CS 326: Operating Systems

CPU Scheduling

Lecture 10

Today's ~~Schedule~~ Agenda?

- Context Switches and Interrupts
- Basic Scheduling Algorithms
- Scheduling with I/O
- Symmetric Multiprocessing

Today's ~~Schedule~~ Agenda?

- **Context Switches and Interrupts**
- Basic Scheduling Algorithms
- Scheduling with I/O
- Symmetric Multiprocessing

Context Switching

- Changing execution from process to process requires **context switching**
  - Saving and restoring the process control block
- You may wonder if transitioning between user and kernel space requires a context switch
  - In general, no
  - Only a privilege change occurs
    - Ring 3 > Ring 0; or: User Mode > Superuser Mode

Reminder: Process Control Block

- Executable machine code
- Memory (isolated from other processes)
  - Call stack
  - Heap
- Metadata
  - Permissions, ownership
  - Open files and access isolation
  - Environment variables
- Context
  - Registers
  - Stack pointer, program counter
  - Memory addressing

When to Switch

- How do we know when to context switch? Programs don’t have to do it…
- Accomplished via interrupts
- The OS configures a hardware timer to fire on a set interval:
  - “Interrupt CPU in **X** *ms*”

Interrupt Handling

- The CPU receives a hardware event every X ms
  - Trap table is used to determine what to do (what code to run) when the hardware timer fires
- This moves us back into kernel space
- Then we can save the current PCB and replace it with the next in the **run queue**
  - This is called **preemptive multitasking**
    - Processes do not have to explicitly yield control to others!

Basic Scheduling

- Vintage OS often used a very basic form of scheduling:
  - Running tasks sequentially (mainframes! punchcards!)
  - Cooperative multitasking
- We can implement our very own user space “thread library” by context switching within our own programs
  - Done by runtimes such as Go

Making Decisions

- How do we decide what to run next?
- Several **scheduling algorithms** exist, all with different run time properties
- Maybe you want your algorithm to be fair to all processes?
- …or certain processes may get higher priority than others

Today's ~~Schedule~~ Agenda?

- Context Switches and Interrupts
- **Basic Scheduling Algorithms**
- Scheduling with I/O
- Symmetric Multiprocessing

Scheduling Concerns

- How long each process runs for
  - Scheduling **quantum**
- When jobs start or stop
  - At boot: lots of processes starting at once
  - During run time: less frequent start/stop
- Whether the process performs I/O
- Process priorities
  - Kernel threads, multimedia, games, etc.

Scheduling Metrics

- CPU Utilization
- Throughput
- Turnaround time
- Response time

Scheduling Metrics

- **CPU Utilization**
  - Amount of time the CPU spends doing productive work
    - A bad scheduler won't keep the CPU pipeline full of tasks
  - Utilization: $1 - t_{idle}$
- Throughput
- Turnaround Time
- Response Time

Scheduling Metrics

- CPU Utilization
- **Throughput**
  - Number of processes that are completed per time unit: $(\frac{completed}{t})$
  - For instance: 54 processes per second
- Turnaround Time
- Response Time

Scheduling Metrics

- CPU Utilization
- Throughput
- **Turnaround Time**
  - Amount of time from process arrival to process completion
  - $t_{turnaround} = t_{completion} - t_{arrival}$
- Response Time

Scheduling Metrics

- CPU Utilization
- Throughput
- Turnaround Time
- **Response Time**
  - How quickly a process starts running after it arrives (submitted/launched)
  - $t_{response} = t_{start} - t_{arrival}$

Demo: a “scheduler”

