A Very Brief History of OSes

ICS332

Operating Systems

Henri Casanova (henric@hawaii.edu)

The Pre-History

- Early OSes were just libraries
  - Just some code as wrapper around tedious low-level stuff that users just didn't want to write
  - No real abstractions
  - No virtualization
  - No resource allocation
- One program ran at a time, controller by a human operator
  - This was known as "batch mode"
  - A big challenge was that the machine shouldn't be idling, due to high cost
  - Absolutely no interactivity

System Calls

- Beyond Libraries
  - People realized that user code should be differentiated from kernel code, and that kernel code should be “special”
  - In old OSes, any program could do anything to any hardware resource
  - So a bug in your code could crash the computer/ devices, which reduced productivity and caused anxiety :)
- Development of the concept of a system call
  - Programs now written as “please OS, do something for me” as opposed to as “I’ll do it myself”

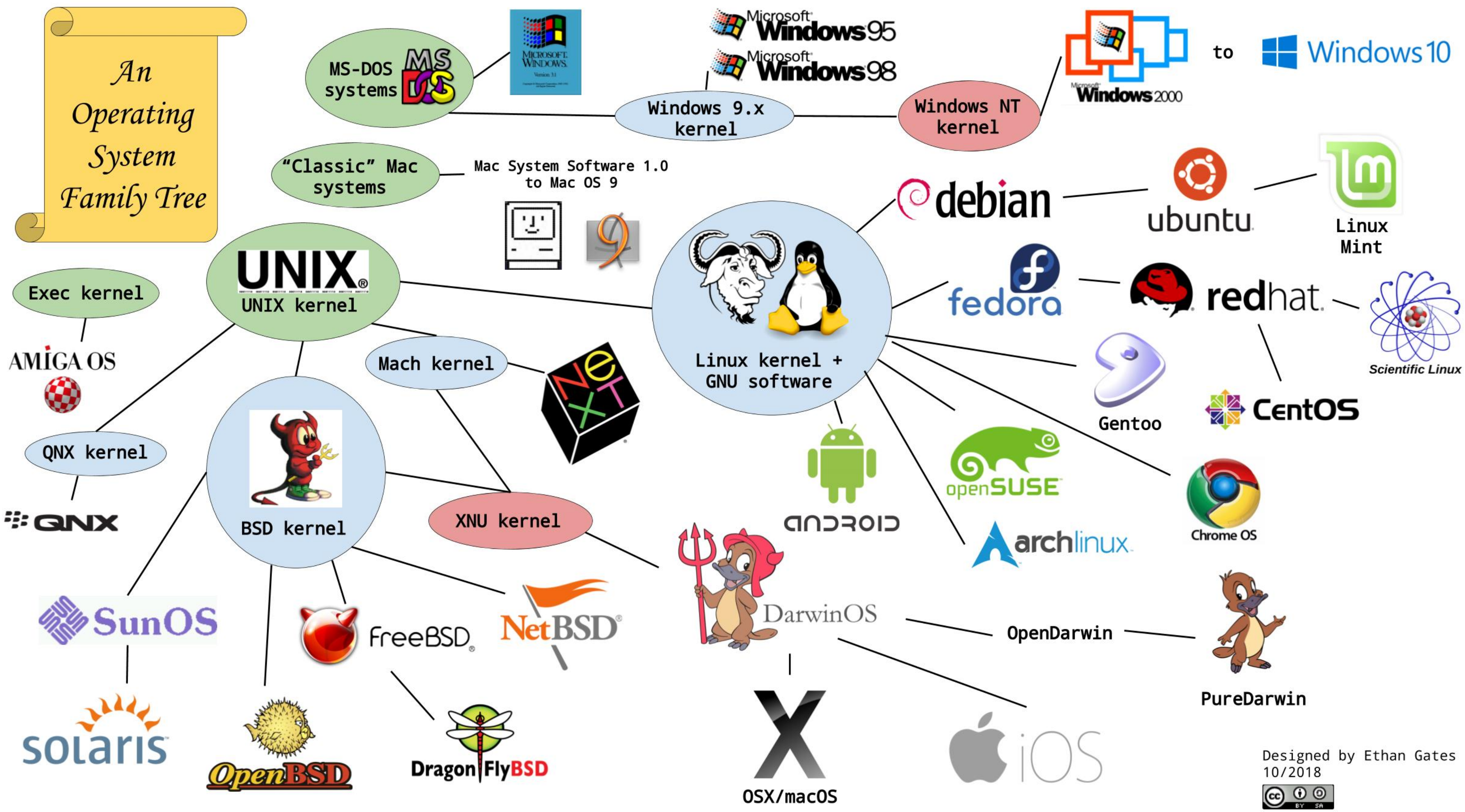
Multiprogramming

- Multiprogramming led to the first “real OSes” (from our modern perspective)
- Came about to improve CPU utilization (while program #1 is idling, program #2 should be able to utilize the CPU)
- Development of context-switching and memory protection (which we’ll discuss at length)
- Beginning of concurrency
- Development of UNIX
- Make sure you read the “Importance of UNIX” box in OSTEP 2.6 (page 15)

The Modern Era: PCs

- The PC changed the world (IBM, Apple)
