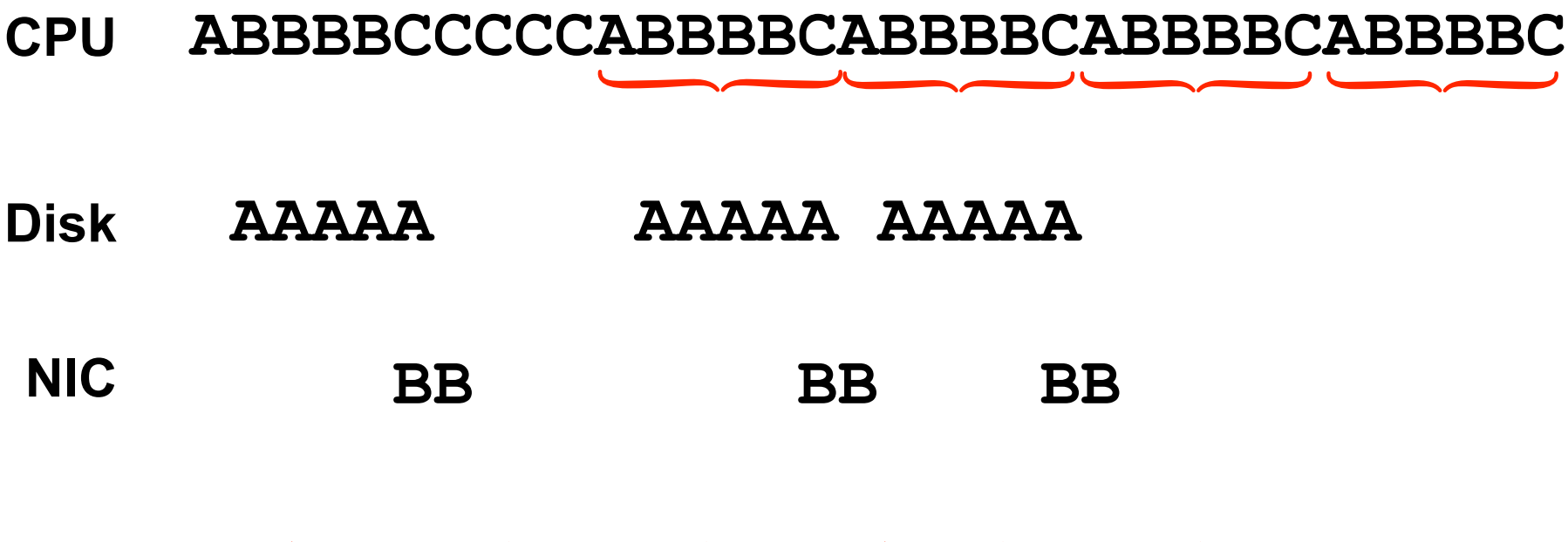
Solution

- A: 1ms / 5ms; B: 4ms / 2ms; C: ∞ ms
- Top queue: 5ms; Bottom queue: 20ms



Solution

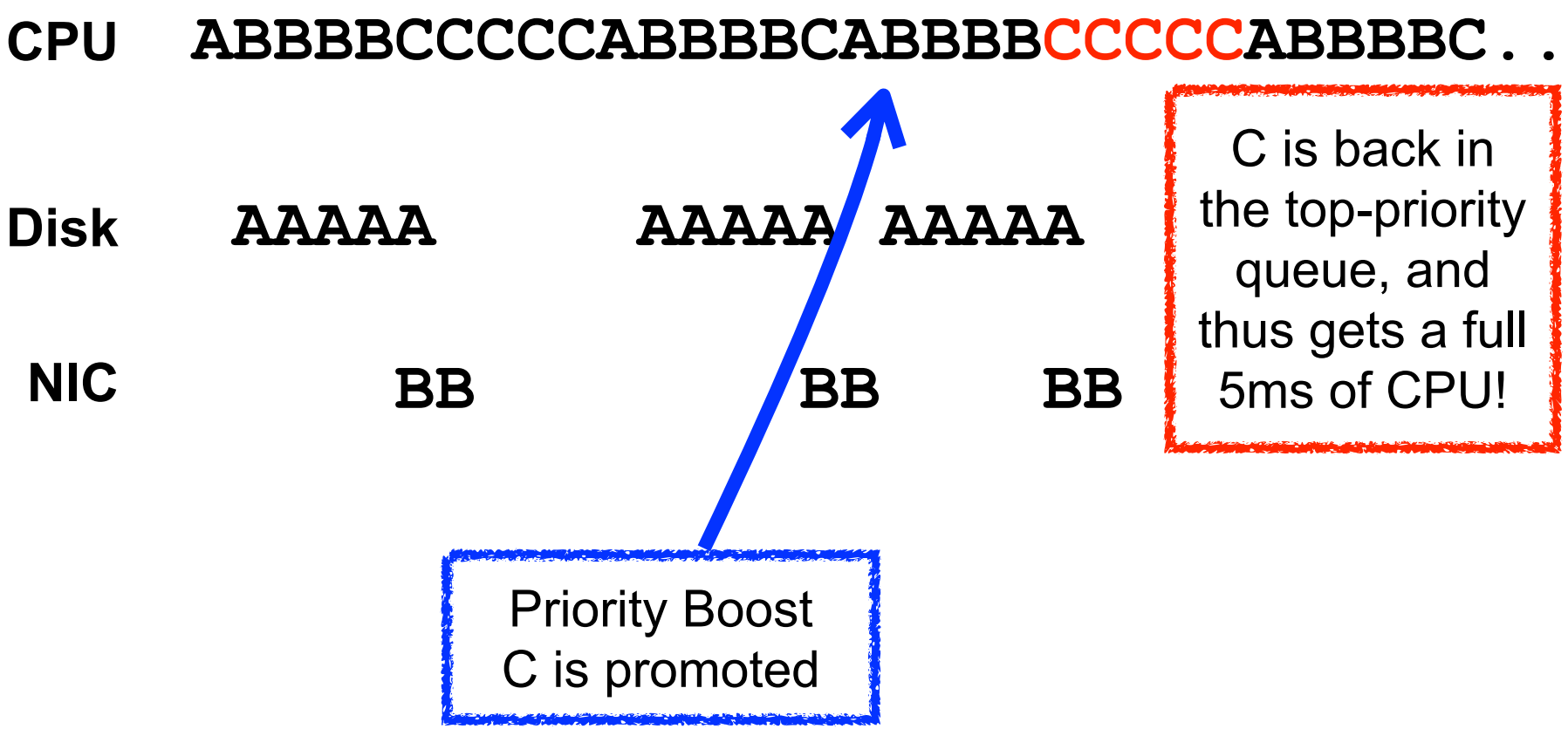
- A: 1ms / 5ms; B: 4ms / 2ms; C: ∞ ms
- Top queue: 5ms; Bottom queue: 20ms



Note that the time quantum of the Bottom queue doesn't matter in this example as job C runs for at most 1ms anyway

Solution (with priority boost)

- A: 1ms / 5ms; B: 4ms / 2ms; C: ∞ ms
