CSE 451: Operating Systems

Autumn 2020

Module 3

Operating System

Components and Structure

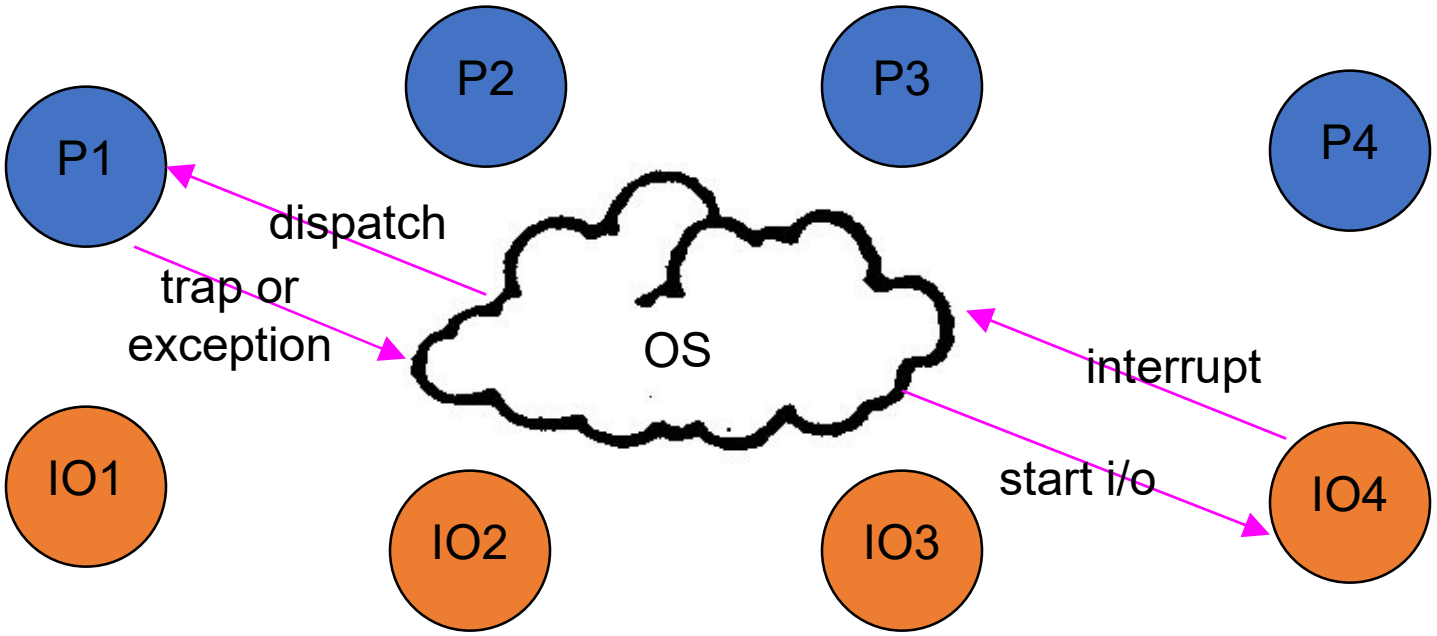
John Zahorjan

Lecture Questions

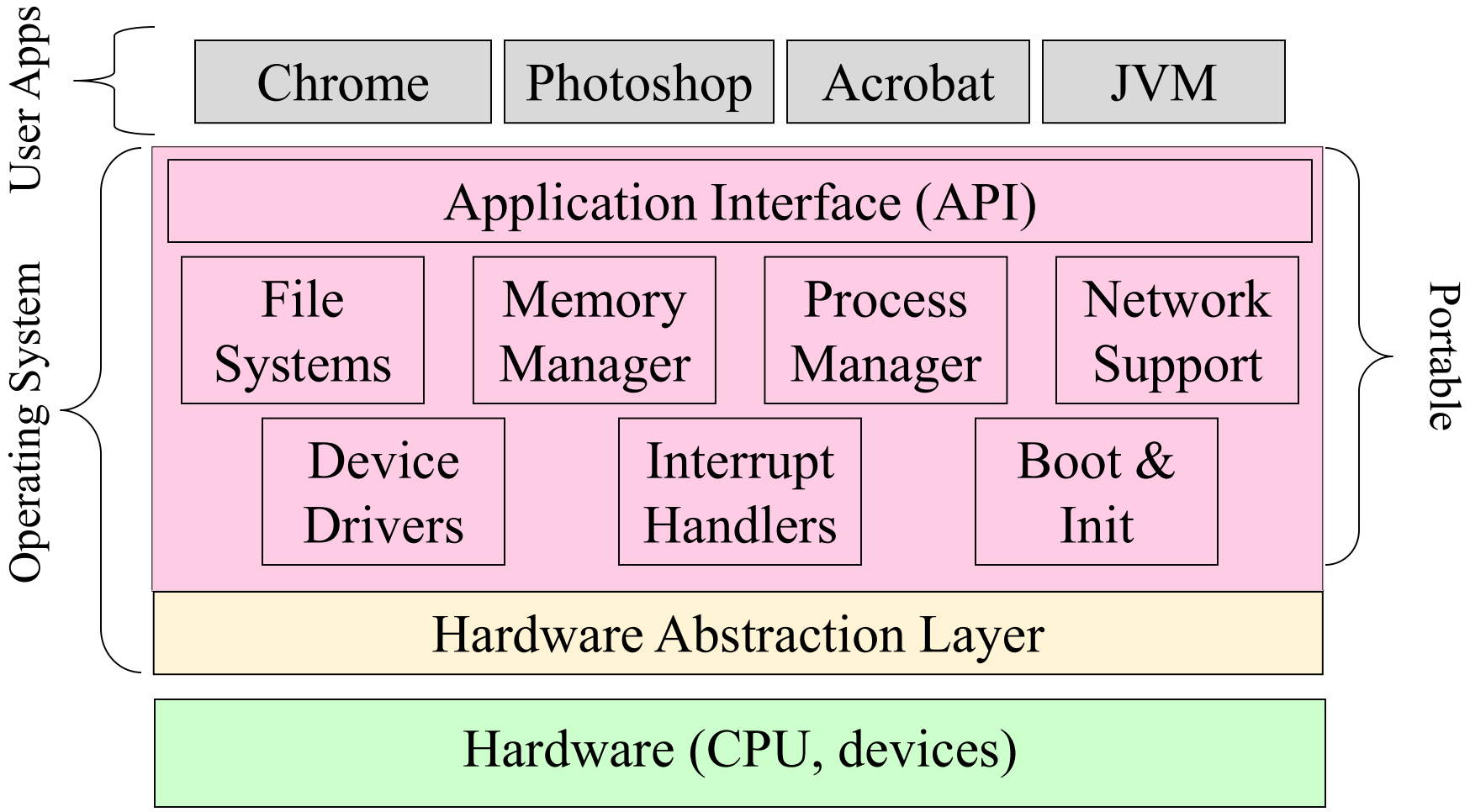
- What are some important components of an OS?
- “Program” vs. “Process” vs. “Thread”
- What are the (execution) states of a process?
  - What does it mean to be “blocked”?
- What is the relationship between processes and address spaces?
- What’s special about device drivers?
- What’s the difference between a disk and a file system?
- What is the “scope” of a namespace and why does it matter?
- If there were no adversaries would I need the OS to provide protection?
- What does “OS structure” mean and why do we care?
- What alternatives to monolithic have been tried?
- Why would you build a software layer that presents an interface identical to a hardware layer?
- What is the real goal, and can you get it some other way?