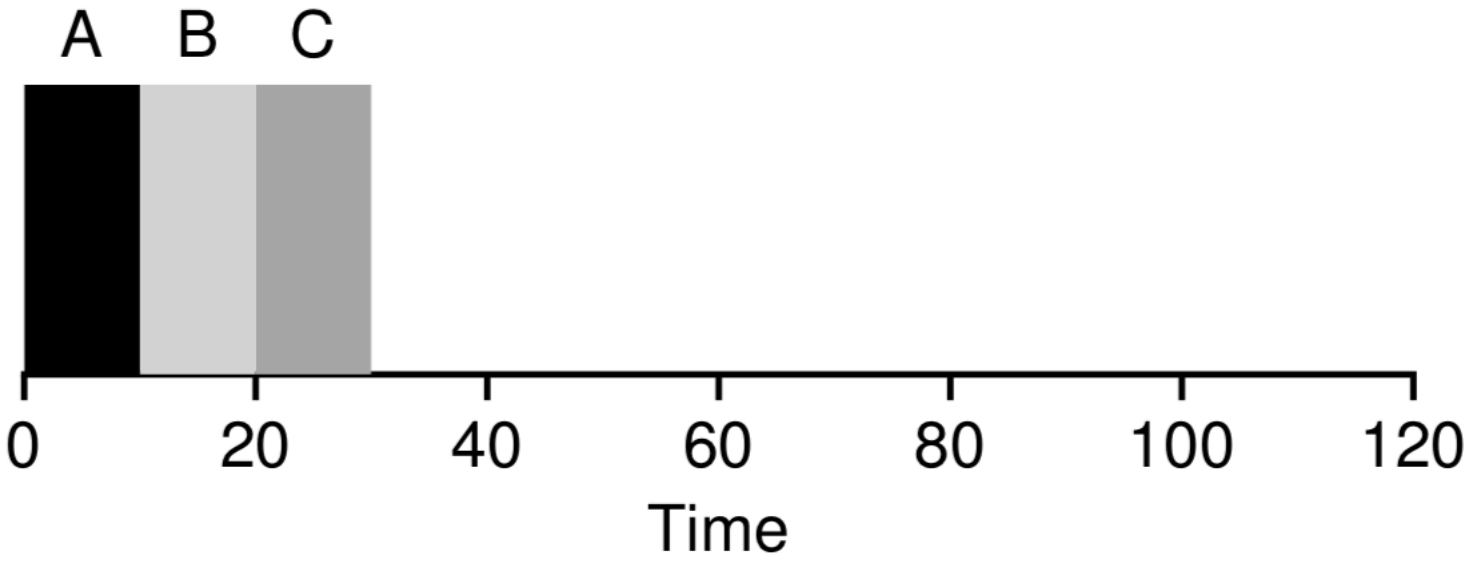
Basic Scheduling Algorithms

- FIFO
- SJF
- STCF
- Round Robin
- Priority
- Lottery

FIFO

- The first in, first out (FIFO) scheduler works just like a queue data structure
- Processes are executed in the order they arrive:
  - If A arrives, followed by B:
    1. A executes completely
    2. B executes completely
- Works reasonably well IF the processes run for about the same amount of time