- Top queue: 5ms; Bottom queue: 20ms

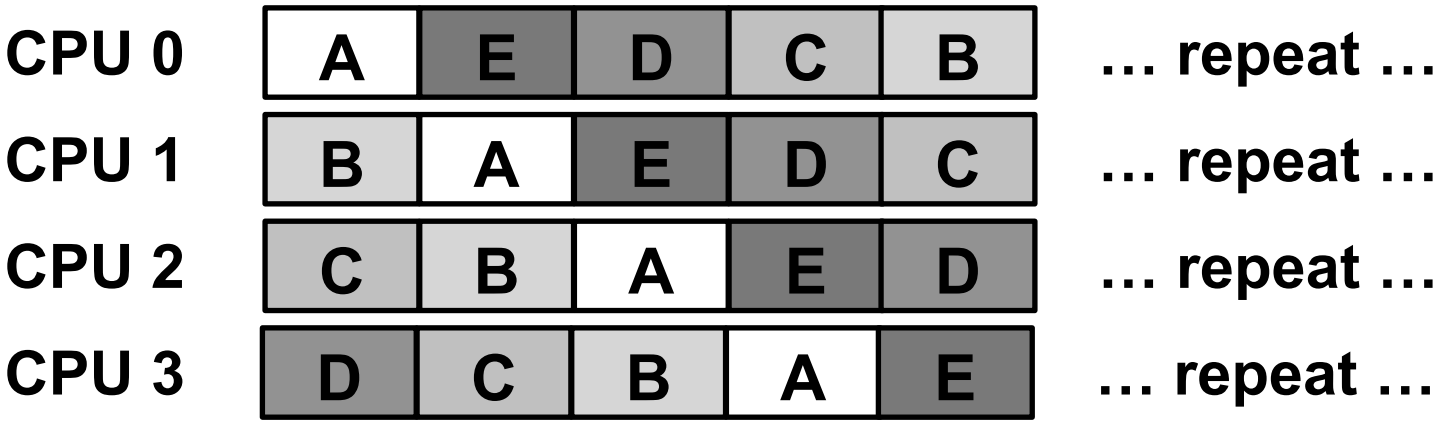


Outline

1. ~~Multi-Level Feedback Queue (OSTEP Chapter 8)~~
   - ~~Approach used in many real-world OSes, including Windows~~
2. Multi-Processor Scheduling
   - We'll only skim the surface here (OSTEP chapter 10 if you want more details)
3. What Linux does
   - We'll only skim the surface here as Linux scheduling has a long/complex history with many different approaches

