BIOS and Scheduling

David Marchant

2025-09-24

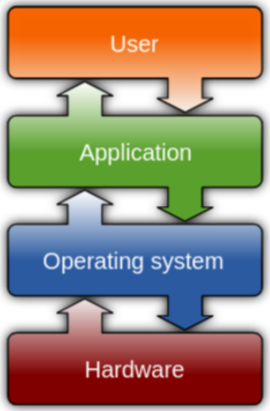
Summary

Hopefully from the material you've learnt:

- Why we need a scheduler
- What metrics we might use in scheduler decision making
- Why concepts like starvation are a problem
- What non-preemptive schedulers are
- What preemptive schedulers are

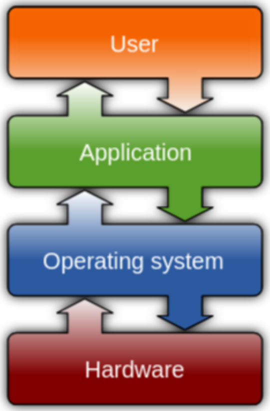
The Operating System

- The OS exists as a sort of middleware between applications and hardware
- It's not strictly necessary to run a computer, and some systems exist without an OS at all. And they run fast
- But the OS simplifies so much, and without it you really need to do everything from scratch, often in machine code (Boo)



An Aside - Layering

- We've seen this sort of thing already with language levels
- Layering is an important concept used repeatedly in Computer Science (and other places)
- It's a way of building on previous knowledge



Hardware and Software

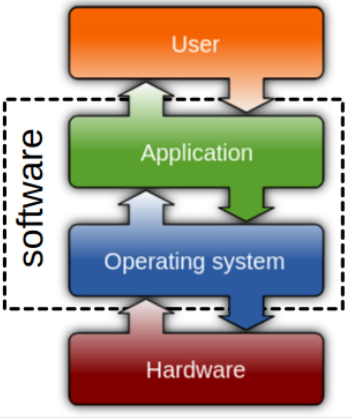
- Worth setting out what we mean by these words
    - Hardware is physical machinery that takes signals to manipulate physical components, albeit small ones such as disk storage
    - Software is a collection of signals used to control that physical hardware
- Hardware performs a specific job that cannot be altered
- Software needs hardware to run on, but can alter how it uses that hardware

A CPU is Hardware

- Worth setting out what we mean by these words
  - Hardware is physical machinery that takes signals to manipulate physical components, albeit small ones such as disk storage
  - Software is a collection of signals used to control that physical hardware
- Hardware performs a specific job that cannot be altered
  - CPU, Screen, Lamp
- Software needs hardware to run on, but can alter how that hardware is used
  - Spotify, Minecraft, Windows

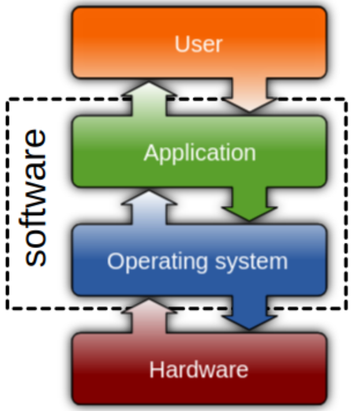
The OS Is All Just Software

- The OS is just software!
  - Note that some components are handled in dedicated hardware (cache, MMU, ALU, FPU, etc.)
  - But this is just because they are operations that have become standard enough that we can make dedicated hardware for them.
  - Originally many of these were software
- This makes it powerful, configurable, and slow(er than just machine code)



The OS Is All Just Software

- Consider everything we've covered so far
  - Caching, Processes, Floats, Memory Allocation, etc.
- These are all managed by software, even when they seem to depend on hardware
- And most of them are managed by the OS



Let's Consider Processes

- They're just a collection of (meaningful) data
  - Virtual Memory Space (stack, heap, code, etc.)
  - Execution Context (registers, etc.)
  - Plus some other things added by the OS to track each process (PID, parent, children etc.)
- There's nothing in hardware defining any of this, and the data that defines a process can just be saved to memory/disk just like any other file
- What makes it a meaningful process, is that the OS swaps out these saved collections of data on the fly

The Operating System

- It manages the complete runtime of the computer
- It's essentially *the* hard coded 'program' that starts everything else
- All those processes, memory accesses, etc etc...