- The OSes on these machines were... lacking
- Many see them as a step backward when compared to UNIX
  - Worse memory protection (MS-DOS)
  - Worse concurrency (MacOS v9)
  - See the “Unfortunately, …” paragraph in OSTEP 2.6 :)
- But eventually, the good features of older OSes crept back in
  - Mac OS X has UNIX as its core
  - Windows NT was radically better than its predecessors
- The OSes you use (and like?) today have more to do with those from the 1970’s than those from the 1980’s
  - My Apple laptop and my Android phone basically run UNIX!
- Make sure you read the “And then came Linux” box in OSTEP 2.6 (page 16)

OS Genealogy



Unmodified from https://github.com/EG-tech/digipres-posters

OS Design Goals

- Abstraction: to make the use of the computer convenient
  - Building abstractions is of what Software Development is about
  - Designing good abstractions will be part of your careers
- Performance: Minimize OS overhead (time, space)
  - Often conflicts with the previous goal
- Protection: Programs must execute in isolation
  - Comes from virtualization
- Reliability: The OS must not fail
  - Thus OS software complexity is a concern (e.g., is it worth adding 2,000 lines of complex kernel code to improve something by some epsilon?)
- Resource efficiency: The OS must make it possible to use hardware resources as best as possible so that there is little waste

- There is no “best design” to achieve all the above, but many lessons have been learned and we have converged to a common set of widely accepted principles

