Basics of CPU Scheduling

ICS332

Operating Systems

Henri Casanova (henric@hawaii.edu)

CPU Scheduling

- CPU Scheduling: The process by which the OS decides which processes/threads should run (and for how long)
  - Necessary in a multi-programming environment
  - Reminder: only READY processes/threads can be scheduled
- In these lecture notes, like in our and most textbooks, I will use the word job, which should be understood as “process or thread”
- When one says “scheduling” in the context of the OS, one typically means two things together:
  - The **scheduling policy**: “what the OS should do”
  - The **scheduling mechanism**: “how the OS does it”
- Separating policy and mechanism is the way to go!
  - Remember the discussion of this issue in the Overview and Interfaces module (Lecture notes: A Brief History of Operating Systems)

Policy and Mechanisms

- The policy: a scheduling strategy
  - The broad goal is to improve system performance and productivity, including:
    - Maximize CPU utilization (whenever the CPU idle, pick a READY jobs to run)
    - Make jobs "happy" (jobs should feel like they don't wait for the CPU too much, or too unfairly)
- The mechanism: the dispatcher
  - The OS component that knows how to switch between jobs on the CPU (it implements context-switching)
  - It must be fast (i.e., low dispatcher latency)
- There are strong theoretical underpinnings and a huge literature on the topic of scheduling but we will focus on pragmatic issues

Long-term/Short-term Scheduling

- Long-term Scheduling
- Select jobs from a submitted pool of jobs and loads them into memory
- Orchestrate job executions in the long term O(minutes), O(hours)
- Make scheduling decisions every O(minutes), O(hours)
- Can construct complex schedules
- Can use sophisticated decision algorithms (ics632)

Long-term/Short-term Scheduling

- Long-term Scheduling
- Select jobs from a submitted pool of jobs and loads them into memory
- Orchestrate job executions in the long term O(minutes), O(hours)
- Make scheduling decisions every O(minutes), O(hours)
- Can construct complex schedules
- Can use sophisticated decision algorithms (ics632)

- Short-term Scheduling
- Select already-in-memory jobs to run
- Orchestrate job executions in the very short term O(milliseconds)
- Make scheduling decisions every O(milliseconds)
- Cannot make complex decisions
- Use simple decision algorithms (these lecture notes)

Long-term/Short-term Scheduling

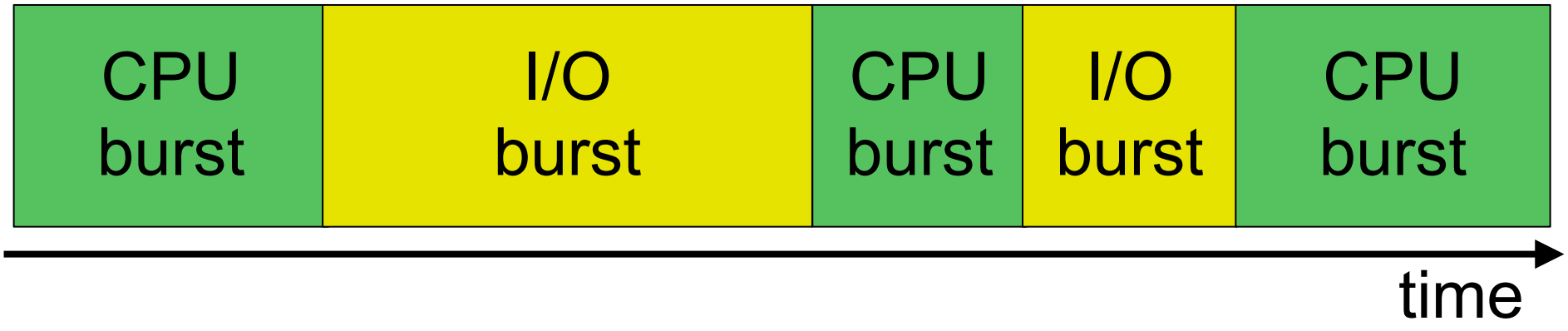
- Long-term Scheduling
- Select jobs from a submitted pool of jobs and loads them into memory
- Orchestrate job executions in the long term O(minutes), O(hours)
- Make scheduling decisions every O(minutes), O(hours)
- Can construct complex schedules
- Can use sophisticated decision algorithms (ics632)

- Short-term Scheduling
- Select already-in-memory jobs to run
- Orchestrate job executions in the very short term O(milliseconds)
- Make scheduling decisions every O(milliseconds)
- Cannot make complex decisions
- Use simple decision algorithms (these lecture notes)

- **OSes do short-term scheduling**
- Long-term scheduling done by non-OS software (e.g., batch scheduler)

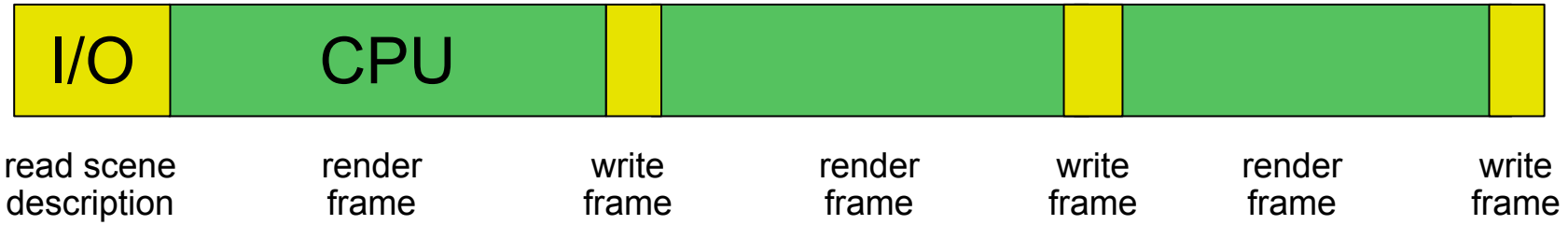
CPU and I/O Bursts

- Most jobs alternate between CPU bursts and I/O bursts
- CPU bursts: a sequence of CPU instructions that do whatever computation until a syscall is placed to do some “I/O” (actual I/O, waiting for something, etc.)
- I/O bursts: a time during which the job cannot execute its next instruction because it’s waiting for the “I/O” to complete. The job is not running and is in the “blocked/waiting” state.