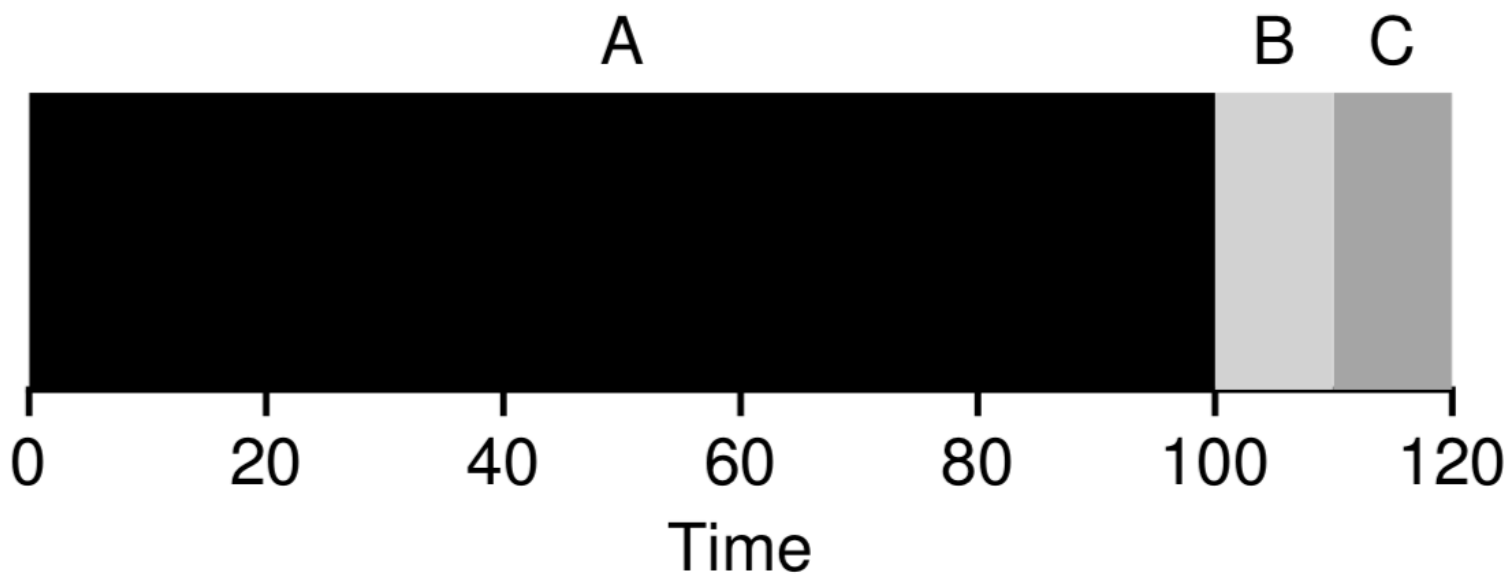
FIFO: Uniform Run Times



FIFO Weaknesses

- If one task takes a very long time, the other (shorter) tasks will have to wait, and wait, and wait…
  - This is called the **convoy effect**
- Here, the average process completion time is low
  - A process may only need 2 seconds to run, but waits for 20 seconds before it can even start

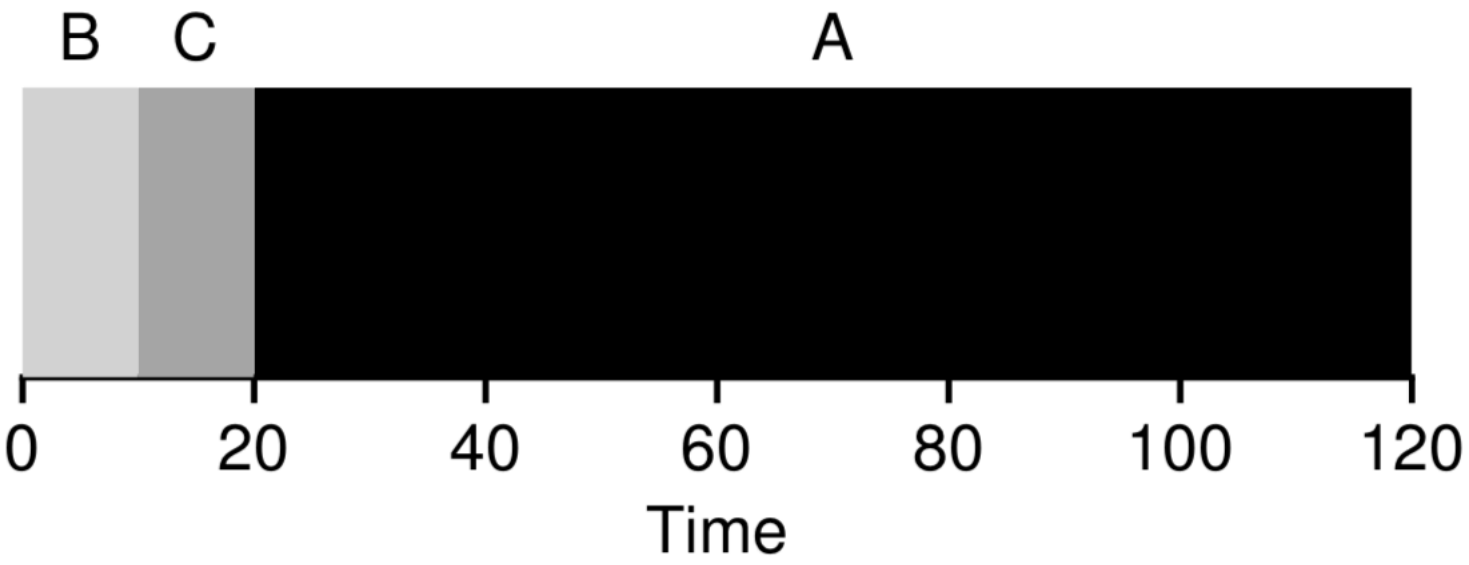
Convoy Effect



SJF

- With shortest job first (SJF) scheduling, we don’t have to worry about long tasks causing a convoy
- Assuming all the jobs are scheduled at the same time, the smallest will be executed first
  - Little jobs get finished quickly
  - Long jobs are delayed, but the impact is not as noticeable