Mechanism / Policy

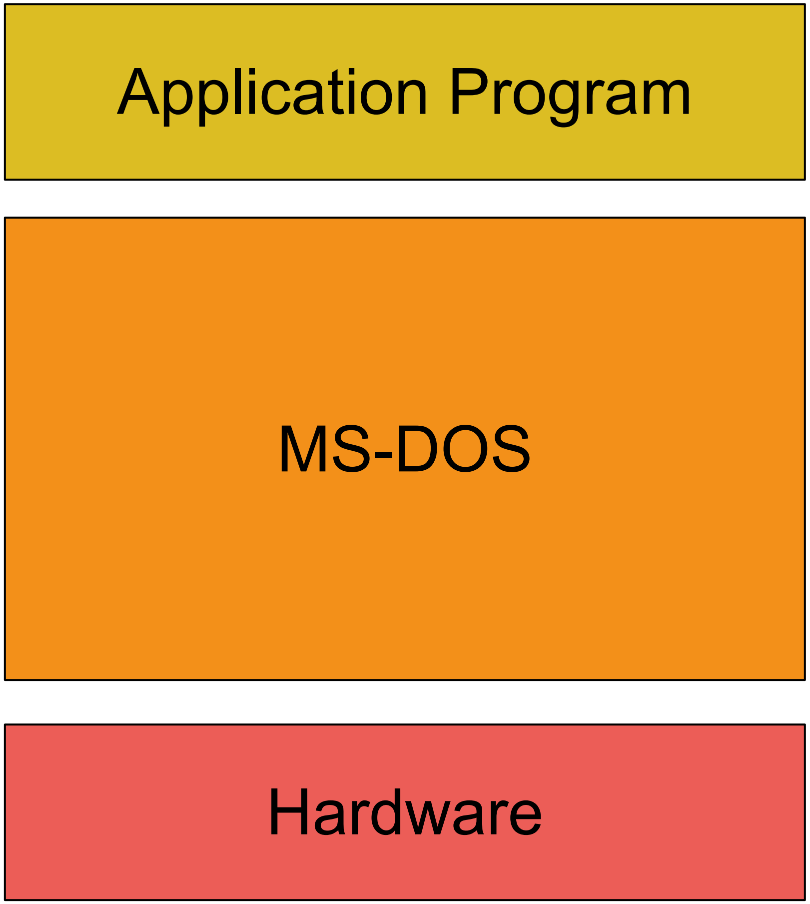
- One ubiquitous principle: **separating mechanisms and policies**
  - Policy: what should be done
  - Mechanism: how it should be done (e.g., API functions)
- Separation is important so that one can change policy without changing the mechanisms
- Mechanisms should be *low-level enough* that many useful policies can be built on top of them
  - e.g., Too high-level APIs may simply not allow you do do what you need to do in your program
- Mechanisms should be *high-level enough* that implementing useful policies on top of them is not too labor intensive
  - e.g., Too-low-level APIs may require you to write hundreds of lines of code that you'd rather not have to write/debug
- Some OS designs take this separation principle to the extreme (e.g., Solaris), and others not so much (e.g., Windows 7)

Separating Mechanisms and Policies

- This idea of "separating of mechanisms and policies" probably sounds pretty vague/abstract/useless to many of you
  - As it did to me in college back when dinosaurs walked the earth
- Yet, you will be confronted to this issue in your future careers
  - And it's even on Wikipedia
- But until you've worked on a big system and/or worked on designing APIs for others to use it's hard to really get it
  - Designing good APIs is WAY harder than you think!
  - An OS course is full of fundamental/useful stuff that one realizes is fundamental/useful often years after taking the course
  - I'll do my best to try to avoid this, but there are limits on how much "this is important" jumping up and down I can do (convincingly)