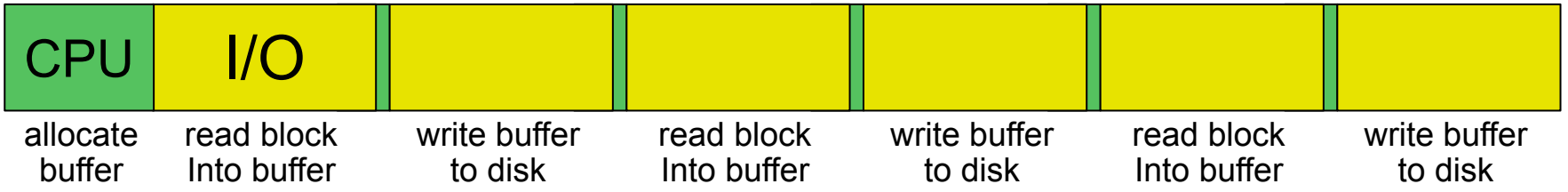


CPU- or I/O-bound jobs

- CPU-bound job: a job that mostly uses the CPU with mostly (possibly many) short or few I/O bursts
  - e.g., a raytracer to render 3-D scenes



- I/O-bound job: a job that mostly waits for I/O with mostly (possibly many) short or few I/O bursts
  - e.g., a program that copies a file (`/bin/cp`)



The CPU Scheduling Challenge

- The OS keeps ready jobs (their PCBs) in a list: the Ready Queue
- The OS must make scheduling decisions
  - Pick one of the jobs from the Ready Queue
  - Decide how long to give the CPU to the picked job
- The big challenge is that the only thing the OS knows is the number of jobs in the system, but the job population can be very diverse
  - Some jobs are I/O-bound, some are CPU-bound, some are in between, some change behavior after some point
  - Some jobs will run for 0.001 sec and terminate, some jobs will run for hours
- When you start a job, you don't tell the OS: "Hi OS! This is a mostly I/O-bound job at first that will run for 10 minutes, and then will become CPU-bound and run for 20 minutes"
- Instead, you hit <enter>, double-click, call fork(), create a thread, and here is a new job that the OS knows almost nothing about that it should make scheduling decisions for

The Scheduling Problem

- Let's take a VERY simplified/unrealistic view of the scheduling problem
  - 1. All jobs arrive at the "same time" (within an epsilon)
  - 2. Each job runs for the same amount of time
  - 3. Once started, each job runs to completion
  - 4. All jobs are 100% CPU-bound (no I/O)
  - 5. The run-time of each job is known
- We will remove assumptions as we go along…

- **Question:** What do we want to achieve?
- Which is really: which metric do we want to optimize?
- A very natural metric is average job turnaround time

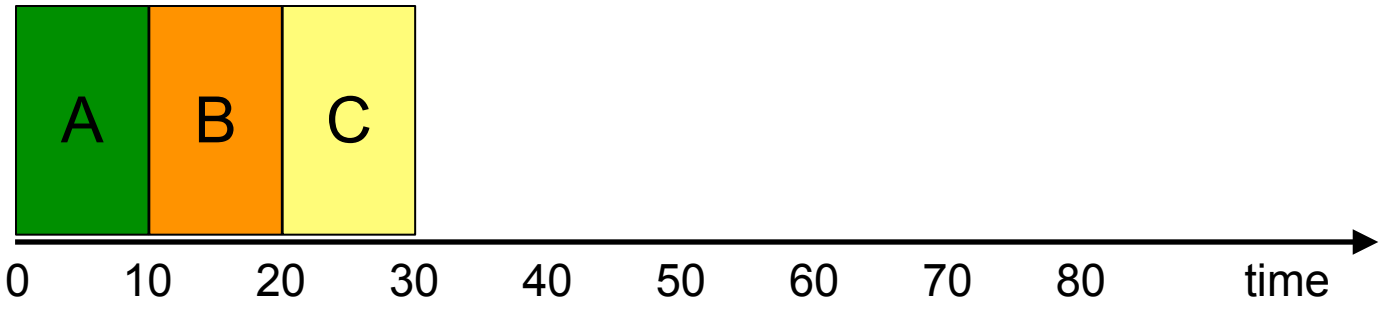
Job Turnaround Time

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

- Intuitively pretty obvious that owners of CPU-bound jobs care about turnaround time
- But for other jobs perhaps this metric makes little sense
  - e.g., your Web browser!
- There are many, many other possible metrics
  - e.g., metrics that capture notions of fairness among jobs

Simplest Algorithm: FCFS

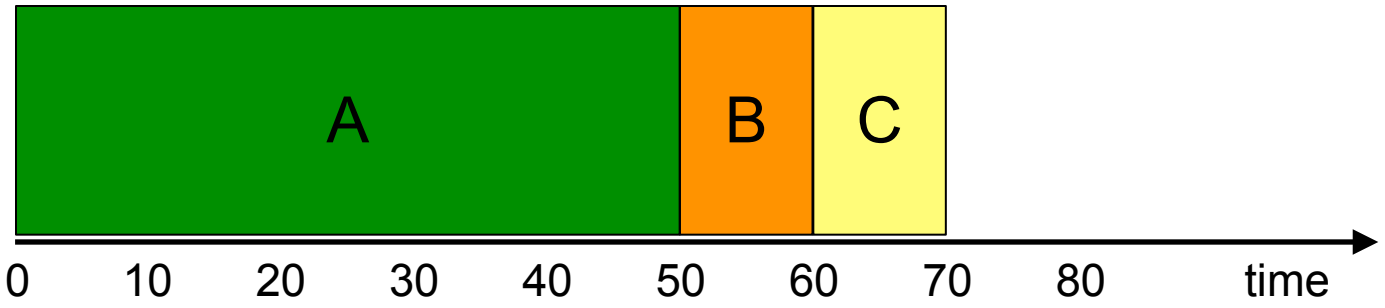
- First Come First Serve
- Say we have 3 jobs, each wanting to run for 10 seconds
- These jobs are in some order in the Ready Queue, say: A, B, C