Multi-Processor Scheduling

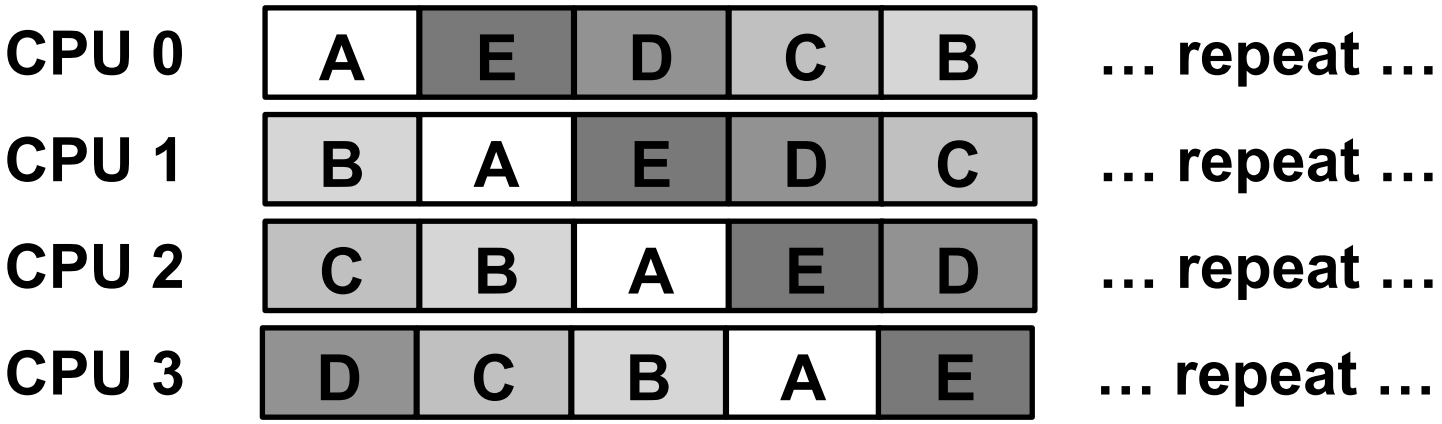
- All our processors are multi-core
- Therefore, OSes need to do scheduling across multiple cores
- You may be thinking: what's the big deal?
  - Whenever a core becomes idle, put a job on it based on whatever MLFQ scheme
- If we do this, we may end up with:



- Anybody sees a problem with this?

Multi-Processor Scheduling

- All our processors are multi-core
- Therefore, OSes need to do scheduling across multiple cores
- You may be thinking: what's the big deal?
  - Whenever a core becomes idle, put a job on it based on whatever MLFQ scheme
- If we do this, we may end up with:



- Anybody sees a problem with this?
- Having jobs jump around cores makes cache use inefficient!