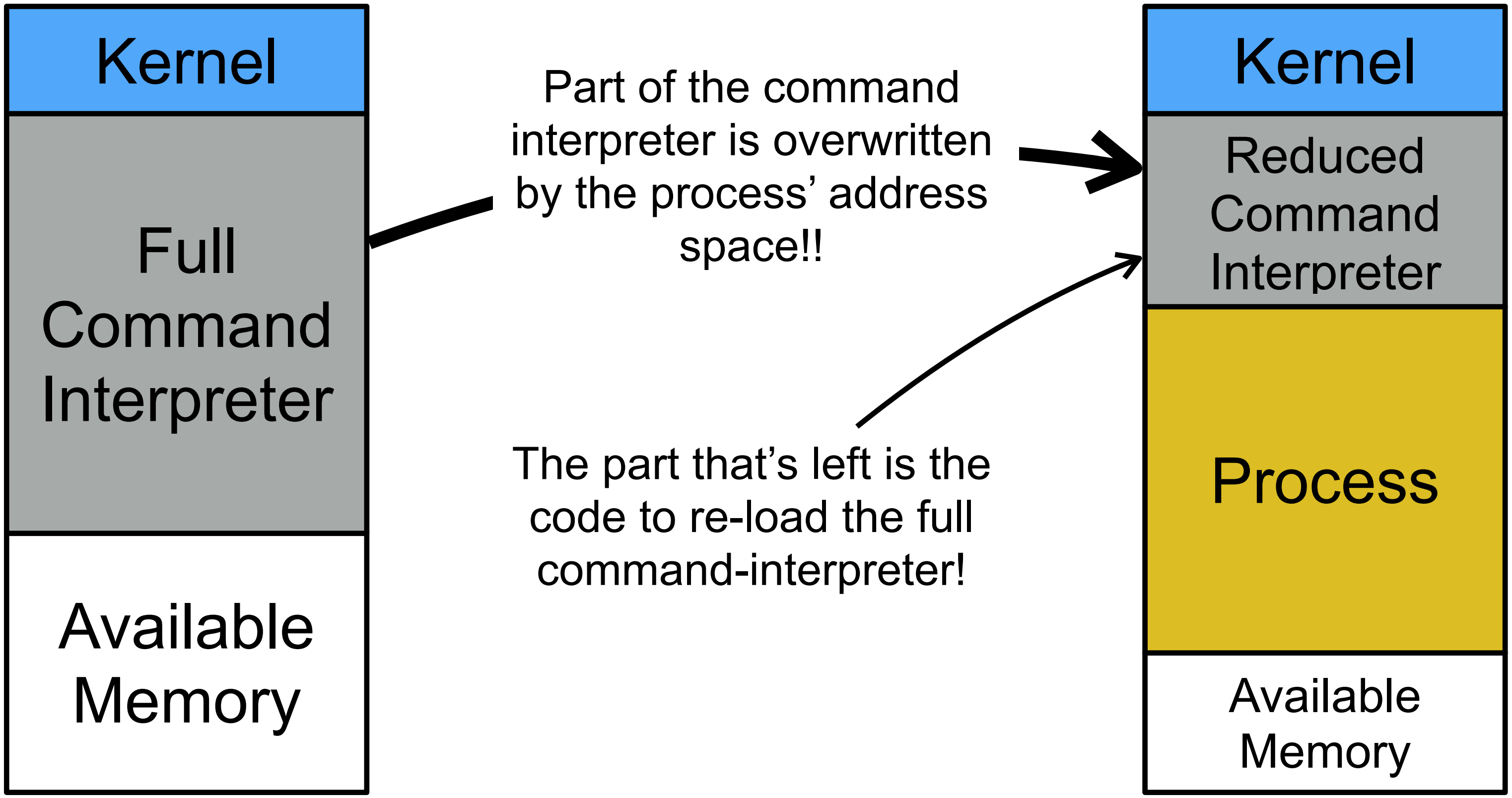
Early OS Designs: Monolithic

- Early OSes (and MS-DOS)
- No precisely defined structure
- New “features” piled upon old ones: snowball effect (usually breaking, difficult maintenance, …)
- MS-DOS was written to run in the smallest amount of space possible, leading to poor modularity, separation of functionality, and security
  - e.g., user programs can directly access some devices
  - e.g., no difference in execution of user code and kernel code (soooo insecure! we’ll see how this is done today...)