OS structure

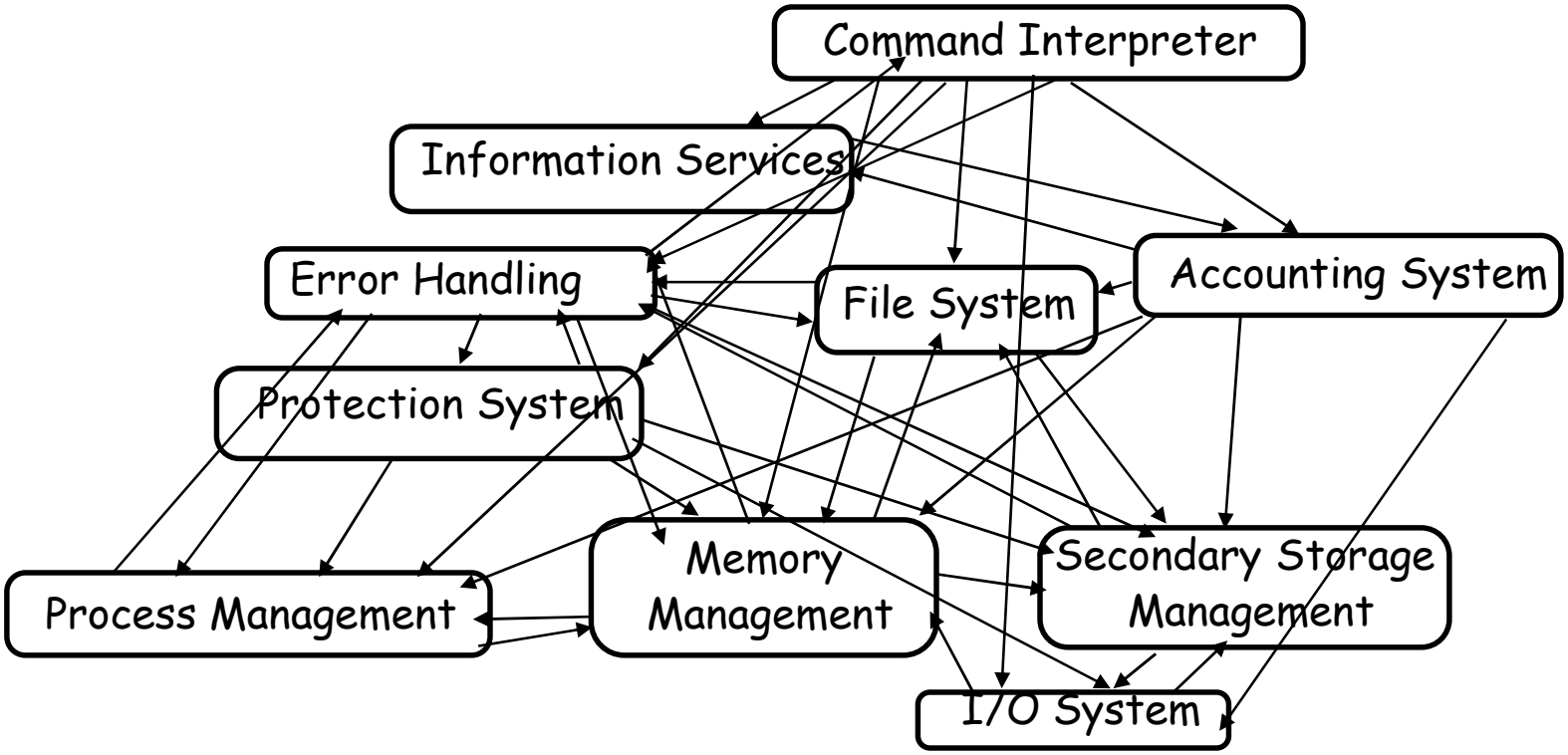
- The OS sits between application programs and the hardware
  - it mediates access and abstracts away ugliness
  - programs request services via traps or exceptions
  - devices request attention via interrupts



OS Static Software Structure



OS Dynamic Call Structure



- Is this the only possible structure?
- Is this the best structure?

Part I: Major OS components

1. Processes / Threads
2. Memory / Address Spaces
3. I/O
    1. secondary storage
    2. file systems
4. Protection
5. Application support / Syscall interface
    1. shells (command interpreter, or OS UI)
6. Windowing system
7. Networking

1. Process management

- An OS executes many kinds of activities:
  - users' programs
  - background jobs or scripts
  - system programs
    - print managers, name servers, file servers, network daemons, …
- Each of these activities is encapsulated in a process
  - a process is a running program
  - a process has an execution context
    - PC, registers, VM maps, OS resources (e.g., open files), … (there's more)
    - plus the runtime instance of the program itself (code and data)
  - the OS's process module manages these processes
    - creation, destruction, scheduling, …
  - a process is a unit of failure
    - What other units of failure might exist?

Processes vs. Threads

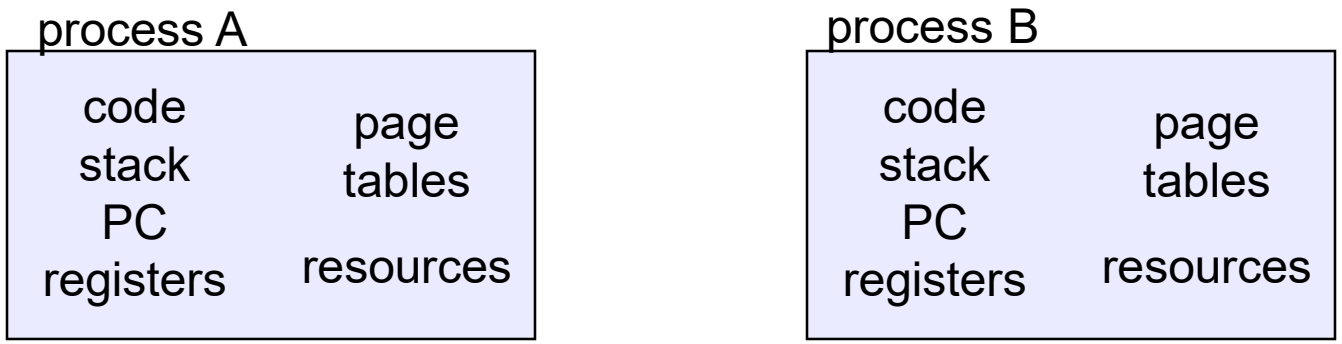
- Soon, we will separate the “thread of control” aspect of a process (program counter, call stack) from its other aspects (address space, open files, owner, etc.). And we will allow each {process / address space} to have multiple threads of control.

