CSE 451: Operating Systems

Spring 2022

Module 1

Course Introduction

John Zahorjan

zahorjan@cs.washington.edu

Today's agenda

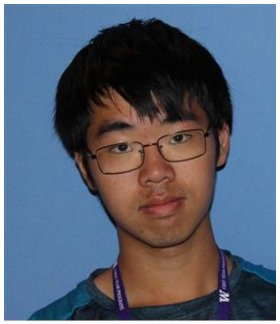
- Administrative Details
  - Course overview
    - course staff
    - general structure
    - "the text(s)"
    - policies
    - homework 0
- OS overview
  - An overview
  - A tour of concepts that keep coming up when talking about operating systems.

Course Staff

- Instructor: John Zahorjan

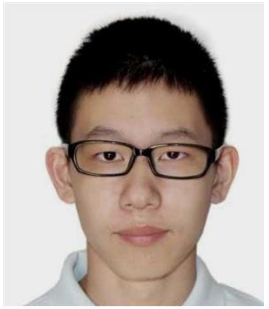
- TAs

Xiao Geng

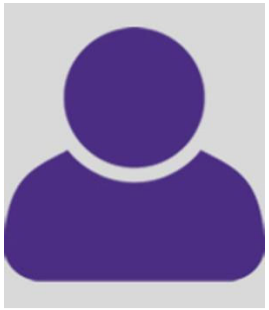


Byron Jin

Wayne Lai



Tom Lou



Ajay Rawat

Course overview

- Operationally, everything you need to know will be on the course web pages:

  http://www.cs.washington.edu/451/ and https://canvas.uw.edu/

- The course email (sometimes for announcements):

  cse451a_sp22@uw.edu

- The course discussion board:

  https://edstem.org/us/courses/21346/discussion/

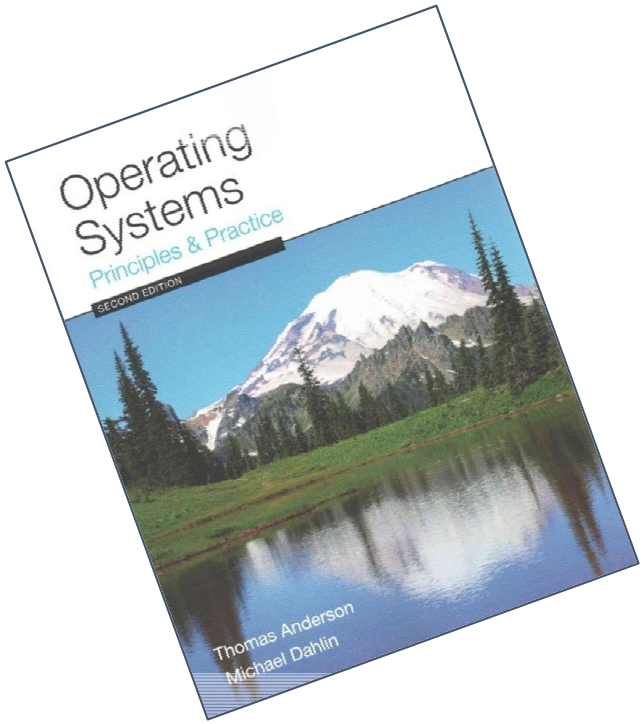
First Course Concept: Policy vs. Mechanism

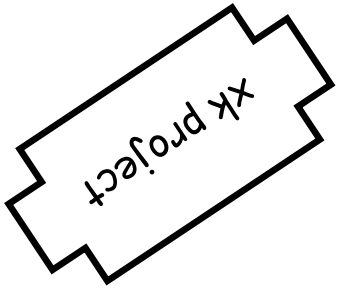
- Policy is what you're trying to achieve
  - All VMs running on a single AWS node should get about the same amount of CPU time per second
    - Warning: that's a crude example, more of a goal than a policy. Actual policies later in the course.
- Mechanism is how you achieve that
  - The OS monitors CPU time consumed and switches tasks when it gets too imbalanced
- Roughly, class/reading is about policy, the projects are about mechanism

Course Issue

Policy

Mechanism





Instructor

TAs

Class Resources

- The text
  - You might be surprised that there is a text...
  - Think of it as helping you to understand, and dig deeper than, the lecture, section, and project material
- Other resources
  - Many online; some of them are essential
- Policies
  - Collaboration vs. cheating
  - Projects: late policy
  - Team Malfunctions
    - I have no magic solution

Exams

- There is likely to be a final exam of some sort
- There is likely to be a mid-term exam
  - If there's going to be a final, you want there to be a midterm
  - (tentatively) May 2nd
- Note: final grades attempt to reflect course mastery at the end of the course
  - If your mid-term grade is low and your final grade high...

The Project(s)

- Projects
  - Start them early
  - Four of them
  - Teams of two. You're likely to be happier if you form a team on your own than if we form one for you!
    - Experience indicates that having a partner who can work during the same hours that you can is a big advantage
  - Experience says that working remotely presents some challenges
    - Four projects instead of five
  - Do not believe that passing the test cases means your code works...
    - Sorry
  - Grading mechanism
    - TBD

Late Policy

- There is one
- It's some balancing act of these principles
  - You know better than we do what all your responsibilities are
  - We owe it to you (and your partner!) to provide sufficient incentive to stay on schedule that you don't get yourself in trouble
- Don't get yourself in trouble!

- Roughly: It's late when the fact that it isn't yet done is causing someone unusual difficulties

Other Operational Issues

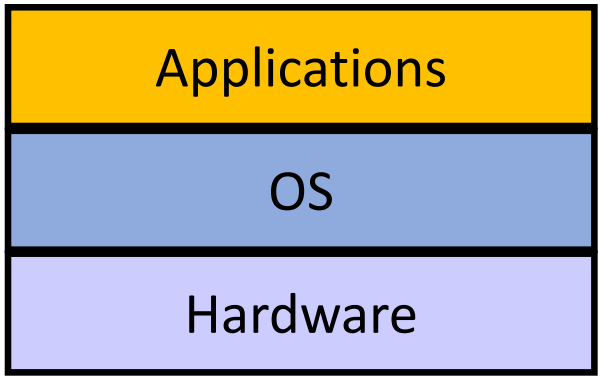
- I have office hours immediately after class Monday (in-person) and on Thursdays (zoom)
  - I'm of only marginal use about the project code
- There will be TA office hours as well

Overview: What is an Operating System?