- The average turnaround time is: ((10-0)+(20-0)+(30-0)) / 3 = 20
- This is 2x the average job duration, but that makes sense because we have competition among the jobs for the CPU

FCFS: Convoy Effect

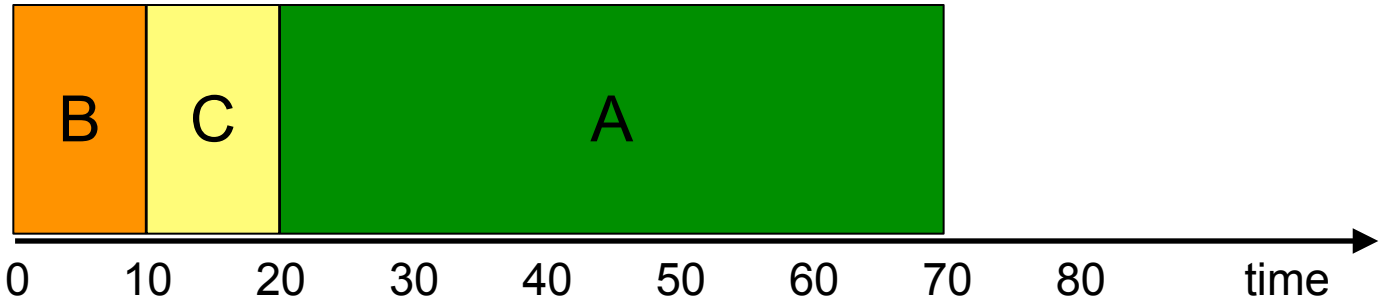
- Let's remove one assumption: job A takes 50 seconds



- The average turnaround time is: (50+60+70) / 3 = 60
- This is ~2.57x the average job duration (and we can make this arbitrarily bad of course, e.g.,if job A takes 100000 seconds)
- Short jobs can be "stuck" behind long jobs
- This convoy effect is basically: you want to buy a banana at the supermarket, but you arrive at the checkout at the "same time plus some epsilon" as somebody who has a cart with 100 items

Another Idea: Shortest Job First (SJF)

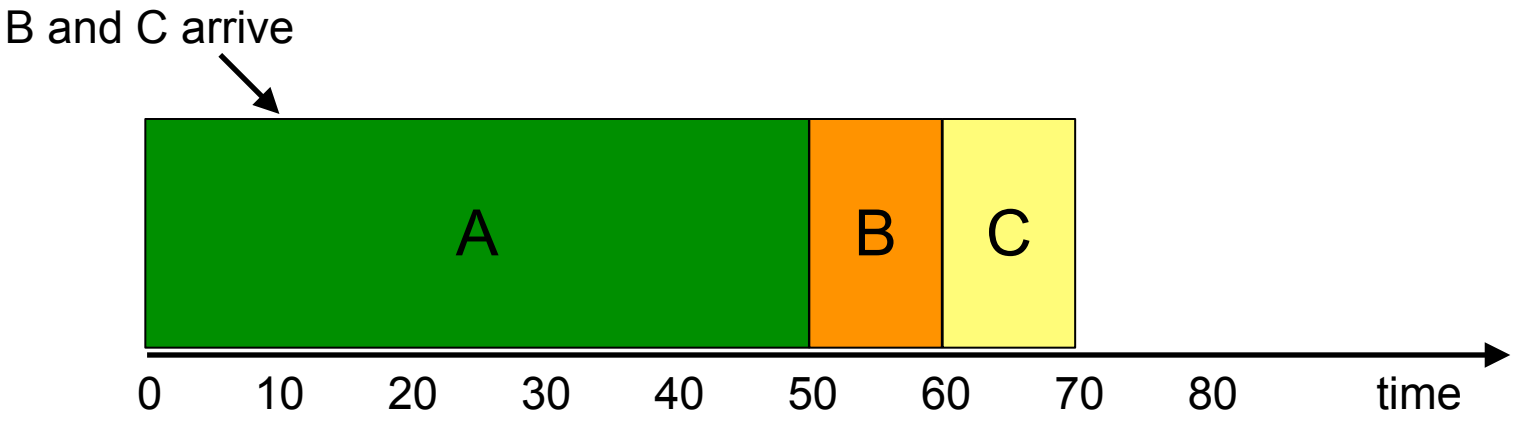
- Dispatch short jobs first, which would give us



- The average turnaround time is: (10+20+70) / 3 = 33.3
- This is ~1.42x the average job duration, which is much better than with FCFS
- It turns out this SJF is optimal with our assumptions and our goal to optimize the average turnaround time
  - There is an elegant proof by contradiction based on assuming that an optimal schedule isn't in the SJF order, swapping the order of two jobs, and showing that the new schedule is better, meaning that the previous one wasn't optimal
- What happens if jobs can arrive at different times?