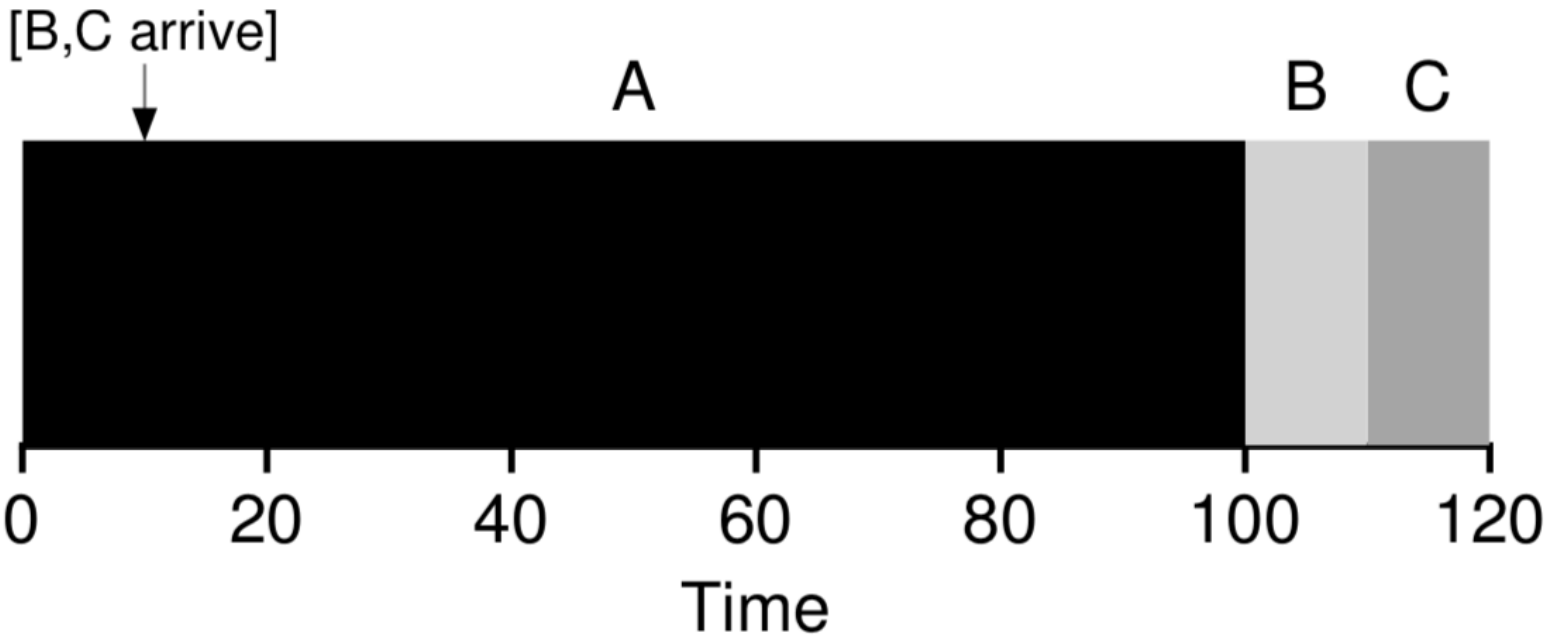
SJF: Prioritize Small Tasks



SJF Weaknesses

- First of all, how can we really know how long a job will take ahead of time?
  - This research question is still relevant today
- If your OS routinely runs lots of small tasks and one large task all at the same time, the larger task will always have to wait
- Once the large task is running, we'll still have a convoy effect if we schedule new, smaller tasks