But That's Not Enough

- The trouble with software is it only does anything when the computer is running
- E.g. powered up and already in a 'runnable' state
  - What does runnable even mean?

Where To Begin?

- Modern computers are large, complex beasts
- Operating Systems only complicate things by adding all their own layers
- We need some hardware to start things going
- Or at least start the ball rolling so the OS can set up the rest

BIOS

- Basic Input/Output Stream
- Performs a few functions that are outside our scope (Well, more outside our scope)
- The three important ones though are running basic setup, performing POST test, and handing off to the OS

The BIOS Chip

- After *seconds* of internet search, this is the best picture I could find, somehow
- The BIOS chip comes pre-installed
- Usually designed to work with a specific motherboard
- A motherboard being the main hardware component of most PCs, linking everything else together and hosting the CPU itself

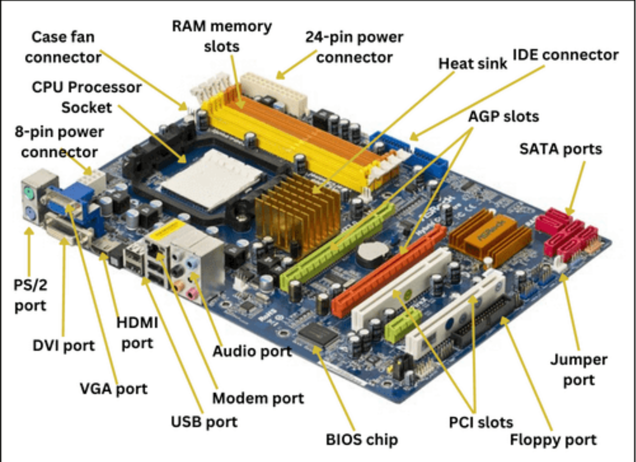
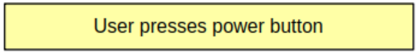
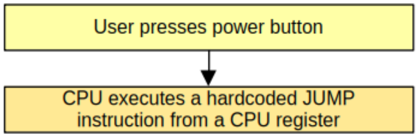


Fig: Main components of motherboard

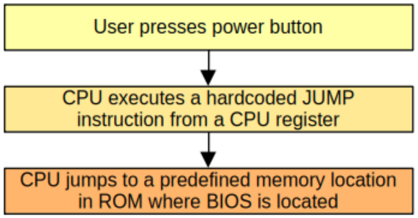
Boot Sequence



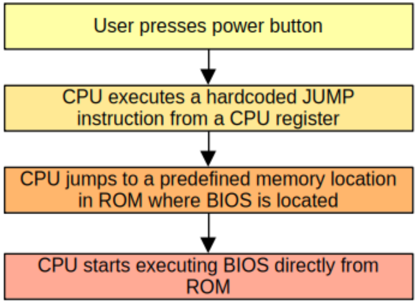
Boot Sequence


User presses power button
CPU executes a hardcoded JUMP instruction from a CPU register

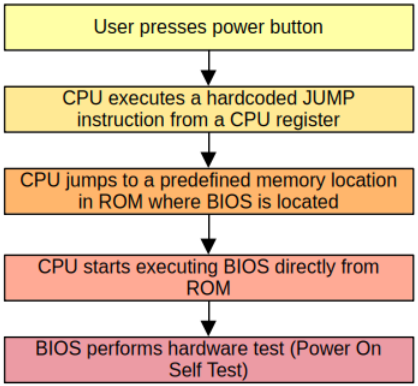
Boot Sequence


User presses power button
CPU executes a hardcoded JUMP instruction from a CPU register
CPU jumps to a predefined memory location in ROM where BIOS is located

Boot Sequence


User presses power button
CPU executes a hardcoded JUMP instruction from a CPU register
CPU jumps to a predefined memory location in ROM where BIOS is located
CPU starts executing BIOS directly from ROM