- Answers:
  - I don't know.
  - Nobody knows.
  - The textbook claims to know – see Chapter 1.
  - They’re programs – big programs
    - The Linux source has over 27M lines of C
    - Windows has way, way more
  - They’re programs that make writing other programs easier
    - The raw hardware is an unforgiving place to execute...

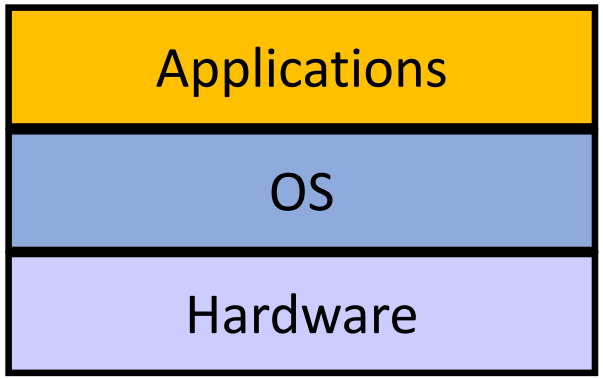
The Traditional Graphic

- The OS sits between your code and the hardware.
- It protects the hardware from your code.
- It protects your code from the hardware (by turning the ugly hardware interfaces into more easily used abstract interfaces that it implements)



- This depiction invites you to think of the OS as a library
- (This depiction invites you to think of the hardware as a library...)

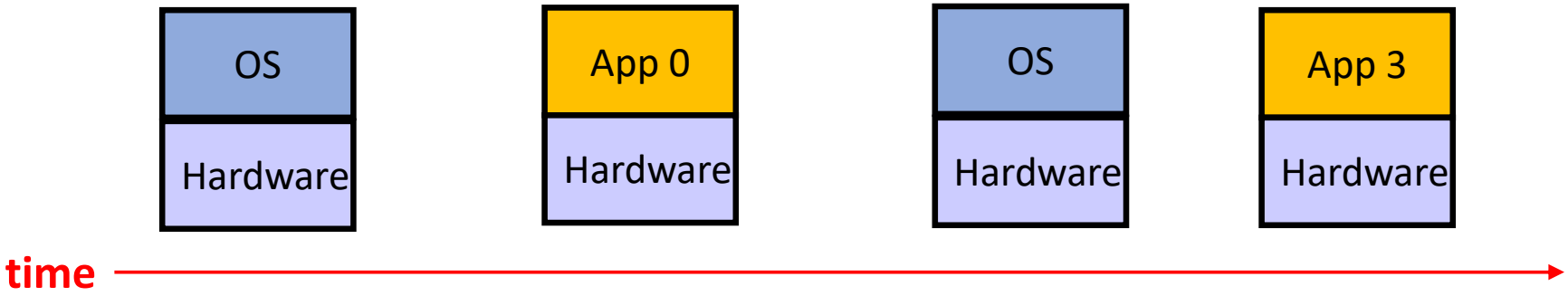
What's Right With That Picture



- The OS *is* between you (the app) and the hardware
  - Your application **cannot** directly manipulate (a lot of) the hardware, it has to ask the OS to do it on its behalf
  - This is the basis of security

- The OS *is (partly)* a library
  - Sometimes apps explicitly invoke it to request some function
  - Some functionality is in the kernel because it turns out to be fastest to put it there, not because it absolutely has to be there (no critical hardware is touched)

What's Wrong With That Last Picture?

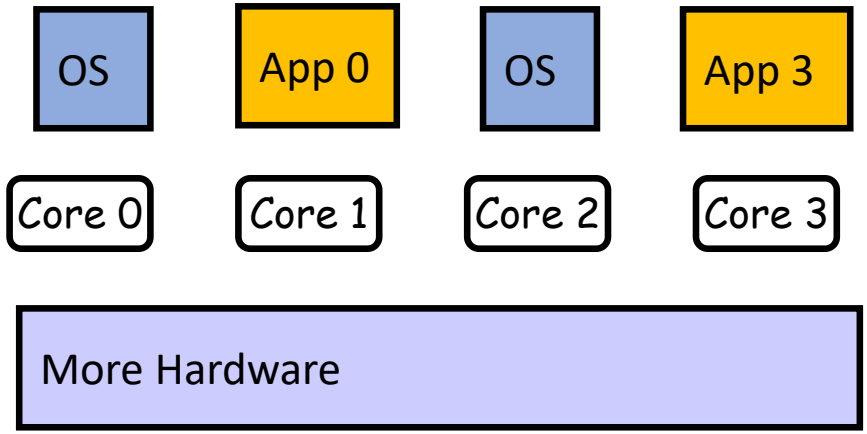


- The OS *isn't* always between you and (some of) the hardware
  - When you're running, your code uses some of the hardware directly
    - Which hardware?
    - Why (not interpose the OS between you and that hardware)?
    - Why not have separate hardware for the OS and the apps?
- The OS *isn't* a library
  - You don't tell it when to run, it tells you when to run
    - *(Okay, sometimes you tell it when to run...)*
  - You don't tell it where (what code) to run, it makes up its own mind

What's Still Wrong With That Last Picture?

It's from 2004

- why take turns when you can all go at once?