- But for now – for simplicity and for historical reasons – consider each {process / address space} to have a single thread of control.

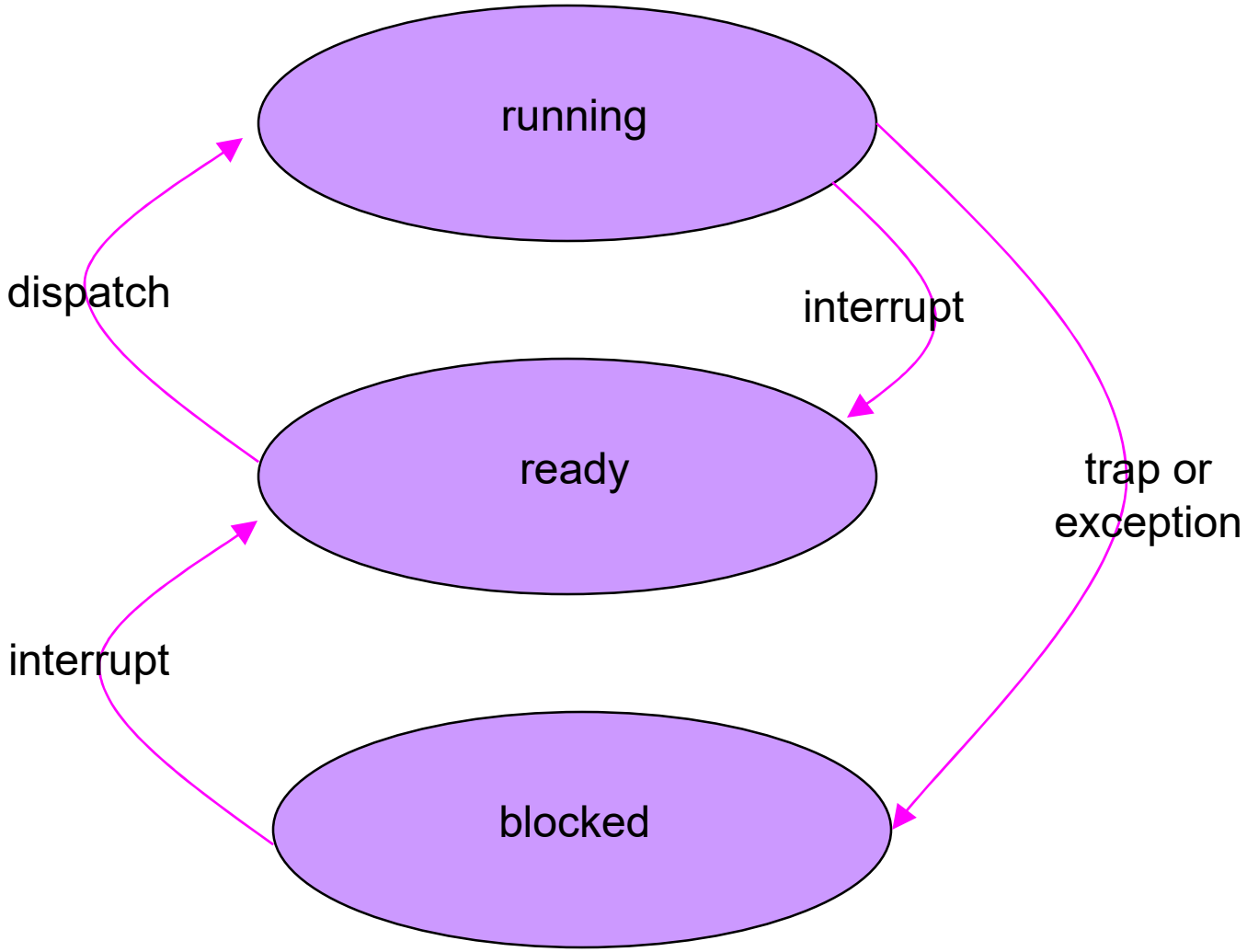
Program / processor / process

- Note that a *program* is totally passive
  - just bytes on a disk that encode instructions to be run

- A *process* is an instance of a program being executed by a (real or virtual) processor
  - at any instant, there may be many processes running copies of the same program (e.g., an editor); each process is separate and (usually) independent
  - Linux: `ps -auwx` to list all processes



States of a user process (thread)



Process operations

- The OS provides the following kinds operations on processes (i.e., the process abstraction interface):
  - create a process (createprocess, fork/exec)
  - delete a process (kill, exit)
  - suspend a process (kill, sched_yield)
  - resume a process (kill)
  - clone a process (fork, clone)
  - inter-process communication (kill, pipe, mmap, …)
  - inter-process synchronization (wait, flock, sem_open, ...)