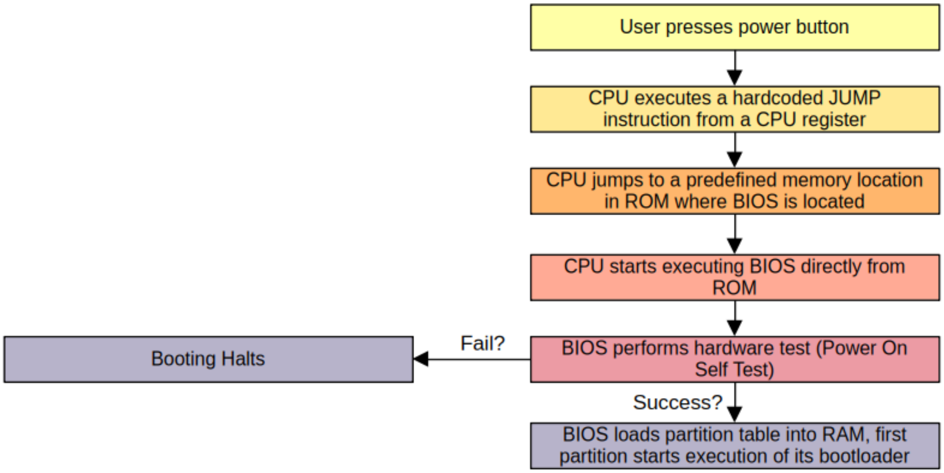
Boot Sequence


User presses power button
CPU executes a hardcoded JUMP instruction from a CPU register
CPU jumps to a predefined memory location in ROM where BIOS is located
CPU starts executing BIOS directly from ROM
BIOS performs hardware test (Power On Self Test)

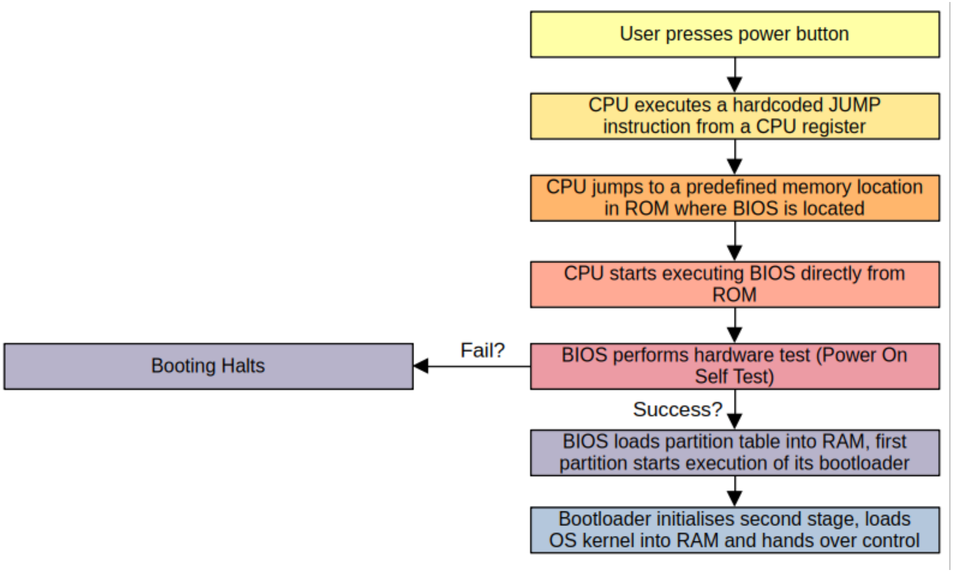
Power On Self Test

- Verifies integrity in BIOS code
- Verify basic hardware components (Registers, timers)
- Verify main memory (RAM)
- Initialise system buses
- Identify connected devices
- Plus other things buts its essentially verifying, identifying and sometime initialising

Boot Sequence


User presses power button
CPU executes a hardcoded JUMP instruction from a CPU register
CPU jumps to a predefined memory location in ROM where BIOS is located
CPU starts executing BIOS directly from ROM
BIOS performs hardware test (Power On Self Test)
Fail?
Booting Halts
Success?
BIOS loads partition table into RAM, first partition starts execution of its bootloader

Boot Sequence


User presses power button
CPU executes a hardcoded JUMP instruction from a CPU register
CPU jumps to a predefined memory location in ROM where BIOS is located
CPU starts executing BIOS directly from ROM
BIOS performs hardware test (Power On Self Test)
Fail?
Booting Halts
Success?
BIOS loads partition table into RAM, first partition starts execution of its bootloader
Bootloader initialises second stage, loads OS kernel into RAM and hands over control