Execution from the Application Software Point of View

- Memory
  - I have values in local and global variables and you can't touch them.
    - You don't even exist.
- CPU
  - I execute this statement, then I execute the next statement, sometimes I loop, sometimes I call, but it's all me and my code (or library code called by me)
    - My code determines where execution goes
- Isolation in a Shared System
  - Okay, I know there are lots of program running but I don't care because they have no effect on me
    - except maybe for performance impacts
    - or because I let them (e.g., shared files)
  - *Isolation is one of the primary (and oldest) goals/abstractions of the OS*
    - *Simplifies writing applications*
    - *Promotoes efficient use of the hardware by allowing multiple applications to run at once*

Execution from the Hardware POV

- Tick, tick, tick, ...
- Fetch an instruction, execute an instruction
  - Each instruction may modify some machine *state*
    - Update some register or memory value(s)
  - Each instruction depends only on the current hardware state
    - It doesn't matter how the hardware got into that state
- From the hardware's point of view, there are no programs
  - No variables, just addresses
  - No statements, just instructions
- The hardware is a state machine
  - there is no OS or application(s)
  - there is just the current state of the machine and the next state
    - the state of the machine is more complicated than what you can see from the application level
    - for example, the state includes the set of page tables currently in use
- *However, the hardware is designed to enable us to write OS's and applications and do other useful things*

Execution from the OS POV

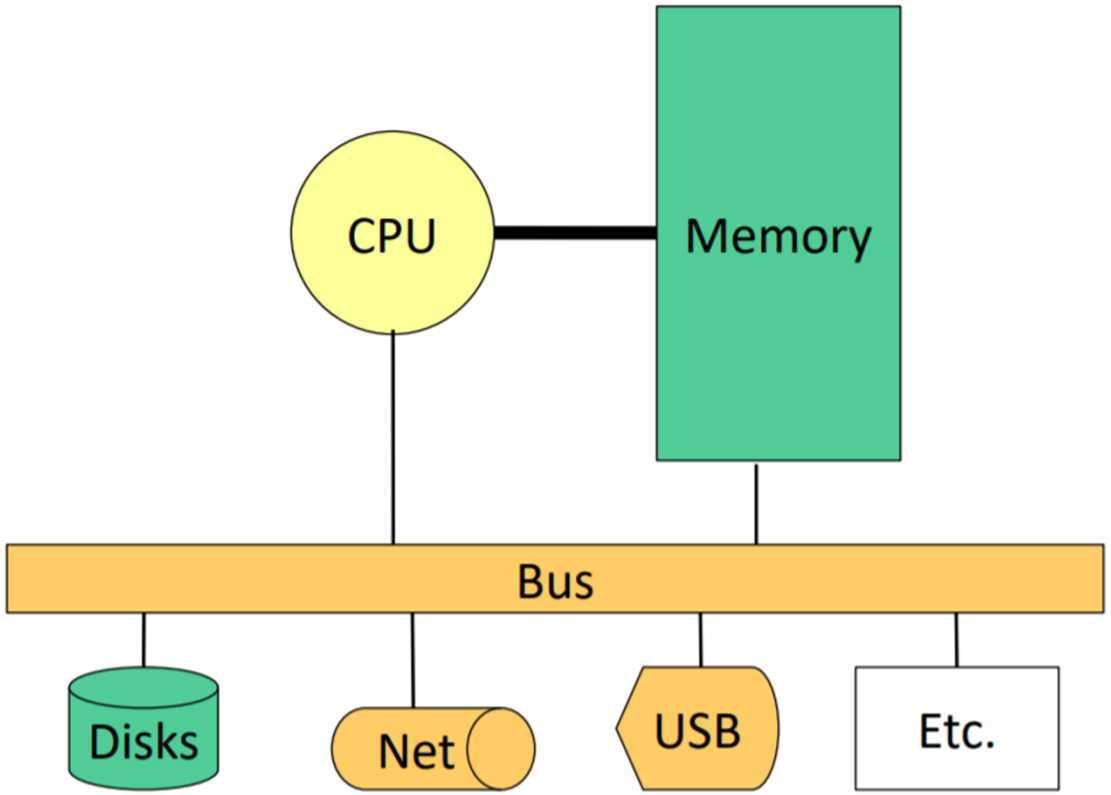
- On the one hand, the OS is a benefactor of the abstractions it creates
  - A lot of its execution feels just like application code
- On the other hand, the OS interfaces with/controls/responds to the hardware
  - That code can feel more foreign
- Some features of the hardware are designed to make it possible to write a (reasonable/efficient) OS
  - The OS insulates application programs from ever having to deal directly with many aspects of the hardware
- The OS has to deal with hardware asynchrony
  - The OS provides apps an abstraction of orderly execution, where nothing unexpected happens
  - The hardware doesn't
  - The next instruction to be executed sometimes has nothing to do with the one currently being executed, or its location in memory

Execution from the OS POV

- Things are more complicated…
- Suppose the system is running both processes A and B
  - each has its own (virtual) address space
  - each has its own stack
  - when the OS reallocates a core from A to B it must switch the address space the core operates in
    - which address space is it using while it's switching between the two?
    - which stack is it using while it's switching?
    - what about values in the hardware cache? do they remain valid?
- The OS must sometimes deal with physical memory
  - so, an address could mean a value in A's address space, or B's, or physical memory's
- Control flow is odd
  - "Events" happen asynchronously (that is, "whenever")
  - each event may cause the core to abandon the instruction sequence it was executing and switch to an event handler

The Abstract View of Hardware

Hardware: Logical View



The xk View of Hardware (Partial)