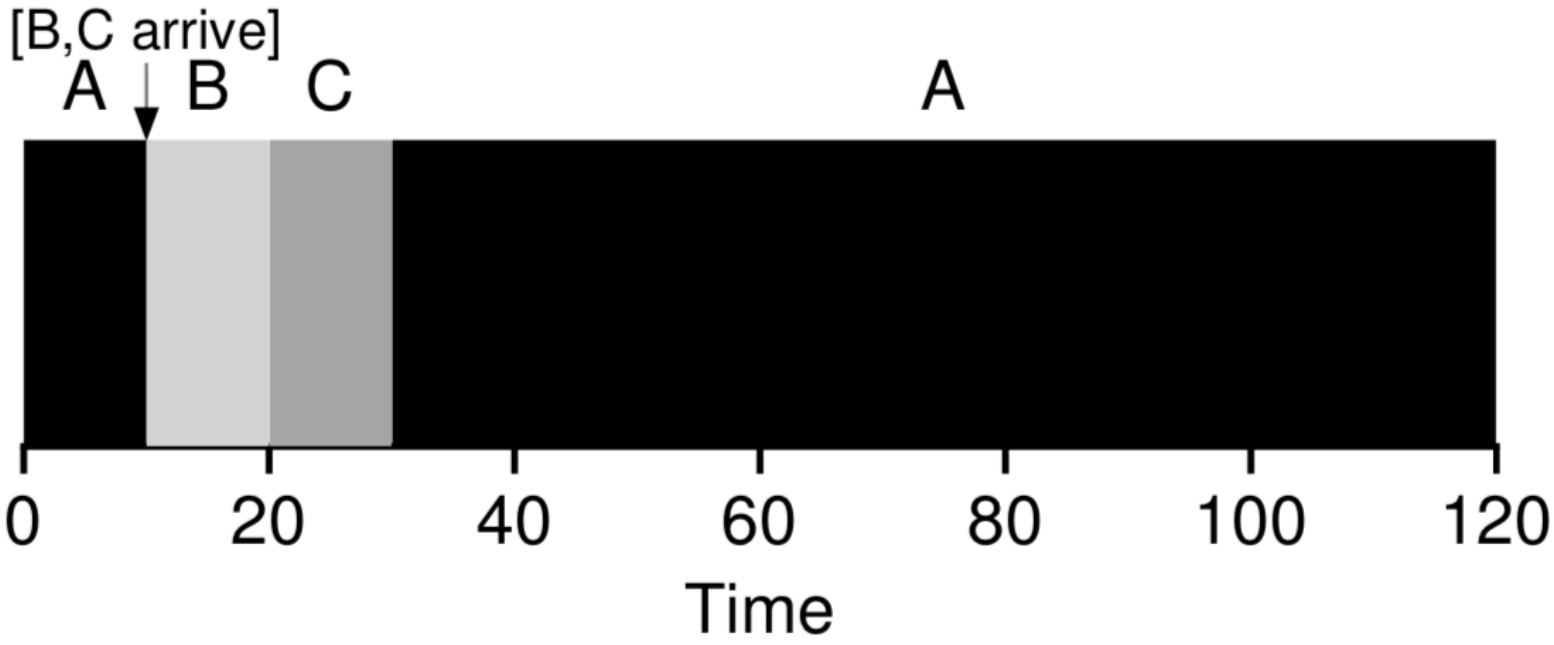
SJF: Late Arrivals



STCF

- In the shortest time to completion first (STCF) algorithm, we introduce **preemption**
  - Run whichever task will complete first and preempt running processes that don’t meet the criteria
  - In other words, this is SJF with preemption!
- Now the large task causing the convoy effect can be swapped out when small jobs arrive
- This keeps average turnaround time high: most small tasks will be taken care of quickly