SJF with Jobs Arriving at Different Times

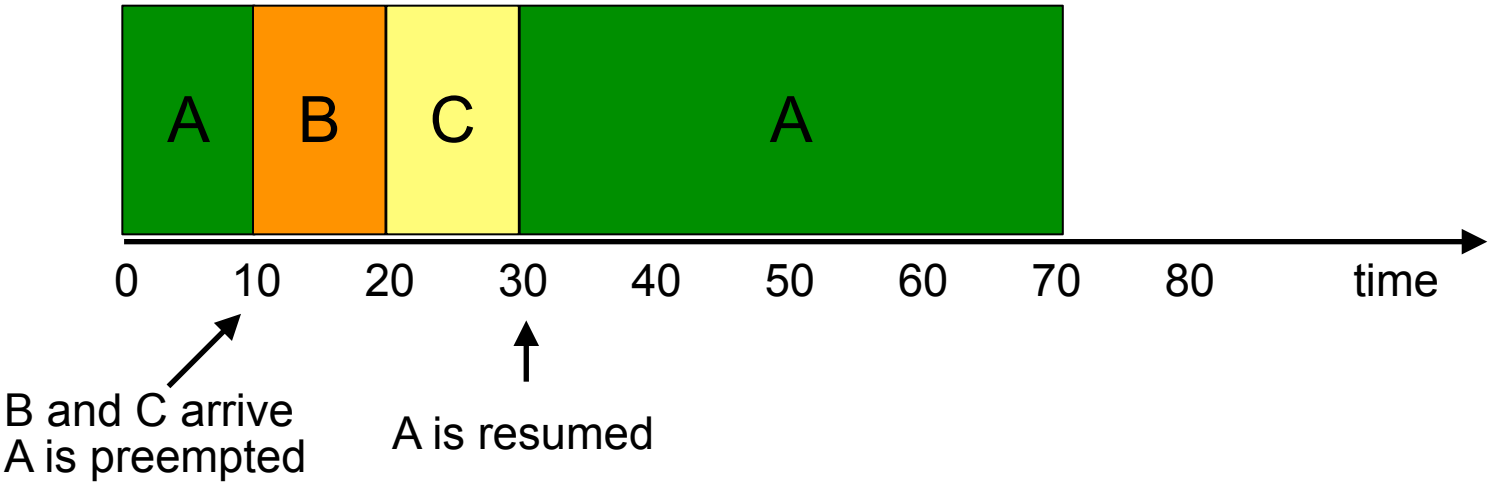
- Say A arrives at time 0, and B and C arrive at time 10
- At time 0 the OS has to schedule job A (it's the only one in the Ready Queue, and as far as it knows, no other job may arrive for the next decade)



- The average turnaround time is: ((50 - 0) + (60 - 10) + (70 - 10)) / 3 = 53.33
- This is ~2.28x the average job duration
- Jobs B and C are very "unhappy" as we still have a convoy effect
  - Arriving to the checkout lane 10 seconds after a person with a full cart is still bad
- What can we do?

Preempting Jobs

- We need to remove the “Once started a job runs to completion” assumption
- The OS must be able to **preempt** a job (kick it out of the CPU)
- A simple algorithm: Shortest Time-to-Completion First (STCF)
  - Whenever a new job arrives, if it has shorter time-to-completion than the running job, then preempt the running job and run the new job
  - Whenever a job finishes, run the job with the shortest time-to-completion



- The average turnaround time is: ((70 - 0) + (20 - 10) + (30 - 10)) / 3 = 33.33
- This is ~1.42x the average job duration
- We fixed the convoy problem (not happening at the supermarket 😕)

We Could Keep Going…

- Turns out that STCF is optimal for average turn-around time with the assumption that jobs can be preempted
  - Just like SJF is optimal for average turn-around time when jobs cannot be preempted
- But now we have a starvation problem: a job may never run
- Anybody sees why?
- A stream of “short” jobs can forever overtake a long job!
  - e.g., Somebody with a full cart at the supermarket is in line, and every second somebody arrives to buy just a banana, which takes one second, forever

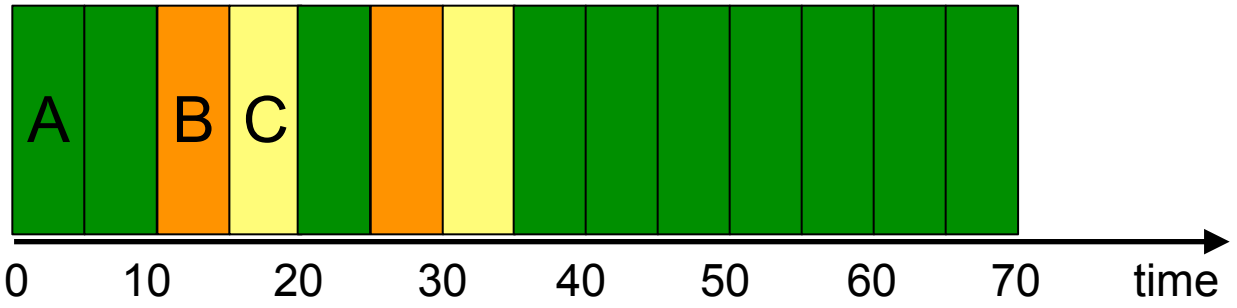
- The scheduling problem is just very hard
- It is a long-standing and still very active area of research in Computer Science
- Most of this research, however, is not at the OS level because…

The OS Does not Know Job Durations!

- But remember, you don't tell the OS how long your program is going to run when you start it
  - Unlike in a batch system, like on "supercomputers" in which you do, and if your job runs longer then it's killed!
  - We could design OSes like this, but people would hate them immediately
- So the "nice" algorithms we just saw like SJF or STCF cannot be implemented in the OS
  - Just like most sophisticated algorithms in the scheduling research literature
- One important lesson from these algorithms, though, is that job preemption is necessary to avoid the convoy effect
- So what can the OS do?
- If you don't know how long jobs take, you run each job for a little bit, preempt it, run another job, and keep going forever!