Bootloader

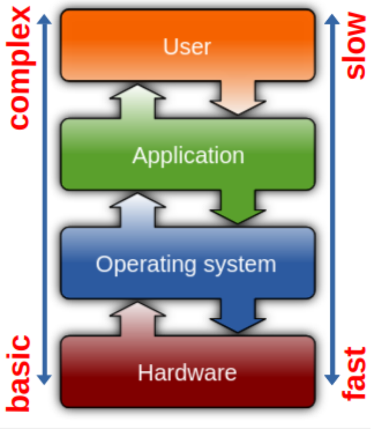
- Stored in non-volatile, read-only memory
- Initialises the different components if necessary, most notably the RAM
- Loads the OS kernel into RAM
- Then hands off control to that OS kernel which will handle everything from then on
- *This* is why a kernel isn't really a process, and the OS is just software

Why BIOS in Hardware?

- Most fast memory (SRAM and DRAM) is volatile
- We need to start somewhere
- We could just make all our memory out of non-volatile memory
    - But that's expensive

Why OS in software?

- Typically, the further up the layers we go, the less time efficient it is
- But also the further up we go, the more concept efficient it is
- Design is always about compromise, computer design is no different
- The OS can provide all sorts of powerful features allowing for concurrency, multi users, security, data management, pretty pictures



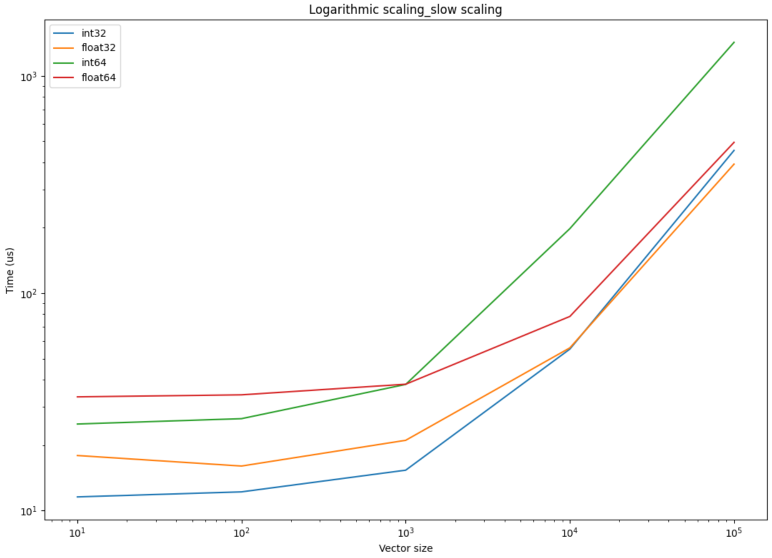
But Some Software Becomes Hardware

- MMU used to be just software lookup
- Cache always requires dedicated hardware, but more of it now and its better managed
- ALU/FPU used to be just regular CPU operations
- Implementing in hardware is quick!

Another Aside - Hardware vs Software

- Anything we can support directly in hardware will be *fast*
- Primarily this is down to pipelining of operations
- Can lead to some unintuitive results ...
- The following are some timing tests of large numbers of int and float operations

Another Aside - Hardware vs Software



Another Aside - Hardware vs Software

- For small problems ints are faster, as expected
- But as we get larger problems, floats actually become faster
- Each (probably) computed on dedicated hardware
  - ALU: Computes int and logic operations
  - FPU: Computes floating point operations
- We'll return to pipelining towards the end of the course...
- For now just see that implementing in hardware can make things *considerably* quicker

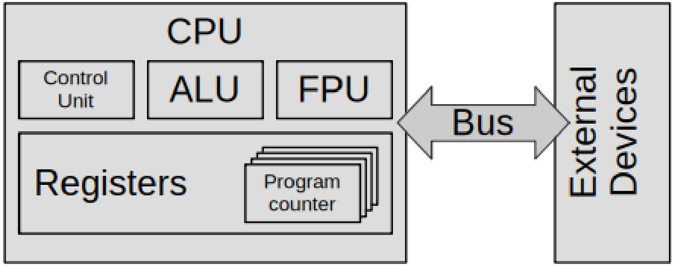
Could OS operations be implemented in hardware?