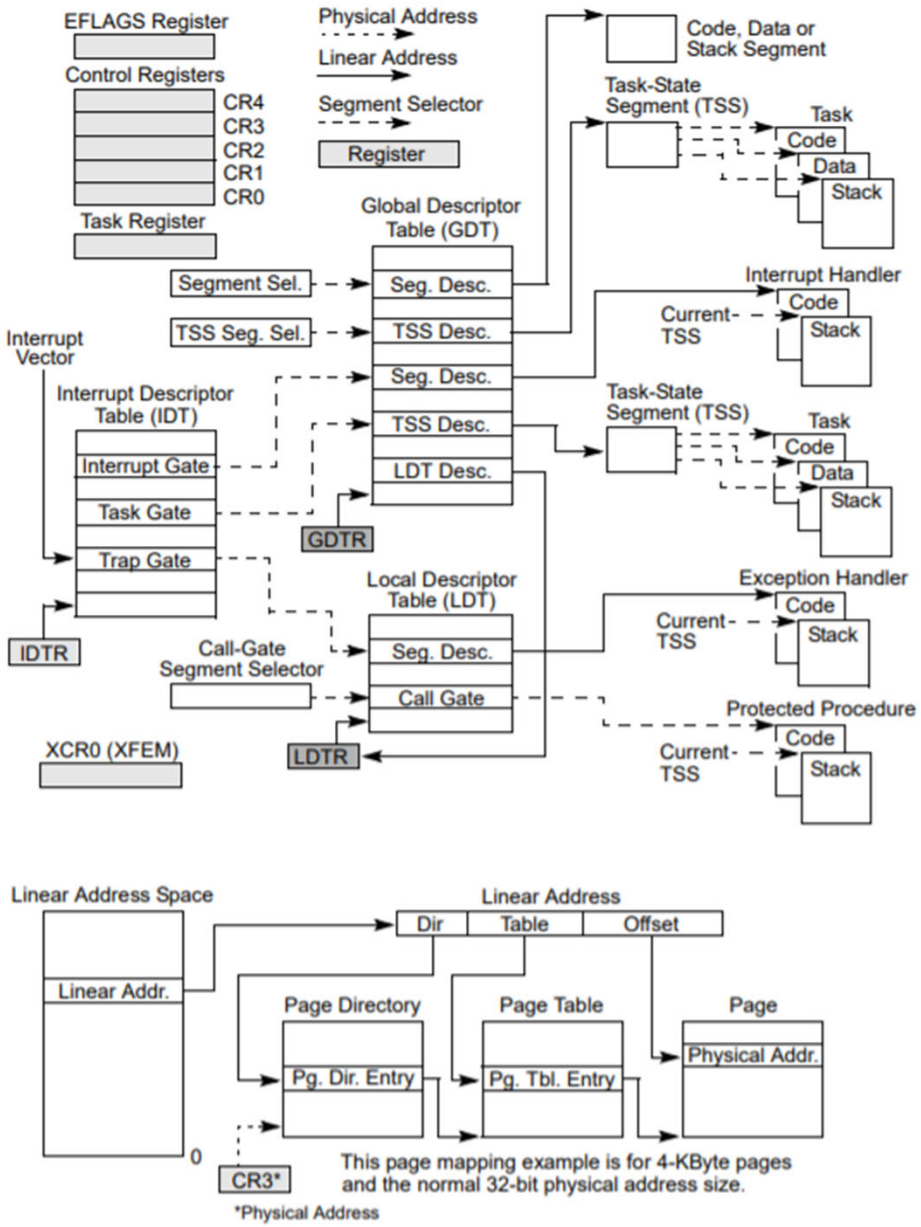


Figure 2-1. IA-32 System-Level Registers and Data Structures

The OS and Hardware

- An OS mediates programs' access to hardware resources (*sharing* and *protection*)
  - computation (CPU)
  - volatile storage (memory) and persistent storage (disk, etc.)
  - network communications (TCP/IP stacks, Ethernet cards, etc.)
  - input/output devices (keyboard, display, sound card, etc.)
- The OS abstracts hardware into logical resources and well-defined interfaces to those resources (*ease of use*)
  - processes/threads (CPU, memory, instruction execution)
  - files (disk)
  - sockets (network)
  - streams (keyboard, display, sound card, pipes, …)

Why bother with an OS?

- Application benefits
  - programming **simplicity**
    - see high-level abstractions (files) instead of low-level hardware
    - abstractions are reusable across many programs
  - **portability** (across machine configurations or architectures)
    - device independence: 3com card or Intel card?
- User benefits
  - **safety**
    - isolation: program "sees" its own virtual machine, thinks it "owns" the computer
    - OS protects programs from each other
    - OS multiplexes resources across programs
  - **efficiency** (cost and speed)
    - share one computer across many users
    - concurrent execution of multiple programs

The Major OS Issues

- **structure**: how is the OS organized?
- **sharing**: how are resources shared across users?
- **naming**: how are resources named and what is the scope?
- **protection**: how is one user/process protected from another?
- **security**: how is the integrity of the OS and its resources ensured?
- **performance**: how do we avoid making it all slow?
- **availability**: can you always access the services you need?
- **reliability**: what happens if something goes wrong (either with hardware or with a program)?
- **extensibility**: can we add new features?
- **communication**: how do programs exchange information, including across a network?

More OS Issues…

- **concurrency**: how are simultaneous activities (computation and I/O) created and controlled?
- **scale**: what happens as demands for resources increase?
- **persistence**: how do you make data last longer than program executions?
- **distribution**: how do we allow a computation to span hardware (machine/network) boundaries?
- **accounting**: how do we keep track of resource usage, and perhaps charge for it?
- **auditing**: can we reconstruct who did what to whom?

There are tradeoffs – not right and wrong!

(Ok, some things are clearly wrong, but there is no right.)

Possibly Useful Concepts

1. Efficiency
2. Dealing with latency
3. Policy vs. mechanism
4. Who sets policy?
5. Interposition (virtualization)
6. Naming
7. Synchronization

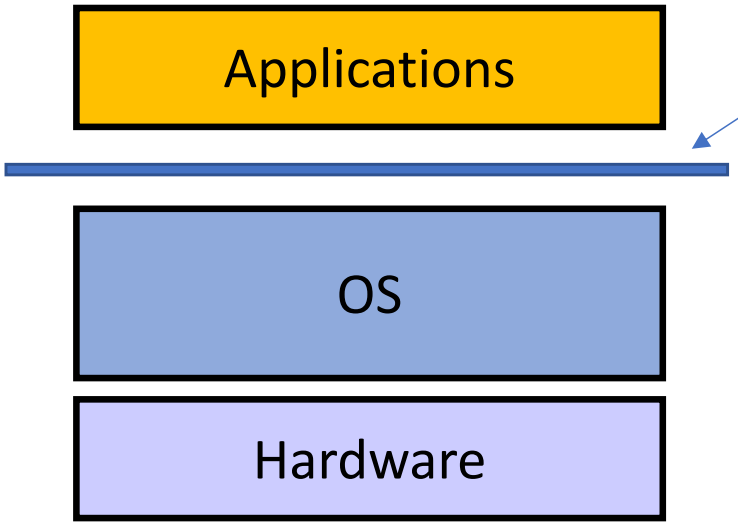
1. Efficiency

- What might it mean for an OS to be *efficient*?
  - (What might it mean for a compiler to generate efficient code?)
- When you read about operating systems, you'll often see that their efficiency is evaluated by how long it takes them to do some operation(s) on some benchmark hardware
  - Different OS's may have different operations...
- The point of the OS is to enable efficient implementation of applications that actually do something useful
- "efficient implementation"
  - There's coding time, and
  - There's run time

1. Efficiency

- One way to think about runtime efficiency
  - If I were to take the time to write all the required for my application to boot and run on raw hardware, how much faster could it be?
  - This is asking what the penalty/cost is to get what the OS provides:
    - Sharing of the hardware among apps
    - “Limited damage” when programs have bugs
- What about code time efficiency?
  - The OS has a big influence on code time efficiency as well, because of the abstractions/interfaces it implements
    - Think of how easy/hard it is to write apps in the various programming languages you’ve used...