Round-Robin (RR) Scheduling

- What OSes do is Round-Robin Scheduling: Let a job run for at most a little bit, then preempt it, then run the next job, etc.
- The (maximum) amount of time a job runs for before preemption is called the time quantum (or scheduling quantum, or time slice)
- Same Example as for STCF in a previous slide:
  - Job A (50sec) arrives at time 0, Jobs B (10sec) and C (10sec) arrive at time 10
  - **Say the time quantum is 5s**



- Average turnaround time: ((70 - 0) + (30 - 10) + (35 - 10)) / 3 = 38.33
- 1.64x the average job duration
- Compared to 1.42x with STCF if the OS had known (magically?) all job durations - **not bad at all!**

RR Time Quantum

- Key question: How to pick the time quantum?
- **Time Quantum too large:** poor response time
  - A job may start its first time quantum a “long” time after it arrived into the system, which will then cause it to have a poor response time
  - The larger the time quantum, the closer RR scheduling gets to being FCFS scheduling!
- **Time Quantum too small:** too much context-switching overhead
  - Say you have a bunch of CPU-intensive jobs, a 9 ms time quantum, and a 1ms context switching overhead
  - Then your CPU spends 10% of its time doing context-switches (instead of doing useful work)!
  - This is why OS developers put a lot of working into making context-switching as fast as possible
    - So that time quanta can be relatively small, within limits

The Story So Far

- Turn-around time seems like a reasonable metric
- We can have algorithms that do ok for that metric
- Especially if we allow jobs to be preempted
- But turn-around time is impossible to optimize for if we don't know the job durations!
- So we instead use RR Scheduling

- Bottom Line: All OS Schedulers use flavors of RR Scheduling
- Let's look at our assumptions again….

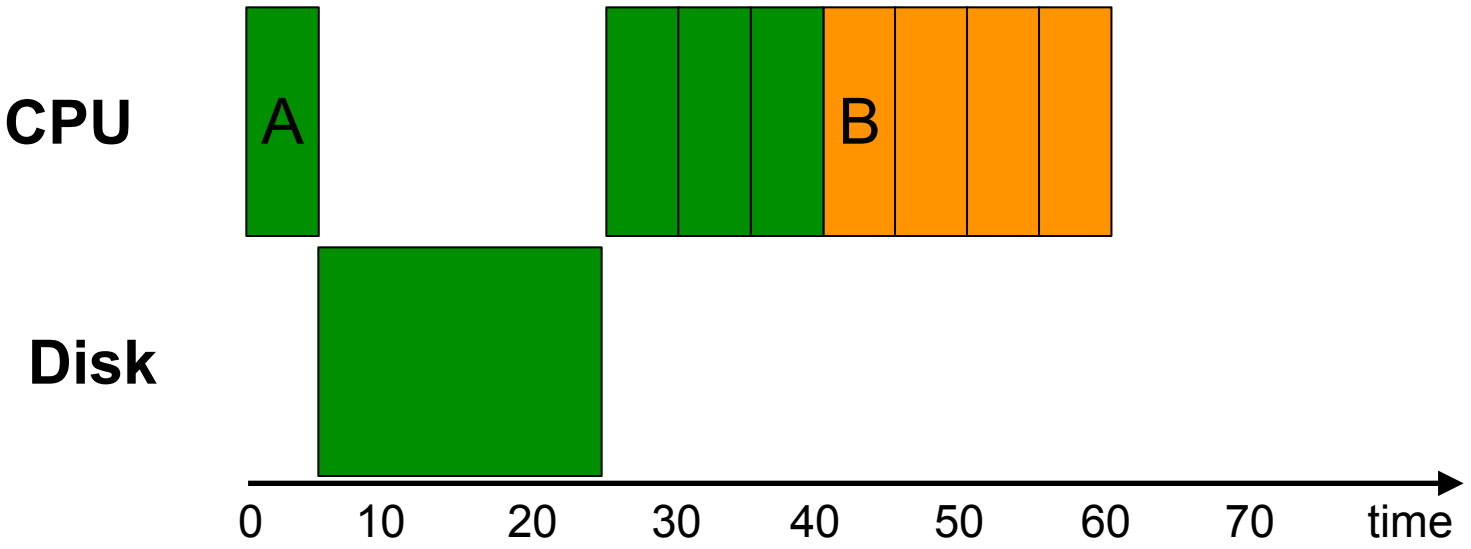
Back to Our Assumptions

- Our original assumptions
  - ~~1. All jobs arrive at the "same time" (within an epsilon)~~
  - ~~2. Each job runs for the same amount of time~~
  - ~~3. Once started, each job runs to completion~~
  - 4. All jobs are 100% CPU-bound (no I/O)
  - ~~5. The run-time of each job is known~~

- Let's now remove assumption #4: Each job alternates between CPU bursts and I/O bursts