2. Memory/Address Space Management

- Primary memory is the directly accessed storage for the CPU
  - programs must be resident in memory to execute
  - memory access is fast
    - (cse 351: memory access times are complicated…)
    - (UMA (uniform memory access) vs. NUMA (non-uniform memory access) architectures)
  - but memory doesn't survive power failures

- OS must:
  - allocate memory for address spaces (processes)
  - deallocate space when needed by rest of system
  - maintain mappings from virtual addresses to physical
    - page tables
  - switch CPU context among addresses spaces

  - decide how much physical memory to allocate to each process
  - decide when to remove a process from memory

Mechanism

Policy

3. I/O

- A big chunk of the OS kernel deals with I/O
- The OS provides a standard interface between programs (user or system) and devices
  - file system (disk), sockets (network), frame buffer (video)
- Device drivers are the routines that interact with specific device types
  - encapsulates device-specific knowledge
    - e.g., how to initialize a device, how to request I/O, how to handle interrupts or errors
    - examples: PCIe device drivers, Ethernet card drivers, video card drivers, sound card drivers, …
- Device drivers are written by the device company
  - but execute in the OS address space and run at high privilege
    - Why?

4. Secondary storage

- Secondary storage (spinning disk, ssd, usb drives) is persistent memory
  - survives power failures (hopefully)

- Routines that interact with disks are typically at a very low level in the OS
  - used by many components (file system, VM, …)
  - may handle scheduling of disk operations, error handling, and often management of space on disks
    - a trend is for disks to do more of this themselves

- Disk device as storage vs. file system
  - device => raw storage
  - file system => layer of abstraction providing structured storage
    - Analogy: array vs. a linked list embedded in an array

5. File systems

- Secondary storage devices are crude and awkward
  - e.g., “write a 4096 byte block to sector 12”
- File system: a more convenient abstraction
  - hardware independent interface presented up to apps
  - hardware dependent implementation looking down to hw
- FS defines logical objects, like files and directories
  - files represent *values*, stored somewhere on disk
  - directories represent file *meta-data*, like name, owner, creation time, …
  - The disk device knows nothing about files
- FS defines operations on objects, like creat, read, write, stat
- FS defines a name space
  - The scope of the name space has traditionally been the entire system
  - The file system namespace has been used to name non-file-y things
    - If I want to share things among processes, the names used must have scope that includes all those processes

“File system”

- The term “file system” has at least three common meanings
  - The generic notion of providing a more convenient abstraction layered on some storage device
  - A particular software implementation of that generic idea, e.g., NTFS or FAT or ext4 or zfs
  - A self-contained, and so physically portable, bunch of bits on some storage device, written in the format used by some file system software
    - File systems are *mountable*

File system operations

- The file system interface defines standard operations:
  - file (or directory) creation and deletion
  - manipulation of files and directories (read, write, extend, rename, protect)
  - copy
  - lock

- File systems may also provide higher level services
  - accounting and quotas
  - backup (must be incremental and online)
  - (sometimes) indexing or search
  - (sometimes) file versioning
  - (sometimes) encryption

- File systems are also concerned with lower level characteristics
  - Performance may be strongly affected by hardware characteristics
  - Failure resilience maybe strongly affected by hardware characteristics

6. Protection

- Protection is a general mechanism used throughout the OS
  - all resources needed to be protected
    - memory
    - processes
    - files
    - devices
    - CPU time
    - network bandwidth (?)
    - …
- Protection mechanism motivations:
  - Sharing – we want to use the hardware to do more than one thing at a time
  - "I'm not perfect" -- help to detect and contain unintentional errors
  - "There are adversaries" -- preventing malicious abuses

7. Command Interpreter (shell)

- Like the OS in that it doesn't compute anything
  - Instead, it's a general facility to make it easier to run programs that do compute/do something
- A particular program that handles the interpretation of users' commands and helps to manage processes
  - user input may be from keyboard (command-line interface), from script files, from the mouse (GUIs), from voice, from gestures, from biometrics, ...
  - allows users to launch and control new programs
- On some systems, command interpreter may be a standard part of the OS
- On others, it's just non-privileged code that provides an interface the OS syscall layer
- The world changes: Windows now supports Linux, sort of

8. Windowing System

- Abstracts the display, keyboard, and mouse
  - Each window can be manipulated in a way that is independent of the others

- Output:
  - Writing to the window
  - Resizing the window
  - Possibly moving the window

- Inputs:
  - keyboard focus
  - mouse clicks
    - (x,y) position in window coordinates
    - [also (x,y) position in screen coordinates...]