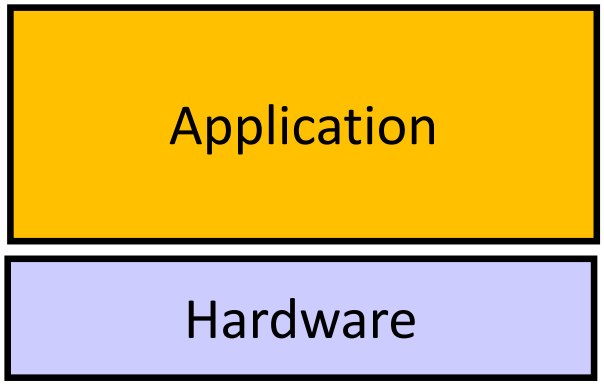
1. Efficiency



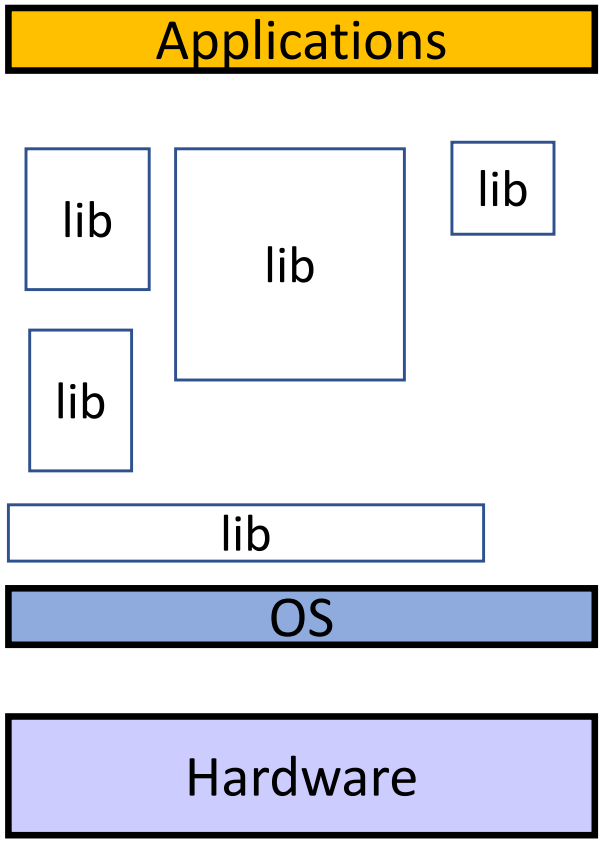
OS API

What abstractions are provided?
How compact do they allow apps to be?
How much code and code complexity do they force on the OS implementation?

VS.



1. Efficiency



General lesson from history:
simpler is faster

Layer functionality, don't build it all into any single layer

Generality has a cost, and results in overhead for all apps, even those that don't need the generality

Brief Aside: What Does It Cost?

- Not asymptotic complexity, picoseconds...
- Let's look at costs in a somewhat CPU independent way
- Let the unit of time (i.e., 1 time unit) be the time to execute each iteration of this loop, averaged over many iterations

```
for (i = 0; i< LOOPCNT; i++) ;
```

- What does it cost to do other things?
  - Call a procedure, passing no parameters
  - Call a procedure, passing many parameters
  - Invoke the OS
  - ...
- Why does this matter?
  - OS design must respect costs...

Step 1: Poll (HW0)

- Part 1: Basic Operation Costs
  - https://docs.google.com/forms/d/1PSeh7aH_1WrLWXoAWo4Uj0Oxz_91a6MqAGGsPynxgn8/edit
- Part 2: Multi-threaded Execution Costs
  - https://docs.google.com/forms/d/1gRkr9bW41XUHLvTyzFp_367mtgNvrWYkiqMX4k50EKI/edit

Possibly Useful Concepts

1. Efficiency
2. Dealing with latency
3. Policy vs. mechanism
4. Who sets policy?
5. Interposition (virtualization)
6. Naming
7. Synchronization

2. Dealing with Latency

- “Latency” is a delay before progress resumes
- “High latency” is a delay that is long relative to the unhindered processing rate
- Some high latency operations:
  - cache miss
  - network communication
  - calling a procedure
  - interacting with an I/O device
  - a thread producing a result
  - invoking the OS
  - creating a process
  - creating a thread

2. Dealing with Latency

- One visualization of latency

- Progress (blue) is *synchronous* with the high latency operation (red)
- The “blue resource” has to wait for the “red resource”

- One possible mitigation technique:
  - **Caching** – don’t do that high latency operation if you can avoid it
    - Is it correct?
    - Does it help?
      - “Temporal locality”

2. Dealing with Latency

- Other possible mitigation techniques:
  - **Asynchrony/concurrency** – Find work you can do at the same time as the high latency operation

  - **Speculation** – An extreme form of asynchrony – start the high latency operation before you're completely sure that you'll need it and/or what the arguments to the operation will be

Examples: browser fetches a linked page; processor fetches instruction at predicted branch target address

2. Dealing with Latency

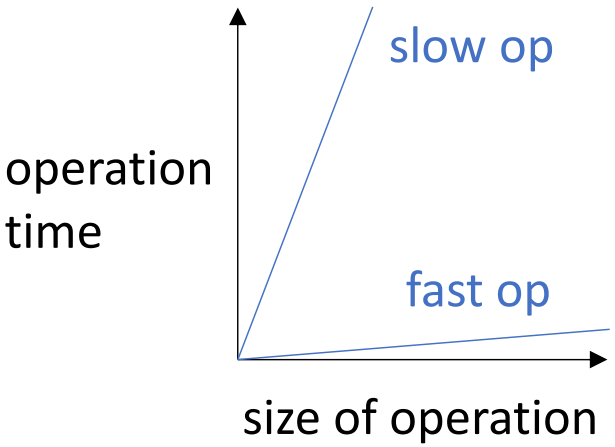
- Multiprogramming (another variant of asynchrony/concurrency):