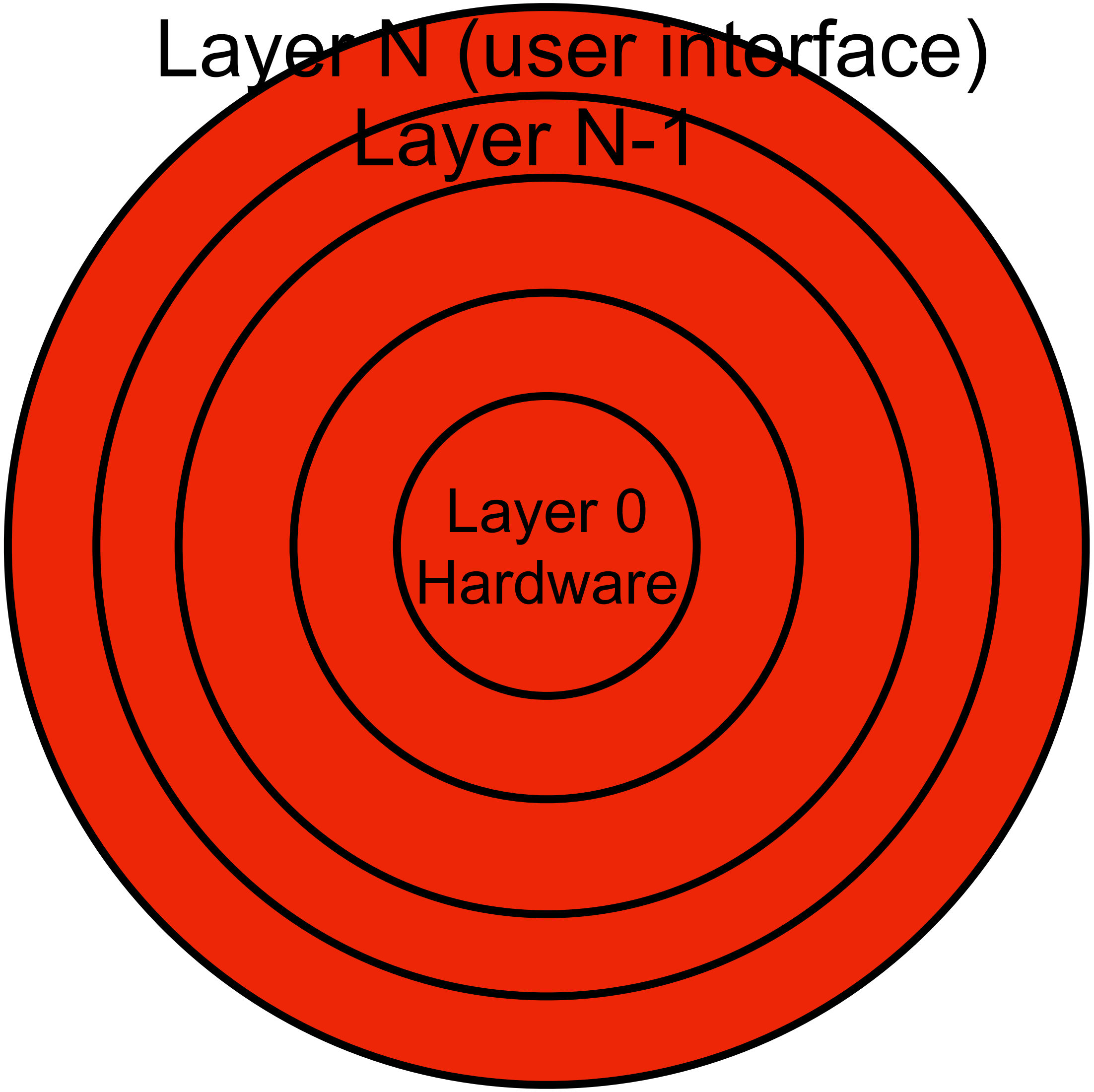
The MS-DOS Memory Trick

- In MS-DOS, due to memory limitations, user programs used to wipe out (non-critical) parts of the OS to get more RAM for themselves