STCF: Preemption



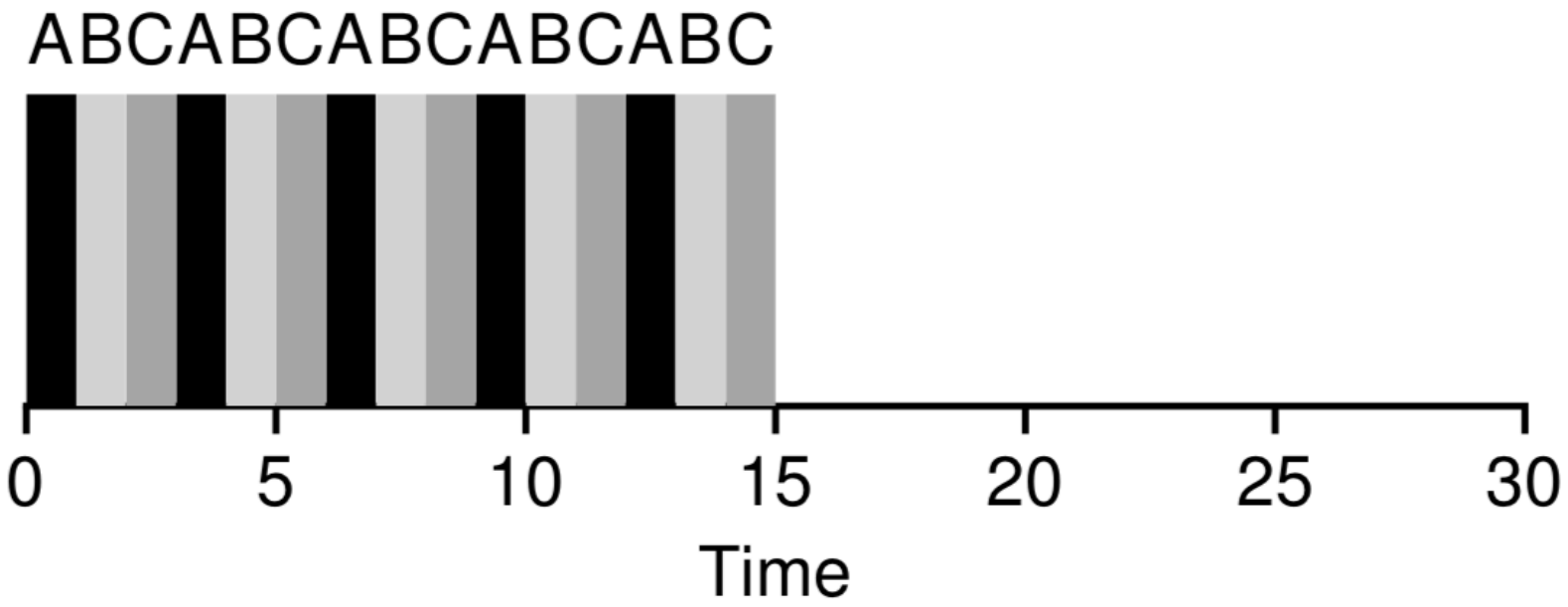
STCF Weaknesses

- STCF de-prioritizes large tasks the most
- Poor large tasks! 😢
- We also have a problem with resource starvation
  - If the OS continuously receives small tasks, the large task may never get a turn to execute

Round Robin

- Round robin scheduling tries to be as fair as possible
- Using preemption, all processes get a scheduling quantum (time slice)
- If I have 10 processes and a quantum of 1ms, then I'll run each process for 1ms for each iteration of the algorithm
  - After one pass completes, start back over again

Round Robin: Response Time



Round Robin Weaknesses

- While RR is fair, the scheduling quantum has a big impact on performance
  - Too small? Not much actual work getting done, most effort expended on context switching
  - Too large? Programs aren't responsive
- Exhibits poor turnaround time
- As the number of processes grows, response time can also begin to take a hit

Priority Scheduling

- **Priority** based scheduling allows some tasks to be considered more “important” than others
- In this scheme, a number is assigned to the process to designate its priority
- The scheduler sorts tasks by their priorities and executes them in that order
- Unix systems: process **nice** level

Lottery Scheduling

- With **lottery** scheduling, each process is allocated lottery **tickets**
- The scheduler picks a number at random, and the winning ticket gets the CPU
- The more tickets a process has, the higher the chance it will be scheduled
  - This lets us allocate tickets to assign CPU priorities
  - Want to give a process 30% of the CPU? Give it 30% of the overall number of tickets

Lottery Scheduling Weaknesses

- If processes frequently stop/start, assigning tickets becomes difficult!
- Lottery scheduling works well on systems with long-running processes
  - Great for server environments:
    - Web server gets 70% of the CPU
    - Database server gets 20%
    - 10% goes to other background processes
- Workloads with highly variable thread counts suffer

Basic Scheduling: Wrapping Up

- By now you may be wondering “what’s the point of these algorithms?”
  - They all do have glaring weaknesses
  - In general, they are not suitable for consumer operating systems like Windows, macOS, Linux
- However, they can be used for problem-specific OS
- The tradeoffs here impact the more complicated schedulers as well

Real-Time Operating Systems

- RTOS operate in embedded applications and have different scheduling constraints
  - Processes expect to be scheduled at certain intervals and run for fixed amounts of time
- For example, your car might need to update the LCD speedometer display every 10 ms
  - This needs to be done constantly, and will probably take the same amount of time to execute
- Embedded systems, robots

Linux RT Scheduling

- Linux can be configured to provide real-time (ish) scheduling
  - SCHED_RR
  - SCHED_FIFO
- The time quantum can also be configured:
  - `cat /proc/sys/kernel/sched_rr_timeslice_ms10`