- The blue processing is synchronous with the high latency operation
  - That's a pretty simple programming mode
    - No concurrency/asynchrony
    - It's the one we're used to
- Green (another process, say) isn't waiting for the event started by blue and doesn't interact/interfere with blue, so no reason green can't run…

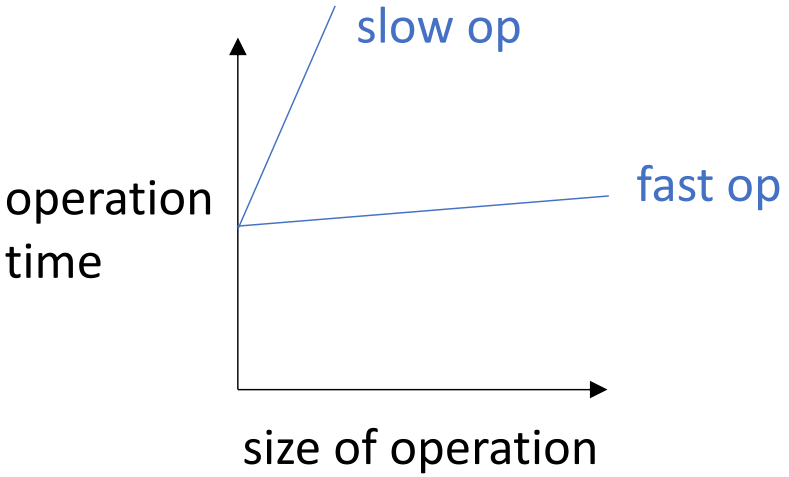
2. Dealing with Latency

- A second visualization

No/low latency operation



High latency operation

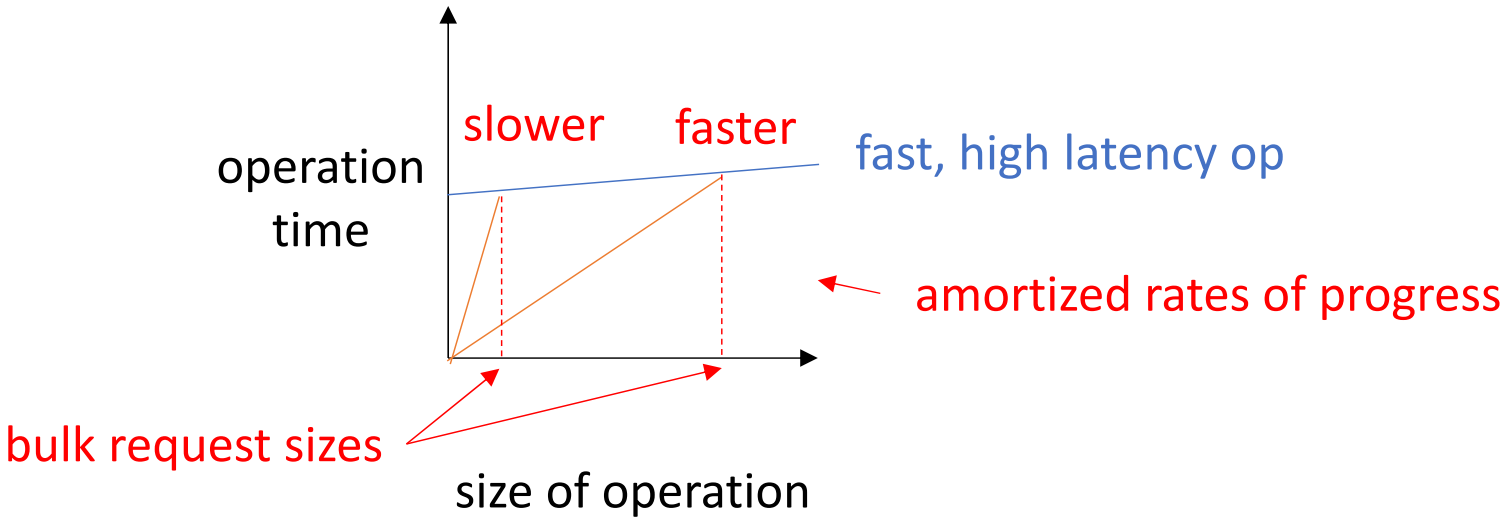


Examples: I/O operations; network operations

It takes 5 msec. to contact the remote server to ask for data, but data comes back at 1Gbps once it starts

2. Dealing with Latency

- Another mitigation approach: bulk requests (amortization)



- Bulk requests
  - A single request for a million bytes is much faster than a thousand requests for a thousand bytes
- The bigger the bulk size, the lower the cost per unit work
- There's a natural interplay with speculation
  - "I only need 500 units right now, but it won't cost much more to fetch 1000 units in this one operation and maybe the extra 500 will turn out to be useful, so why not?"

3. Policy vs. Mechanism

- Mechanism is the set of things that you can do
- Policy is how to use mechanism in specific situations
- Non-computer example:
  - A car is mechanism
    - Operations: go, stop, turn
  - The mechanism is general
    - It can help you get from the UW to Bellevue
    - It can help you get from Vancouver to Toronto
  - Contrast with a public bus…
- The mechanism (car) doesn't say anything about how to use the mechanism to get anywhere in particular
    - The driver provides policy by deciding on a route, say
- As self-driving cars are being developed, what possible downside could there be for decisions about routes being built into the cars?