- It's hard for us to fathom the constraints developers worked with in that era…

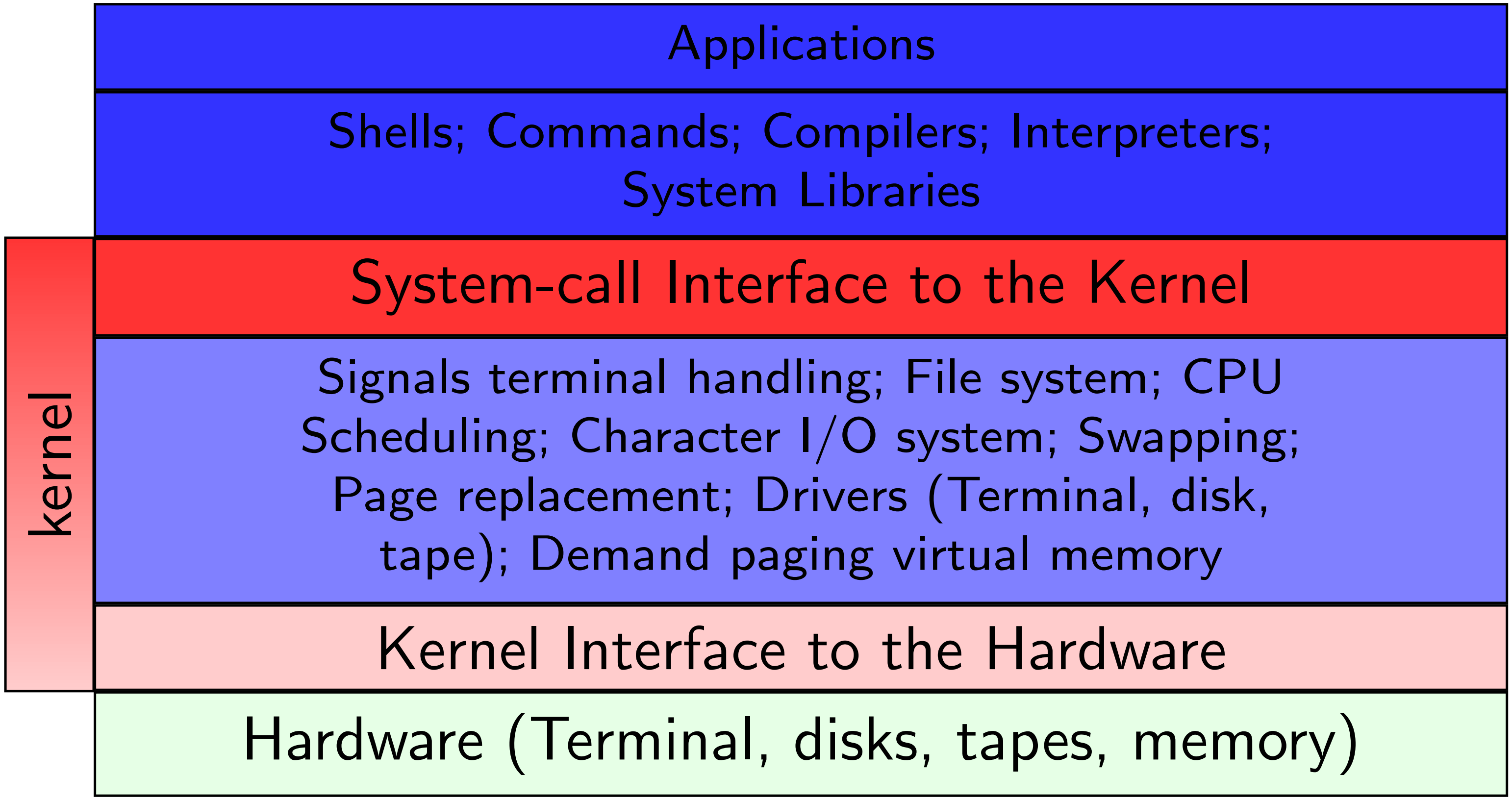
OS Design: Layered



- Layer i only calls layer i-1
- “Looks” like a clean design, but it’s fraught with difficulties
- Deciding what goes in each layer is hard due to circular dependencies
- Deciding on the best number of layers is hard
  - Too many: high overhead
  - Too few: bad modularity