- No
- Hardware requires specifics to cemented to be worth the trouble
- We want to customise a lot of them OS operations (size of virtual memory, users, access to files)
- So we have to accept the overhead for these powerful features

The CPU

- A CPU is a dedicated piece of hardware for running software
- It has a few (ish) set of hard coded operations
  - Reading signals
  - Logical operations
- These are what machine code (and effectively Assembly) are expressing
- Software is just choosing what operation to perform in what order

The CPU



- It's a fundamental limit of a CPU that it can only ever do one thing at a time
- It's essentially just a fancy logic gate (this is a gross oversimplification)
- To run multiple processes, we need to context switch between them

The CPU

- If we don't do this well:

Starvation

One or more processes perpetually being denied the necessary resources to continue.

- This can manifest in a few ways
  - One or more processes will not progress at all
  - The CPU will compute unimportant tasks first
  - Hardware components will stop working
- Any of these are bad news, as presumably anything running has a reason to do so

Metric For Processes

- Once we've decided to decide between processes, we to decide on how we are going to decide between processes
  - How much memory they use
  - How long they take to complete
  - How much work they're actually doing
  - If it's a process that's important to the user
  - If it's a process that's important to the system
- The problem with most of these, is we only really know them once the process has already run.
  - Can estimate at runtime
  - But good estimation is often either accurate or computationally lightweight...
- This is a surprisingly complex topic, we're only going to broadly introduce here

Metric For Processes

- We need mechanisms to select processes that are simple, lightweight and meaningful
- Some of the following strategies will incorporate priority rankings, or time to completion
- Most will just use processes ID's to track that each process is getting some expected 'fair share' of computational resources.
  - Yes its crude, but it's also fast
  - And its usually good enough
  - In practice this whole lecture is somewhat of an oversimplification and many actual schedulers will use complex combinations of many of the following strategies and metrics

FIFO

- First In, First Out'
- Run each process, to completion, one after the other
- Perhaps the simplest scheduling strategy by far, no scheduling overhead except for the inevitable context switch once a process is complete and the maintenence of a process queue
- But if you've only a single core, you can only run a single process at a time
- What happens if task A is far too slow?

SJF

- Shortest Job First
- Variation of FIFO, sorted queue
- Still running each job to completion
- Some low overhead in maintaining a sorted list
- Gets *some* work done ASAP

Computation Time

- Consider three jobs:
  - A: Takes 10 minutes
  - B: Takes 1 minute
  - C: Takes 1 minute
- Now consider how each scheduler computes these:
  - FIFO: A->B->C, Mean completion time: $(10+11+12)/3 = 11$ minutes
  - SJF: B->C->A, Mean completion time: $(1+2+12)/3 = 5$ minutes
- We complete more work in a shorter time
- If all processes are queued at the same time, this is actually the optimal solution

STCF

- Shortest Time to Completion First
- Variation of FIFO, sorted queue
- Some low overhead in maintaining a sorted list
- Essentially the same as SJF, but we sort our process queue each time a job is added
- If a shorter job is added to the queue than the current one has left, context switch

Non-Preemptive Schedulers

- FIFO, SJF and STCF are all Non-Preemptive Schedulers
- They all run until a process has completed or it somehow gives up the processor
- This make it unfeasible to have a modern PC with multiple processes and programs running seemingly at once
- What happens if a process runs forever?
- But they are simple systems, and are still commonly used in many higher level scheduling
  - batch processing
  - thread pools
  - cloud services

Preemptive Schedulers

- We need to context switches because processes:
  - Can go on for a long time, even forever
  - Can need input from external devices or other processes
  - Can be scheduled at any time
  - Do not always know how many resources they will need ahead of time
- Note that even some non-preemptive schedulers still use context switches, but far more infrequently than what we're about to introduce

Round Robin

- The simplest form of preemptive scheduler
- Give each process a certain small period of computation
- Once one process has used up its small period, context switch to the next
- Don't reschedule a process until all others have had a period of computation
- Ensures some progress on everything ASAP
- Relatively low scheduling overhead, but high overhead for a lot of context switches
    - On most systems, context switch is 1μs or 1000 clock cycles
    - But perform enough and they add up
- This cost is constant (ish) so if we just give each process enough time to before the switch then it vanishes into the background.
    - Couldn't find any consensus on how long a typical scheduler gives each iteration but 100-250μs was mentioned reasonably often

Lottery Share

- Sort of similar to Round Robin
- But attempts to give all processes a proportional share of resources
- Assigns each process a number of 'tickets' proportional to the amount of work to be done
- Periodically, randomly pick a ticket, the owning process gets scheduled
  - If same processes drawn twice in a row, no context switching needed
  - Potential for starvation still exists, but there are ways around that
- Leads to probabilistic fairness in a reasonably sized system