3. Policy vs. Mechanism

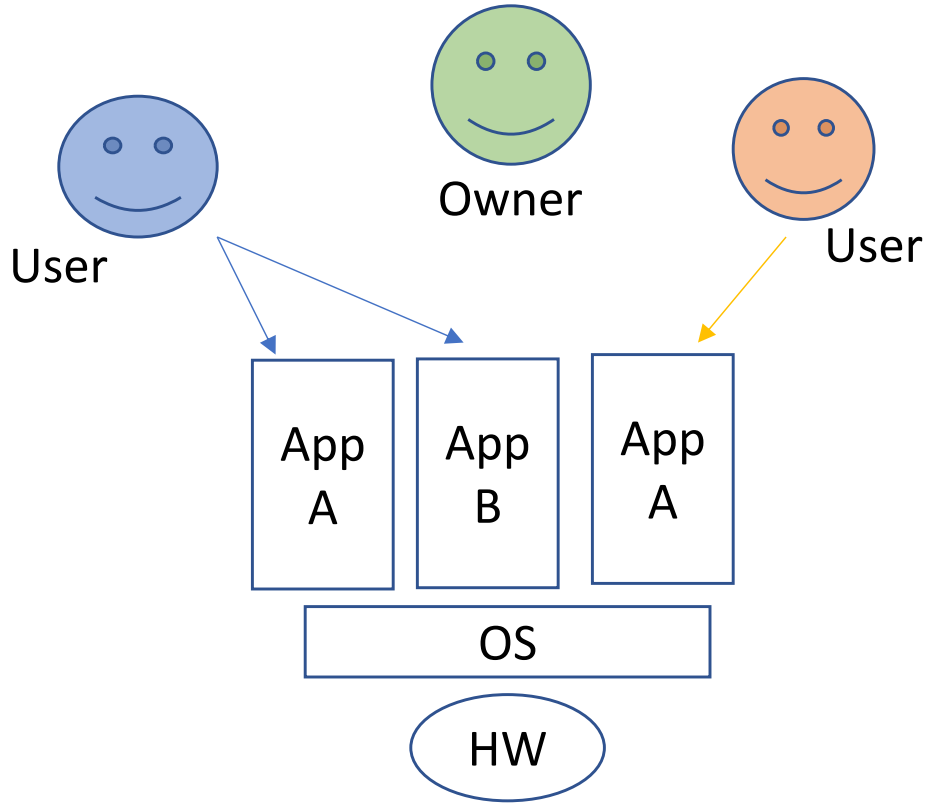
- Historically, operating systems provide both policy and mechanism
  - They provide a way to share resources (like CPU cycles) among users *and* provide a policy that dictates how much resource each user gets
  - On the other hand, there is a long-standing trend to enable movement of policy out of the operating system

3. Policy vs. Mechanism: Computer Example

- You pay Lenovo for a new laptop
  - The hardware is mechanism – a set of things it can do
  - For your convenience, it probably comes with an operating system, which we can think of as more mechanism (lets you run programs)
  - But the laptop doesn't have a defined use
  - It's incredibly useful exactly because the purchaser can decide what software to run on it, so what it ultimately does
  - In that sense, the creators of the hardware and the OS defer the decision of what the system will do to the purchaser
    - The purchaser makes that decision much after the system has been built
- Why all this arduous argument?
  - Note that this isn't the usual subroutine call situation
  - The purchaser makes the decisions, but not by sending a message to Microsoft saying what they want the OS to do, nor to Lenovo saying what they want the hardware to do
  - Instead, somehow Lenovo and Microsoft "invoke the purchaser"
  - This probably sounds confusing/dumb...
  - I mean to contrast it with the standard use of a library
    - Library (mechanism) is written first
    - Policy setter (the app) is written second
    - Policy setter invokes the mechanism using the known library interface

4. Who Sets Policy?

- Who are the “principals” that could be setting policy?



Possible Decision Makers

- Owner of the system
- Users
- Author of app code (A and B)
- Each running instance of the apps (via code implemented by the authors)
- (Each library author or instance)
- The OS
- The hardware

4. Who Sets Policy? Control Flow

Policy decisions can flow down

An app invokes the OS to tell it the priority it wants to assign to each of its threads, knowing that the OS will assign idle cores to threads in a way that respects the priorities

(You pass down arguments, but the only things that can happen are things that the lower level has already implemented)

Deferral of decisions can flow up

When the OS notices it has an idle core it "asks" the app which thread to run on it.

How can the OS invoke an app that wasn't even written when the OS was compiled?

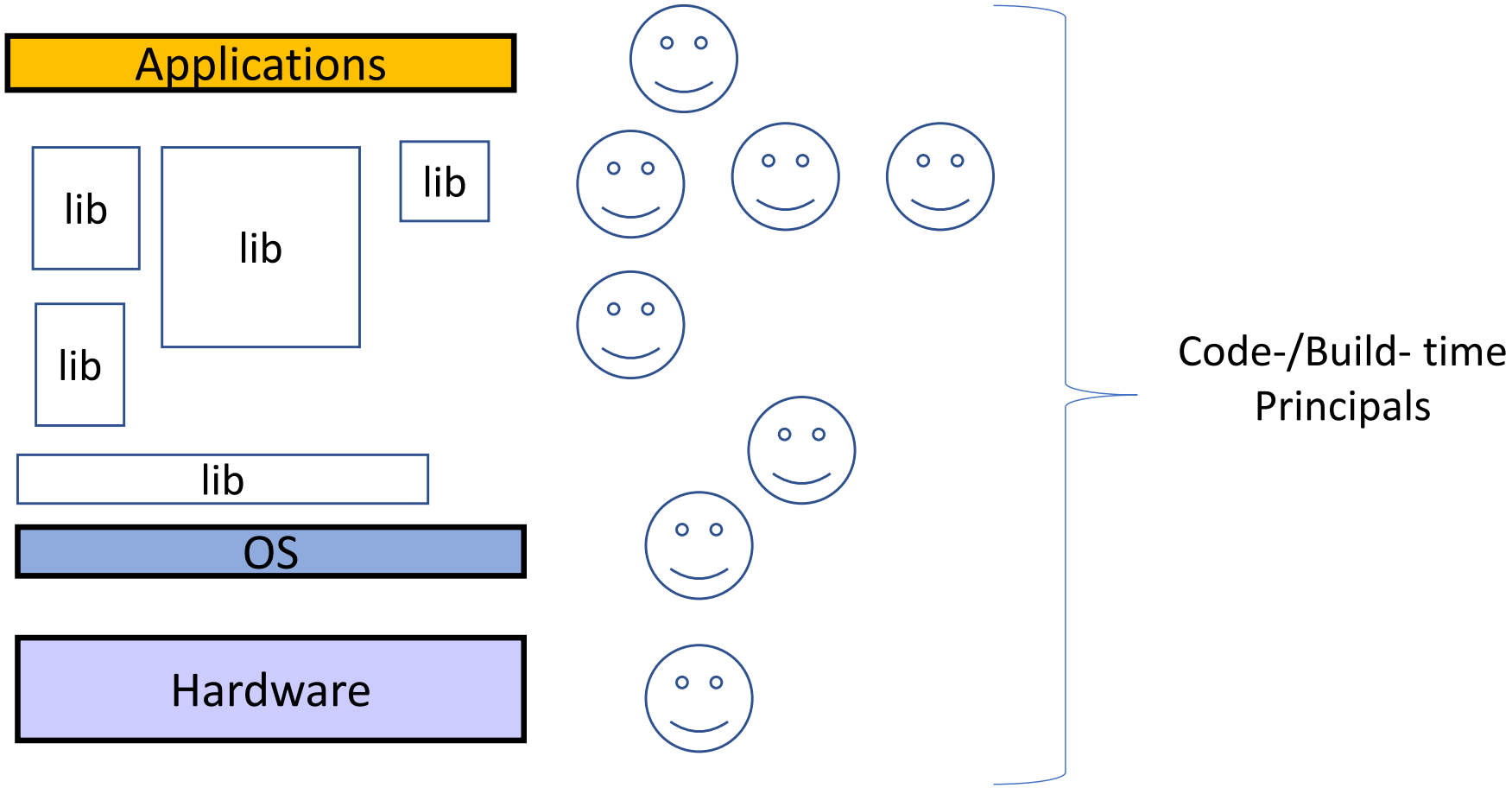
The most general way "to ask" is to let the layer above run arbitrary code – to invoke a method in the layer above. The author of that code can (a) have made a static decision (one made at coding time) or can (b) make a dynamic decision (one computed at run time). The code can do anything that can be done on the system.