Multi-core Architectures

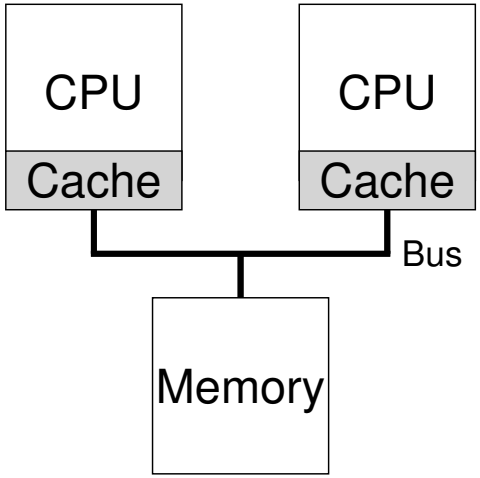
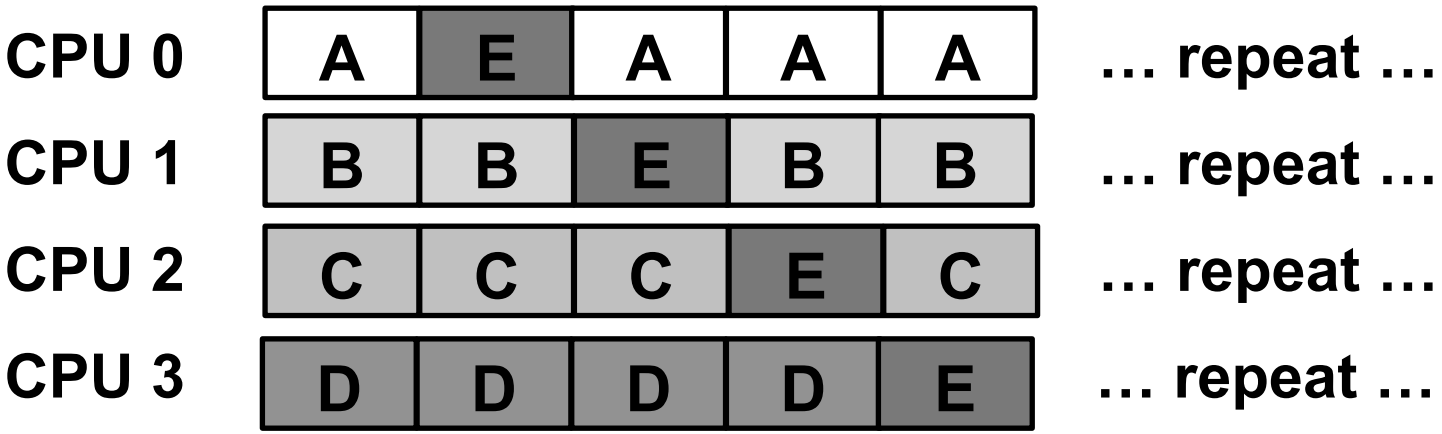


Figure 10.2: **Two CPUs With Caches Sharing Memory**

- A job runs on Core #1 and has its data in the cache
  - It experiences a lot of cache hits during its execution, because of spatial and temporal locality, which is great
- Then it does some I/O, ends up later in the Ready Queue, and then gets scheduled on Core #2
- At that point, it has none of its data in cache and will get many cache misses, which is terrible
- Worst case: jobs keep bouncing between cores as shown in the previous slide
  - Each time a job starts a time quantum, it's "lost" all its data in cache

Scheduling for Cache Affinity

- Each job has some affinity to some core: the core at which it has some/most data in cache
- Most modern OSes ensure that jobs are scheduled on cores while taking affinity into account, whenever possible, for example:



- Job E bounces around, but others stay put, which is good for the cache
- See OSTEP Chapter 10 for more details and fancy solutions (as other problems arise)

CPU Affinity in the Linux Kernel

- On our Docker container
  - Let's look in: **`/usr/src/linux-headers-5.15.0-25/include/linux/sched.h`**
    - Let's search for affinity-related functions
  - Let's look in: **`/usr/src/linux-headers-5.15.0-25/include/linux/cpumask.h`**
- We found out that the PCB includes a bit mask for all CPUs, which is likely used to keep track of current CPU affinity

Outline

1. ~~Multi-Level Feedback Queue (OSTEP Chapter 8)~~
   - ~~Approach used in many real-world OSes, including Windows~~
2. ~~Multi-Processor Scheduling~~
   - ~~We'll only skim the surface here (OSTEP chapter 10 if you want more details)~~
3. What Linux does
   - We'll only skim the surface here as Linux scheduling has a long/complex history with many different approaches

Linux Scheduling

- Linux is an OS for which there has been a lot of scheduling ideas and development
  - Linux is typically considered to have had very good schedulers
- Well-known scheduling algorithms:
  - O(1) Scheduler: multiple queues, bitmap tricks to make quick decisions, accounting of CPU usage by each job, akin to MLFQ
  - CFS (Completely Fair Scheduler): stores jobs in a red-black tree instead of queues, implements proportional-share approach for fairness (similar to what is described in OSTEP Chapter 9)
  - BFS (BF Scheduler): simple algorithm, single queue, also focused on fairness (never made it to the mainstream kernel releases)
- We don't have time to go into any of those, but A LOT of information is available on-line about this hotly debated topic
- The default has been CFS (since Kernel 2.6 - ca 2011)
  - Simpler/cleaner code than O(1) scheduler, which was the default before but had become really hard to maintain / evolve
  - BFS never made it to the mainstream, but is the default in a few distributions
- If you want to use another algorithm, you have to patch your kernel

Conclusion

- Scheduling is a very complex topic studied in many CS contexts
- In the context of OSes, there is a long history of scheduling ideas and implementations
- The common theme is to do Round-Robin and try to be fair among jobs while allowing some jobs to have high-priority (e.g., because they are interactive)
- Real-world OSes all use roughly the same ideas with tweaks, bells and whistles