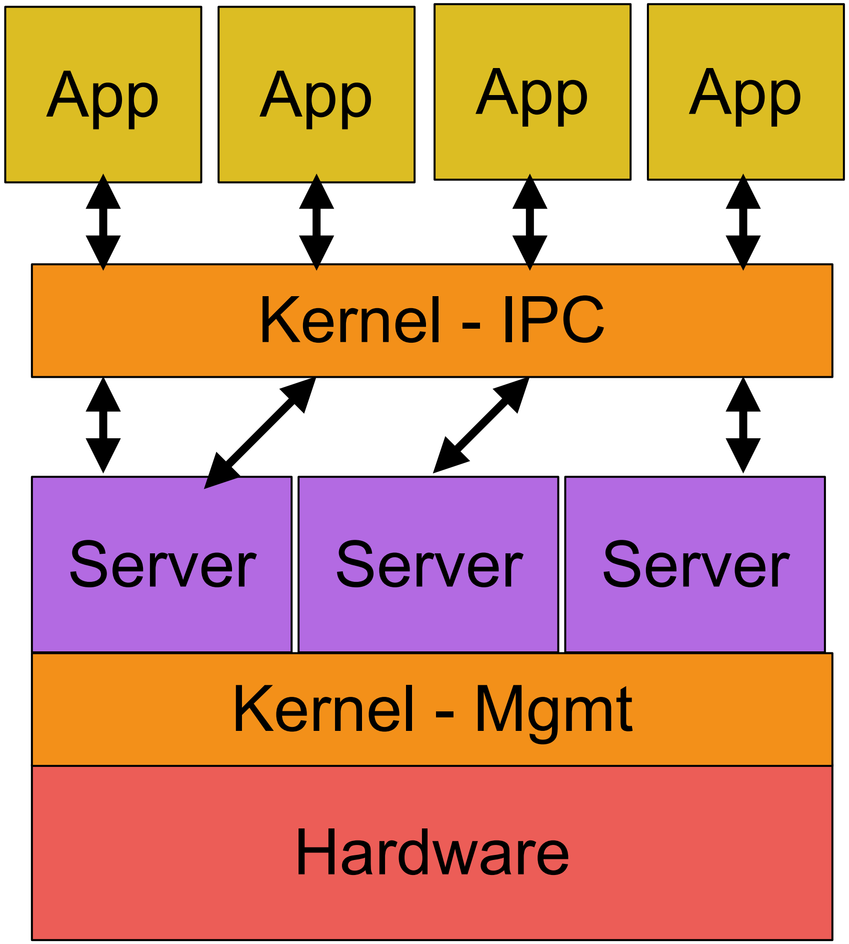
OS Design: Layered

- The First UNIX has some layers
- But the kernel was still very large and difficult to maintain evolve



OS Design: Microkernels

- Concept: 1967; Practice: 1980s
- Basic idea: Remove as much as possible from the kernel and put it all in system programs
- The Kernel only does essential management (process and memory), and basic IPC (Inter-Process Communication)
- Everything is implemented in client-server fashion
  - A client is a user program
  - A server is a running system program, in user space, that provides some service
  - Communication is through the microkernel communication functionality
- This is very easy to extend since the microkernel does not change



OS Design: Microkernels

- 1980s: First LANs
- Led to a "Everything must be distributed" philosophy
  - Client-Server based architectures will solve all issues
  - So the kernel must have a client-server architecture as well
- Mach microkernel (Carnegie Mellon University): Research Project
  - Precursor of Windows NT, MacOS, Linux
- Major issue: increased overhead because of IPC
  - Windows NT 4.0 had a micro-kernel (and was slower than Windows 95)
  - Oops... Microsoft put things back into the Kernel
  - Windows XP (and 10 apparently) is closer to monolithic than microkernel
- Experts were very opinionated about what should be in the kernel and what should not
  - Development/research around microkernels stopped in the 2000s
  - But we know that a huge kernel is a problem!

OS Design: Modules

- Take good things from all kernel design
- Most modern OSes implement modules
  - Use an “object-oriented” approach
  - Each code component is separate
  - They talk to each other over known APIs
  - This is just good software engineering
- Loadable modules: Load at boot time or at runtime when needed
- Like a layered interface, since each module has its own interface
- Like a microkernel, since a module can talk to any other module
  - But communication does not use IPC, i.e., low overhead
- Bottom-line: advantages of microkernels without the poor performance
- Pioneer: Solaris (Sun Microsystems, then Oracle)
  - Small core kernel, 7 default modules loaded at boot, other modules loadable on the fly whenever needed
  - Most agree it was a “nice” kernel / OS

OS Design: General Principles

- No modern OS strictly adheres to one of these designs (except for educational purposes)

- The accepted wisdom
  - Don’t stray too far from monolithic, so as to have good performance
  - Modularize everything else to still be able to maintain the code base
- It’s a complicated balancing act and every kernel does it a little bit differently
  - And it’s hard to compare metrics like LOC (lines of code) because different OSs have different components “in the kernel” or “outside the kernel”

Conclusion

- OSes have a “long” history
- Lessons from past failures and successes have given us current OS designs
  - We’re lucky that we’re now after the “excitement” of competing designs
- A key design principle is Separation of Mechanisms and Policies
- Reading Assignment: OSTEP 2.5-2.6

- We’ll have a quiz on this module next week