This is an "upcall" – a layer (OS) invokes the layer above it (app).

(It's very similar in many ways to how code detecting an error passes control back up to whatever calling code is interested in handling the error)

4. Who Sets Policy, and When?

User – launch-time principal



4. Who Sets Policy?

- It isn't completely clear "who" is in the best position to make efficient policy decisions
  - The OS knows about the overall state of the system
  - The app author knows about the general needs of the application
  - The running app instance could possibly know about the very specific needs of that particular run of the application
- So, we're of two minds
  - On the one hand, we sometimes want to defer policy decisions to higher levels
  - On the other hand, those higher levels often don't want to have to implement policy and would prefer to use a library (including the OS itself), perhaps handing parameters to that library for some amount of customization
- Moral
  - deferring policy is a recurrent and probably unfamiliar idea
  - an important, general way to do it is to run code provided by the higher level whenever some decision is needed

5. Interposition / Virtualization

- Evolution is much more common than revolution
- As computer systems evolve, it's important that already existing software continues to run without modification
  - No one will run a new OS if no apps can run on it
  - No apps will be developed for an OS that no one is running
- One handy way to provide backward compatibility is to preserve an interface but change its implementation
  - That's "interposition"
- Example: virtual memory
  - Existing interface: machine instructions issue memory addresses to load and store values from memory at those addresses
  - Updated system: apps issue exactly the same memory addresses, but we use them just as names for values, not physical memory addresses. The page table gives us great flexibility in how to actually manage the address name space.

5. Interposition / Virtualization

- "Virtualization" is a broadly used term without a precise meaning
- It originally meant to take some existing interface and replace its implementation with something else
  - "Virtual machine" originally was this idea applied to the hardware interface
  - The interface provided by actual hardware was instead implemented in software
  - Code that ran on real hardware would run without modification on the virtual machine
  - The HW interface is a very powerful one, because all code that could run (including the OS) could run on the virtual machine
- These days "virtualization" is used more broadly, to talk about imposing a new layer (more or less)
  - We say, very informally, that the file system virtualizes the disk
    - The disk hardware interface is replaced by the file system interface
  - We say the Java virtual machine virtualizes the operating system and hardware (and so makes the Java app portable)

6. Naming

- The OS creates abstractions
  - E.g., processes, files, open files, thread, address spaces, ...
- To be able to manipulate an instance of an abstraction, we have to have a name for it
  - “Kill process X”
- Integers are convenient names for use by computers/code. Character strings are convenient names for use by people.
  - “int x” in a C program vs “memory location 0x0783EF20” at hardware level

- Names have scope
  - The region over which they are meaningful
  - For instance, /home/zahorjan/myfile.txt has scope of the machine I use that name on, while http://cnn.com has scope of the Internet (roughly)

6. Naming

- Choosing scope appropriately is more important/difficult than it might first seem
- The scope establishes boundaries that can then be hard to cross
  - E.g., process 1911 on this machine can't easily be moved to attu.cs because the name 1911 may be used by some other process there

- Conversely, narrowing the scope of names is a technique used to "virtualize," which provides both protection/isolation and allows replication of the virtual resources while sharing one set of underlying resources
  - Virtual memory: The system-wide namespace of memory addresses becomes a per-address-space set of names
  - Virtual machine: The system-wide namespace of all hardware resources becomes a namespace-per-VM
  - Relatively recent work has been on "namespace isolation" to provide what feel like multiple execution environments on a single hardware platform without all the overhead of providing complete separate virtual machines (Containers)

- Suggested exercise
  - Telephone numbers are a global namespace
  - What if anyone could create a new scope for telephone numbers?

7. Synchronization

- Two events are *synchronized* if one definitely happens before the other
  - Otherwise, they are *unsychronized*
- The execution of instructions by a single thread are synchronized
  - Instructions happen in program order, as seen by that thread
  - *Instructions may not happen in program order, as seen by some other thread*

- Synchronization is a remarkably complicated topic
- To deal with it reliably, we require some simplified constraints
- One is that synchronization between threads takes place using only certain well-defined synchronization operations (e.g., locks)
- Next, we are required to get correct results only for programs that are "correctly synchronized" under our simplifying assumptions
- (Further) Experience with synchronization is an important part of the course

Next time

- Architectural support for OS