One Option: Do Nothing

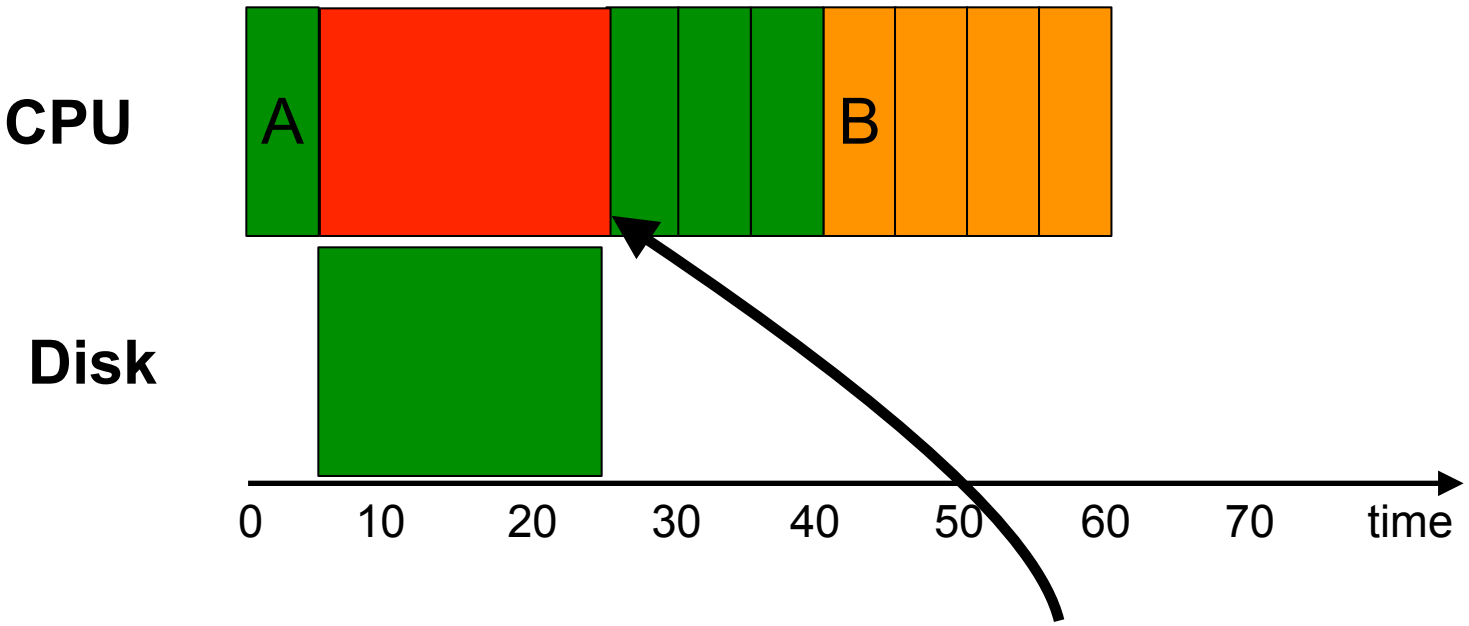
- The OS could just not care, and let a job do its I/O (after all, the time quantum is about CPU usage, and I/O doesn't use the CPU)
- Example with two jobs, time quantum of 20 sec



- This is terrible, anybody sees why?

One Option: Do Nothing

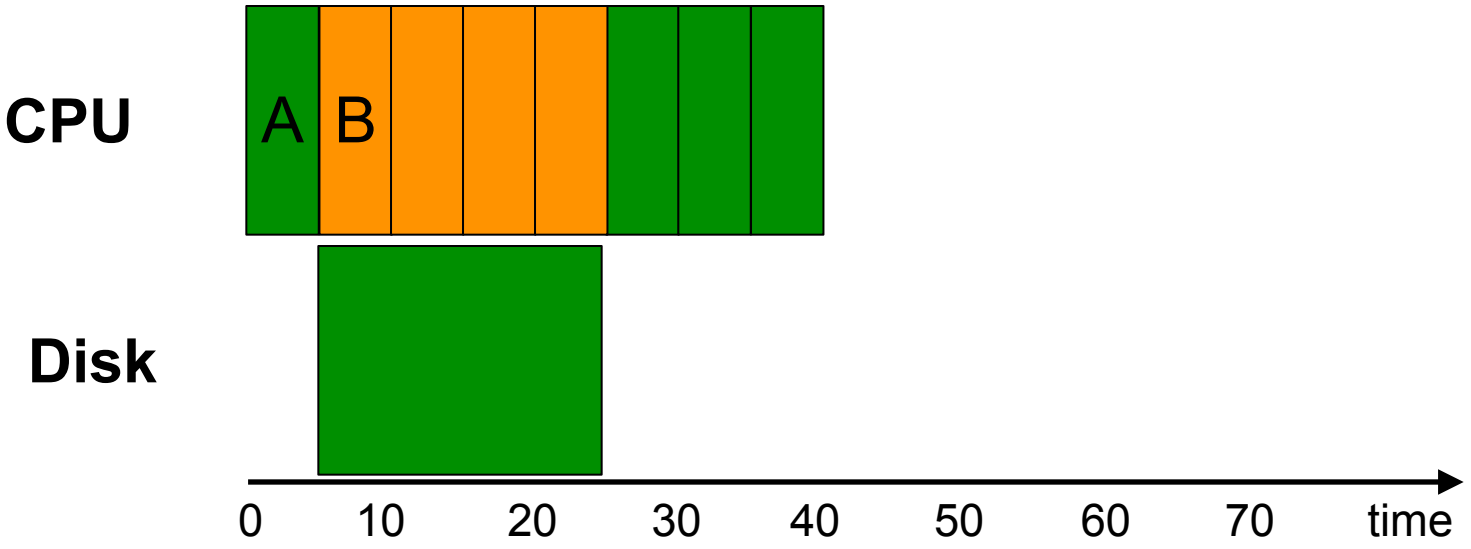
- The OS could just not care, and let a job do its I/O (after all, the time quantum is about CPU usage, and I/O doesn't use the CPU)
- Example with two jobs, time quantum of 20 sec



- Horrible waste of the CPU: stays **idle** during I/O

Scheduling Around I/O?