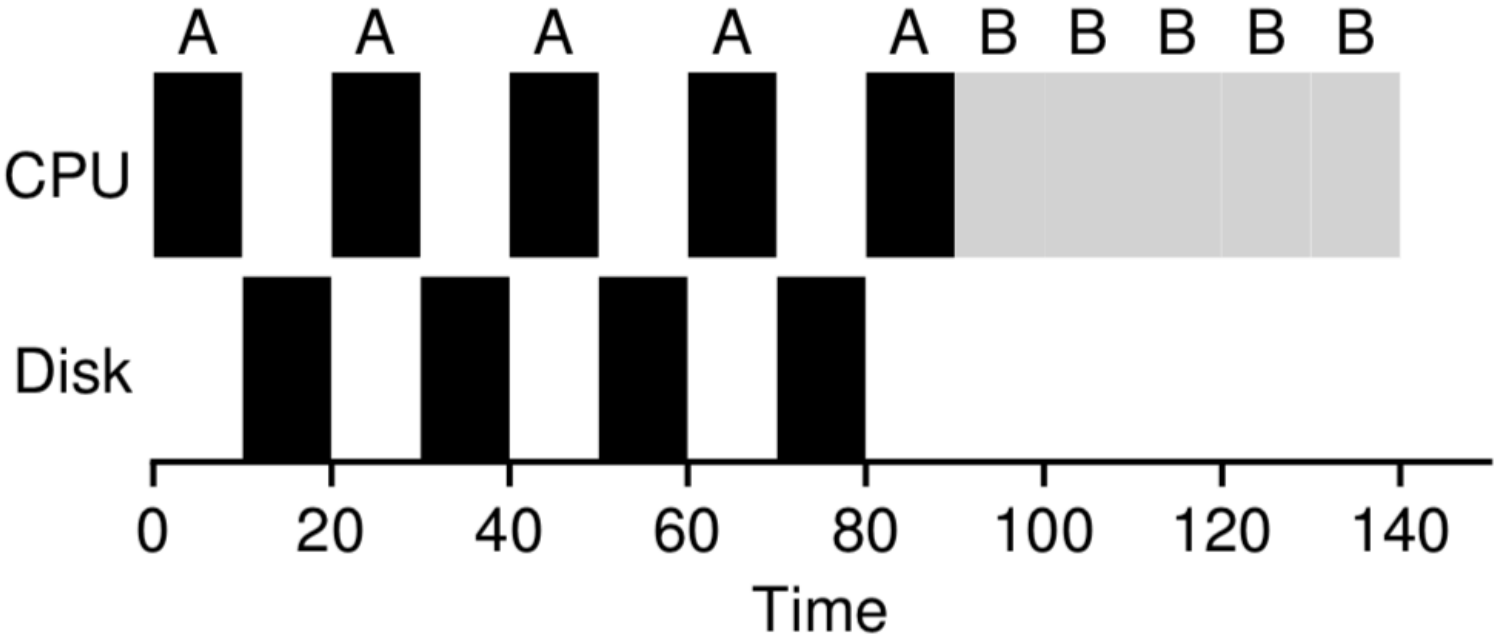
Today's ~~Schedule~~ Agenda?

- Context Switches and Interrupts
- Basic Scheduling Algorithms
- **Scheduling with I/O**
- Symmetric Multiprocessing

Input/Output

- Thus far, we've ignored I/O
  - Reading/writing to disk, network, etc.
- When a process performs I/O, it **blocks** to wait for the operation to complete
  - **Q**: Can a process still work on other things while waiting for I/O?
  - **A**: Yes… by using other threads
- If the scheduler does not account for this, CPU usage drops (CPU is idle during I/O)