The Benefits of Tickets

- Tickets have a lot of useful usability qualities for system administration
- Give users a set amount of tickets each, they can distribute them amongst their processes as needed
- If a process needs to wait for another process, it can lend it some of its own tickets to ensure it can progress
- Can also trivially manipulate ticket numbers for our own ends.
  - Privileging users by inflating their tickets
  - Increasing tickets of longer queued jobs should counter starvation

Stride

- In essence this is a non-random lottery
- Still give process tickets
- Scheduler calculates each processes `stride` inversely proportional to its ticket amount
- Each process start with a `pass` value of 0
- Update the `pass` value by the `stride` each time it runs
- Deterministically pick the process with the lowest `pass` to run
- Randomly select if two or more have the same `pass` value

Stride

- Lets imagine 3 Processes:
  - A = 100 tickets, 100 stride
  - B = 50 tickets, 200 stride
  - C = 250 tickets, 40 stride
- Here we've chosen 10000 as a value to divide the tickets by, but this is arbitrary. It doesn't affect the relationship

Pass(A)	Pass(B)	Pass(C)	Runs?	
0	0	0	A	<-Could also be B or C here
100	0	0	B	
100	200	0	C	
100	200	40	C	
100	200	80	C	
100	200	120	A	
200	200	120	C	
200	200	160	C	
200	200	200	A	<-Could also be B or C here

MLFQ

- Multi Level Feedback queue
- Assign each process a priority value
- Each priority values has an associated queue
- Higher priorities are computed before lower
- Within each queue use another scheduling system, usually some version of round robin.

Setting Priorities

- Sometimes done manually.
  - OS Processes prioritized
  - Hardware drivers
  - Network interactions
- Can also be automatic
  - Starting processes have the highest priority
    - Let everything start ASAP
    - Give opportunity to observe and learn
  - If a job willingly gives up CPU, keep priority the same
    - Probably hardware waiting for interaction
    - Keeps responses quick
  - If a job exceeds allocated computation time, decrement priority
    - Long running analysis that isn't waiting for anyone
    - Halting problem!

MLFQ

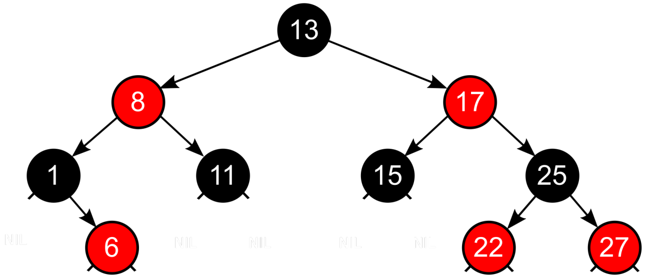
- Starvation is still an issue for lower priorities
  - Generally, long waiting processes have priorities increased
- Also, hardware processes remain high priority yet keep giving up the CPU meaning lots of spurious context switches
  - Allocate each process a 'computation time' across schedulings. Once exceeded, decrement priority
  - Computationally heavy processes will decrement quicker
- Do not need to give each priority queue the same computation allocation
  - After a time, higher priorities should be just the quick, hardware handling processes so give them each a small time allocation
  - Equally the lower you go, the longer running a process probably is, so it can be given a larger time allocation
  - As always, there's a balance to be struck here

CFS

- Completely Fair Schedular
- Used within Linux
- Effectively a weighted round robin system, plus additions
- Set a value, `sched_latency` to set how often (in theory) each process will be computed
- As we add more processes, they are divided equally across the `sched_latency`
- Set an additional value, `min_granularity` to ensure a minimum runtime per process
    - Ensures not too many context switches

CFS

- CFS intends to scale well. Modern systems frequently have thousands of processes
- Intuitively we can just keep a list/array
  - Picking off the next process is easy (if list is sorted)
  - But maintaining that sorting has O(n) complexity
- Better to use a tree, with O(log n) complexity for search, insert and delete



Scheduling

- Obviously this has been a broad overview
- Efficient scheduling is still very much an ongoing field
- By and large we treat it as a black box
- Modern OS's use preemptive scheduling to context switch regularly
- But many remote systems use cruder non-preemptive schedulers