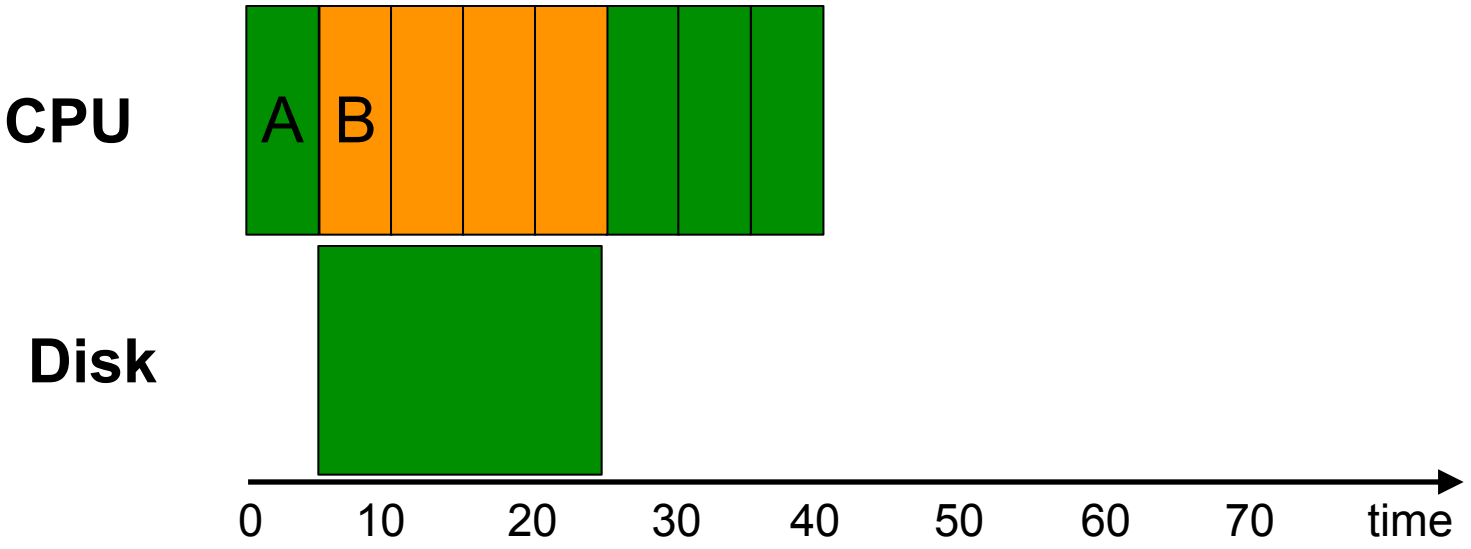
- A much better schedule would be:



- This is what Oses try to achieve
  - In this example it works beautifully because A does 20 seconds of I/O, which is exactly the time quantum duration
  - It's not always this "clean"

Scheduling Around I/O?

- A much better schedule would be:



- **When a Job does I/O, it vacates the CPU**
- **When the I/O is done, the Job is placed in (the back of) the Ready Queue, and will be scheduled as normal in the future**

In-class Exercise

- Consider the following jobs:
  - A: CPU burst time 3ms, I/O burst time 3ms
  - B: CPU burst time 6ms, I/O burst time 3ms
  - Both jobs alternate forever between CPU and I/O bursts, starting with a CPU burst
  - I/O activities from the jobs don't impact each other. Say A uses the Disk and B uses the NIC
- At time t = 0, both are in the ready queue (A is first)
- The OS uses Round-Robin Scheduling with a **time quantum of 5ms**

- **Question #1:** "Plot" the CPU utilization time-line for 29ms
  - Use letter A for 1ms of execution of job A
  - Use letter B for 1ms of execution of job B
  - Use letter I for 1ms of idle time
  - Example "plot": **`AABIIIAAAABIIB`**…
- **Question #2:** On average, in the long run, how much time does A spend in the Ready state each time it becomes Ready?

Solution

- A: CPU=3ms, I/O=3ms; B: CPU=6ms, I/O=3ms
- Time Quantum = 5ms

CPU **AAA**

A's 1st time quantum, which is not fully utilized because A's CPU burst is smaller than the time quantum duration

Disk

NIC

Solution

- A: CPU=3ms, I/O=3ms; B: CPU=6ms, I/O=3ms
- Time Quantum = 5ms

CPU **AAABBBBB**

B's 1st time quantum, which is fully utilized because B's CPU burst is smaller than the time quantum duration, and B isn't done with its first CPU burst!

Disk **AAA**

A's 1st I/O burst, at the end of which A goes back to the Ready Queue

NIC

Solution

- A: CPU=3ms, I/O=3ms; B: CPU=6ms, I/O=3ms
- Time Quantum = 5ms

CPU **AAABBBBBAAA**

A's 2nd time quantum

Disk **AAA**

NIC

Solution

- A: CPU=3ms, I/O=3ms; B: CPU=6ms, I/O=3ms
- Time Quantum = 5ms

CPU **AAABBBBBAAAB**

B's 2nd time quantum, which is not fully utilized because there is only 1sec left to do in B's first CPU burst

Disk **AAA** **AAA**

NIC

A's 2st I/O burst, at the end of which A goes back to the Ready Queue

Solution

- A: CPU=3ms, I/O=3ms; B: CPU=6ms, I/O=3ms
- Time Quantum = 5ms

CPU **AAABBBBBAAABII**

2ms of Idle time because both jobs are doing I/O!