9. Networking

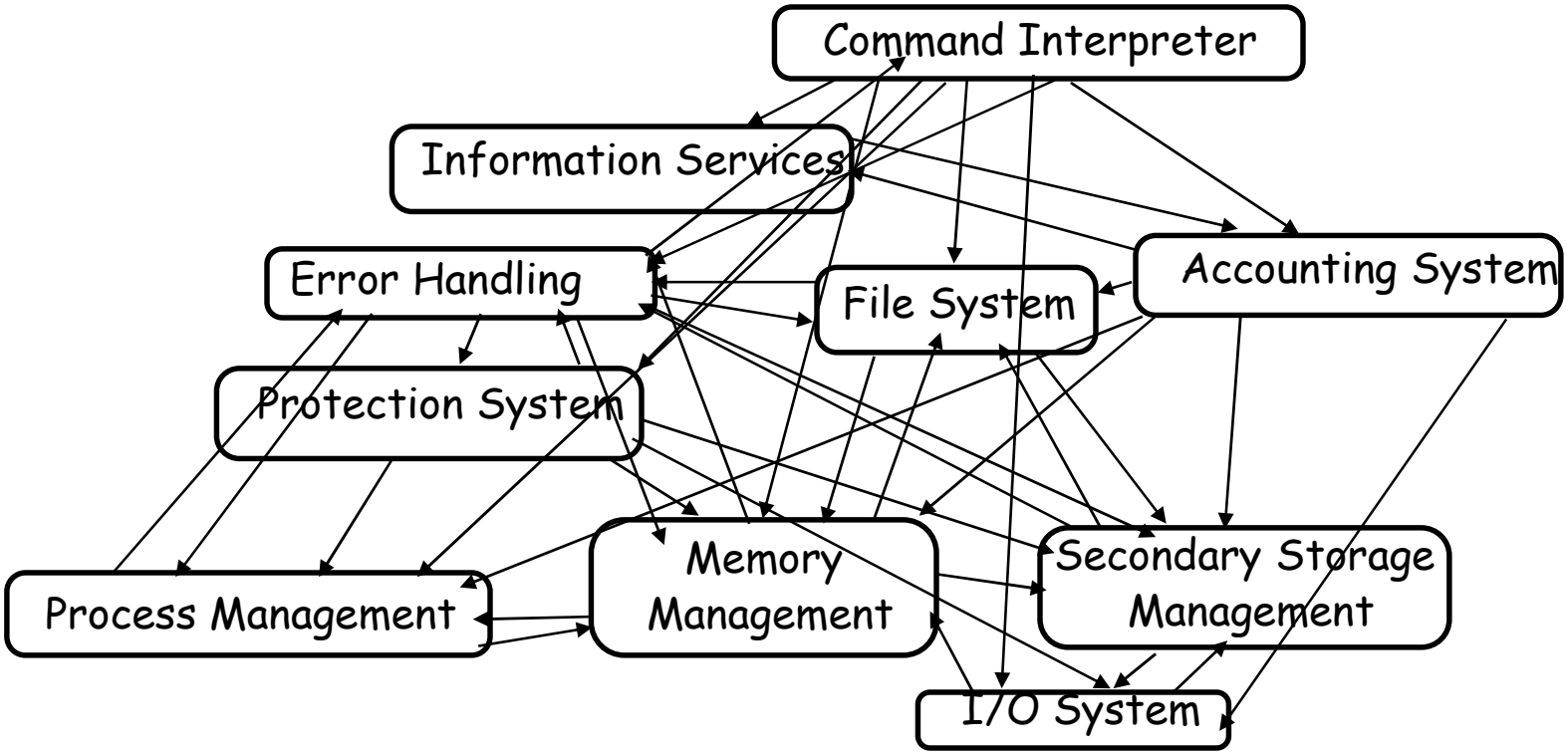
- The Internet moves bits from one machine to another machine
  - An IP address basically names a machine
    - e.g., 128.208.1.137 is attu1.cs.washington.edu
- When bytes are sent "to a machine," who receives them?
  - The operating system
- But I want to ask the web server process on the machine for a page, not talk with the OS…
  - The OS "demultiplexes" incoming messages and delivers them to processes

Networking (continued)

- The IP layer of the network stack is in charge of moving data from one machine to another
  - In a way, it abstracts the network interface card (physical connection to the network)
- The TCP layer runs on top of IP
  - It provides process to process communication, not just machine to machine
  - It abstracts the faulty IP network into an (almost) error-free network
  - Should the TCP implementation be part of the OS, or should it be a service that runs on top of the OS (kind of like a web service)?
- Why should any of this be in the OS, rather than running above it?
- Is there a need for any protocols beyond TCP?
  - How should the system support introduction of new protocols?