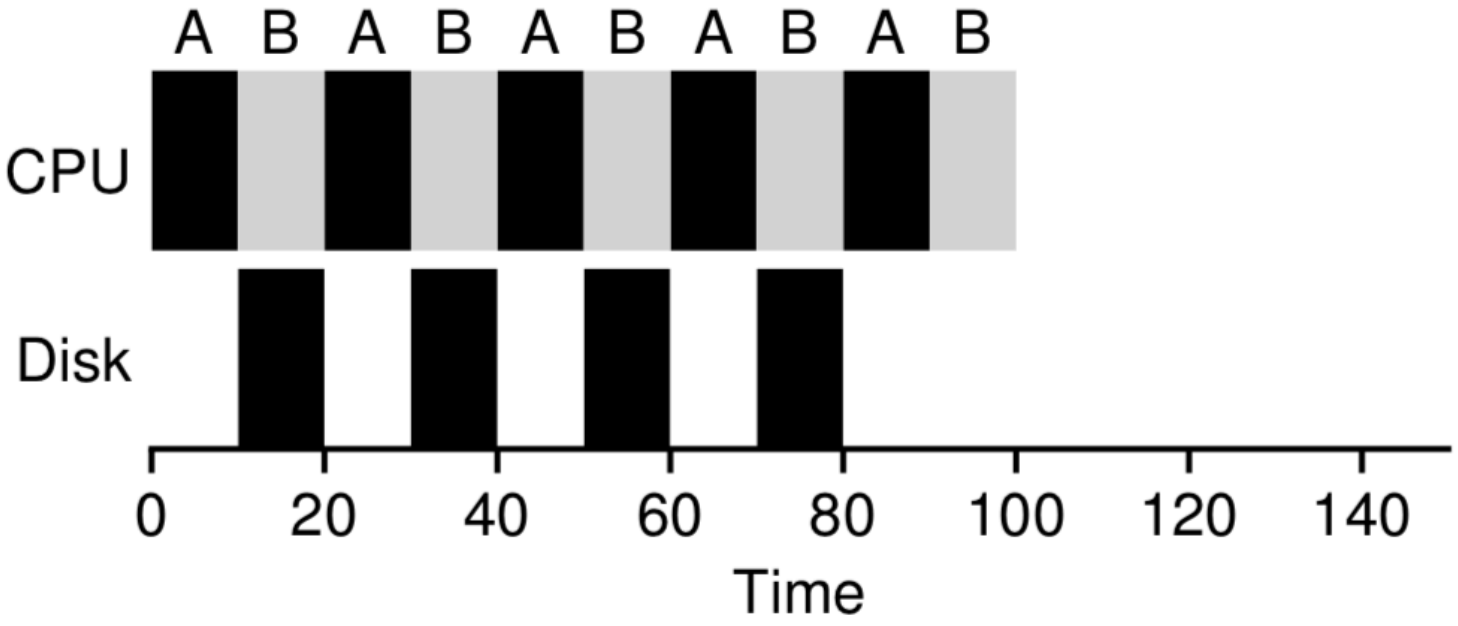
I/O: A Executes, then B



Preempt During I/O

- When a process begins an I/O operation, the scheduler will preempt it
- This allows another process to use the CPU during the I/O
  - Boosts overall CPU usage
- An interrupt informs the OS when the I/O operation completes, allowing the first task to run again
- **I/O Interleaving**

I/O Interleaving



Today's ~~Schedule~~ Agenda?

- Context Switches and Interrupts
- Basic Scheduling Algorithms
- Scheduling with I/O
- **Symmetric Multiprocessing**

Multiprocessing

- The days of having just a single CPU are over
- Even our phones have multiple cores!
- All of our discussion thus far assumes that there is only one CPU
- In symmetric multiprocessing (SMP), each processor has access to the same memory and devices
  - This is the common configuration

Queuing

- In an SMP configuration, each processor gets its own process **run queue**
  - Contains processes ready for execution
  - Sharing a single queue would require locking and coordination
- The OS assigns new processes to the queues based on processor **load**

Processor Affinity

- CPUs generally contain **cache memory** that operates much faster than system memory (RAM)
- As processes are switched in and out, the cache may already contain some program state
  - To take advantage of this, the OS tries to keep processes running on the same CPU/core
- **Processor Affinity**
  - Hard affinity – always scheduled to the same CPU
  - Soft affinity – best effort to schedule to the same CPU

Migration

- Processor affinity boosts performance by reducing context switch overhead
- However, processor affinity can lead to imbalances in load
- **Push migration**: OS finds too many processes in one run queue, moves them to balance it out
- **Pull migration**: OS finds an empty run queue, so it moves a process from a full queue

Kernel Ticks

- If the system gets interrupted every X ms, does this impact multiple processors/cores?
  - Yes!
- Most modern OS kernels are now **tickless**
  - Interrupts are set up and fired on demand rather than periodically
    - (e.g., process A has scheduling quantum of 18ms, so set the interrupt to fire in 18ms)
  - Also known as dynamic ticks
- Idle CPUs can stay idle, reducing power consumption

References

Scheduling figures from this lecture:

Operating Systems: Three Easy Pieces

http://pages.cs.wisc.edu/~remzi/OSTEP/