Disk **AAA** **AAA**

NIC **BBB**

B's 1st I/O burst, at the end of which B goes back to the Ready Queue

Solution

- A: CPU=3ms, I/O=3ms; B: CPU=6ms, I/O=3ms
- Time Quantum = 5ms

CPU **AAABBBBBAAABIIAAA**

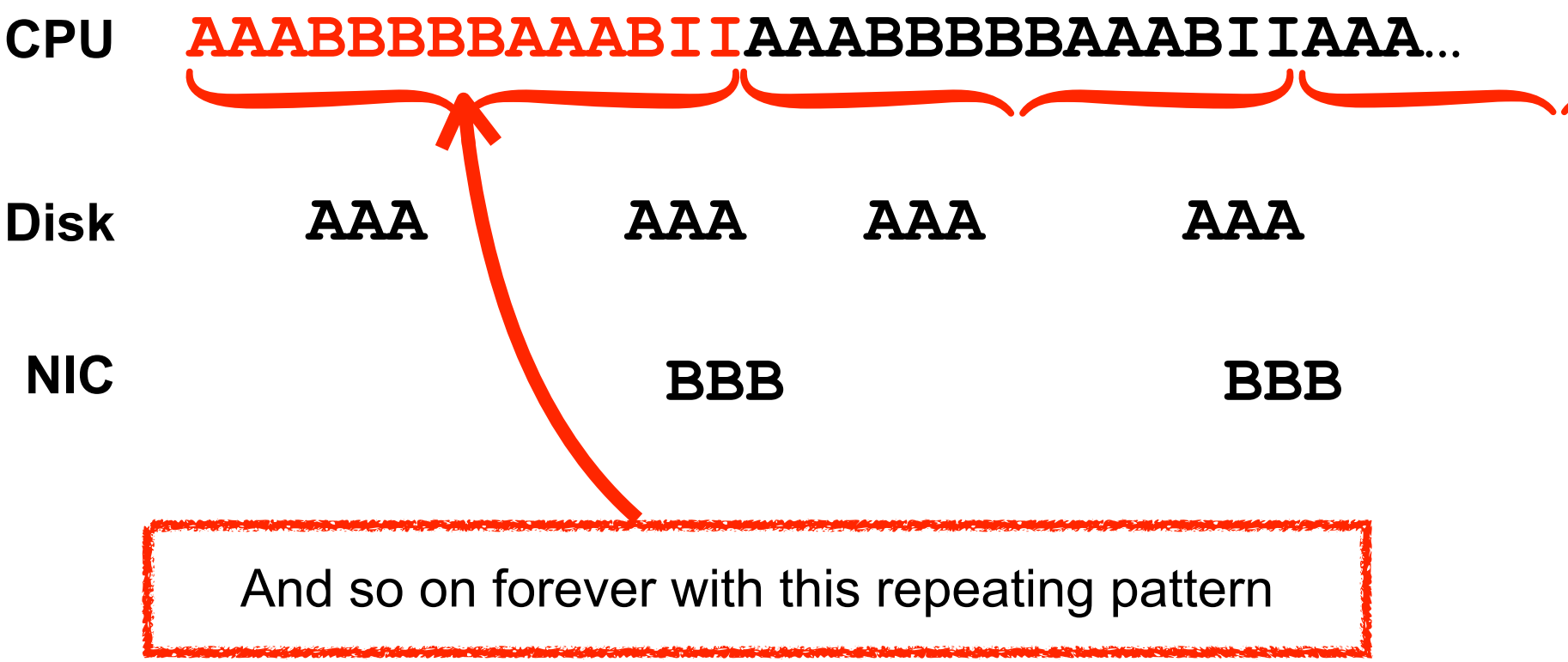
A's 3rd time quantum

Disk **AAA** **AAA**

NIC **BBB**

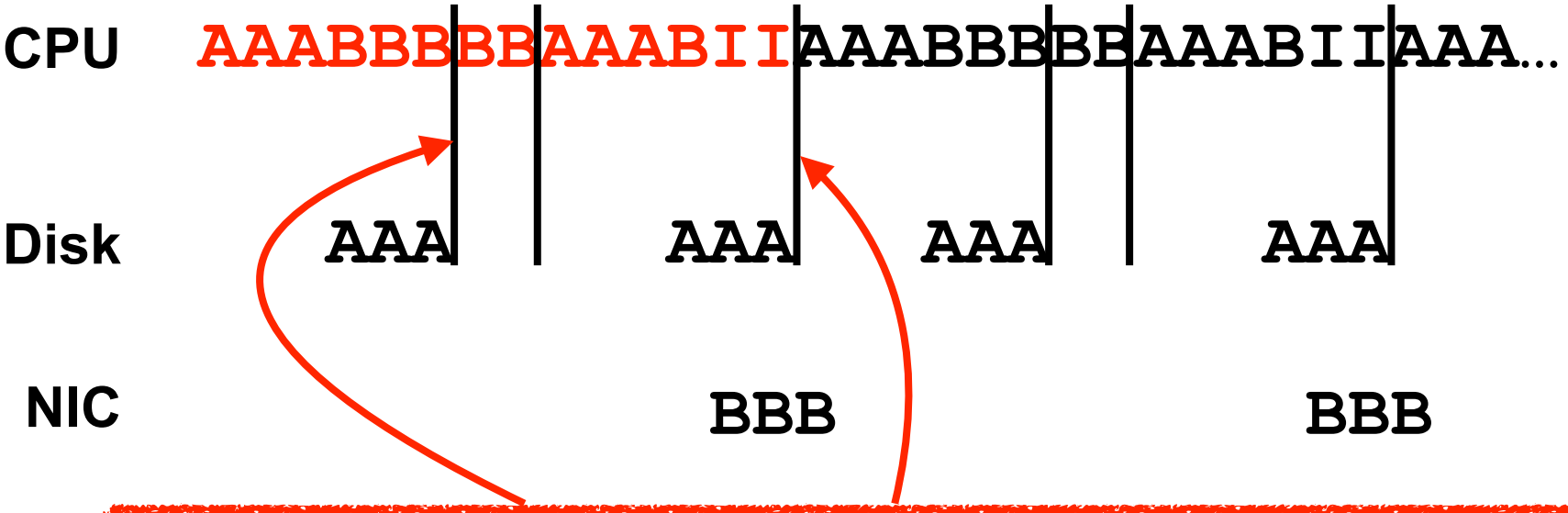
Solution

- A: CPU=3ms, I/O=3ms; B: CPU=6ms, I/O=3ms
- Time Quantum = 5ms



Solution

- A: CPU=3ms, I/O=3ms; B: CPU=6ms, I/O=3ms
- Time Quantum = 5ms



When A becomes ready, every other time it can't run for 2 sec, and every other time it runs immediately. On average it spends 1ms in the Ready State

Conclusion

- Based on what we've said so far, designing an OS scheduler is easy?
  - 1. Pick a time-quantum (based on how fast context-switching is)
  - 2. Do Round-Robin scheduling
  - 3. Treat each CPU burst as a job and kick jobs off the CPU as soon as they do I/O
- This is the basics
- But turns out, if this is all you do in your OS scheduler, things are not going to be great for users in practice
- See the next set of lecture notes….