Part II: OS structure

It's not always clear how to stitch OS modules together



What could possibly go wrong?

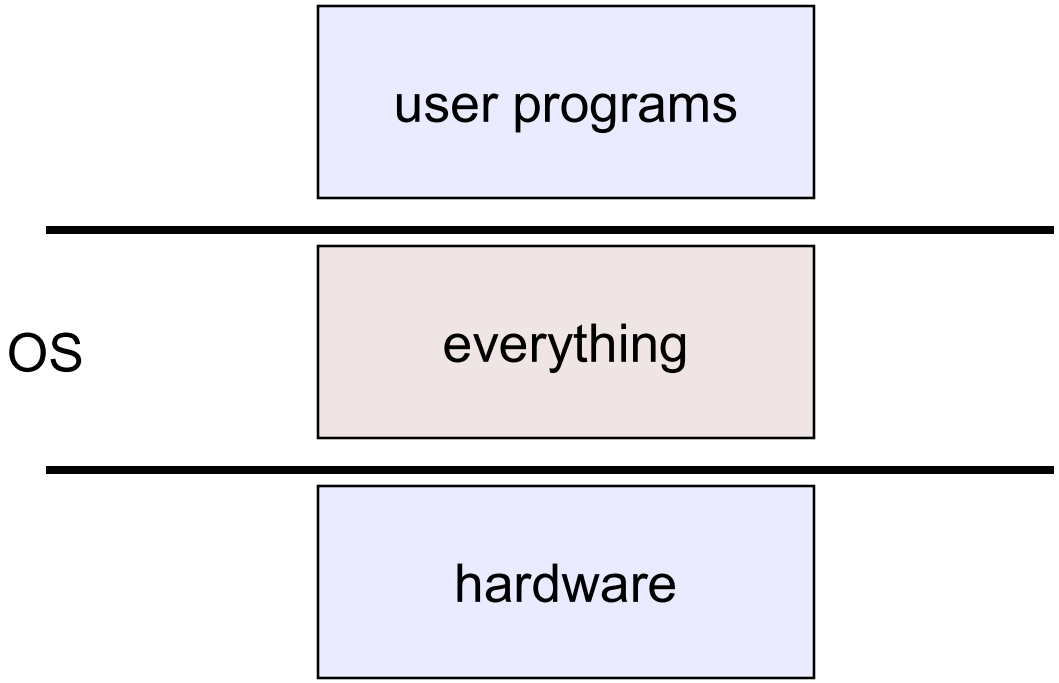
Imagine trying to update code structured like this? (xk is structured like this.)

OS structure

- An OS consists of all of these components, plus:
  - many other components
  - system programs (privileged and non-privileged)
    - e.g., bootstrap code, the init program, …
    - there is a mutually dependent relationship among the compiler and the OS and the CPU ISA
- Major issue:
  - how do we organize all this?
  - what are all of the code modules, and where do they exist?
  - how do they cooperate?
- Massive software engineering and design problem
  - design a large, complex program that:
    - performs well, is reliable, is extensible, is backwards compatible, …

Early structure: Monolithic

- Traditionally, OS's (like UNIX) were built as a monolithic entity:

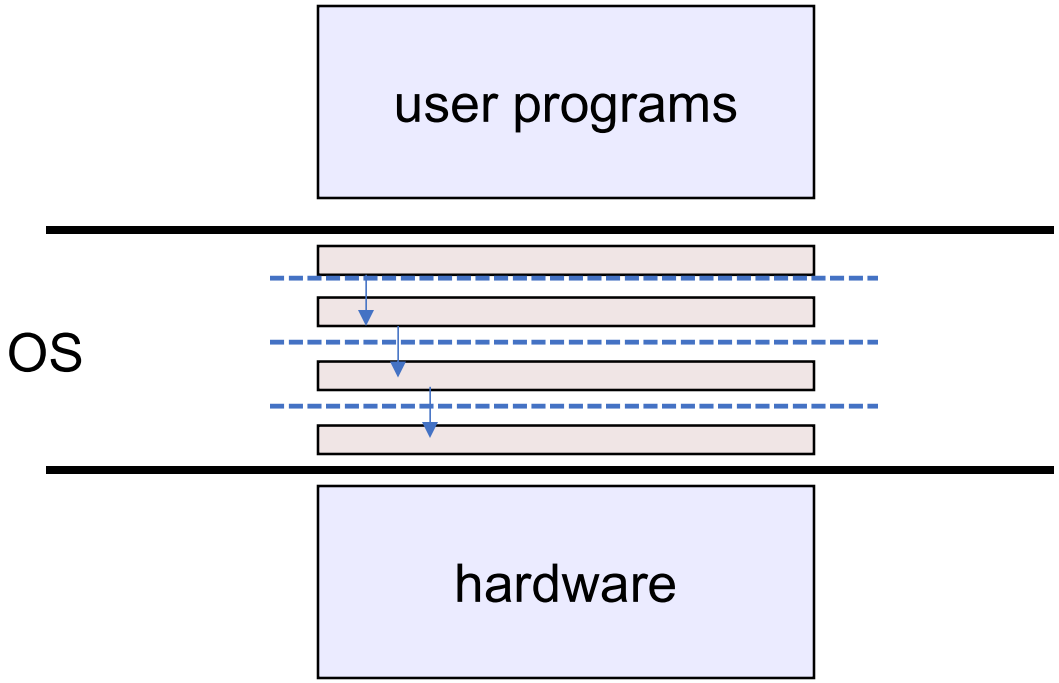


Monolithic design

- Major advantage:
  - familiar
  - cost of module interactions is low (procedure call)
- Disadvantages:
  - hard to understand
  - hard to modify / maintain
  - no help with fault containment (no isolation between system modules)
- What is the alternative?
  - find a way to organize the OS in order to simplify its design and implementation

Alternative to Monolithic: Layering

- One traditional approach is layering
  - implement OS as a set of layers
  - each layer presents an enhanced ‘virtual machine’ to the layer above



Alternative to Monolithic: Layering

- Historically, the first description of this approach was Dijkstra's THE system (1968)
  - Layer 5: Job Managers
    - Execute users' programs
  - Layer 4: Device Managers
    - Handle devices and provide buffering
  - Layer 3: Console Manager
    - Implements virtual consoles
  - Layer 2: Page Manager
    - Implements virtual memories for each process
  - Layer 1: Kernel
    - Implements a virtual processor for each process
  - Layer 0: Hardware

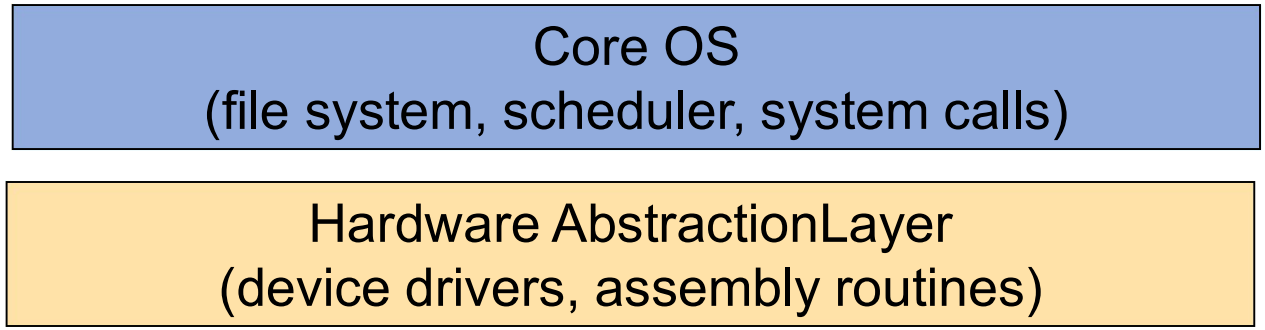
- Each layer can be tested and verified independently
  - Layering helped implementation and aided attempt at formal verification of correctness

- If you can call only down, can't go into a loop across layers

Problems with layering

- Imposes hierarchical structure
  - but real systems are more complex:
    - file system requires VM services (buffers)
    - VM would like to use files for its backing store
- Layering imposes a performance penalty
  - each layer crossing has overhead associated with it
    - static vs dynamic enforcement of invocation restrictions
- Disjunction between model and reality
  - systems modeled as layers, but not really built that way

Hardware Abstraction Layer (HAL)



- An example of layering in modern operating systems
- Goal: separates hardware-specific routines from the "core" OS
  - Provides portability of core code across different hardware
  - Improves readability (by raising abstraction layer above the details of each individual piece of hardware)

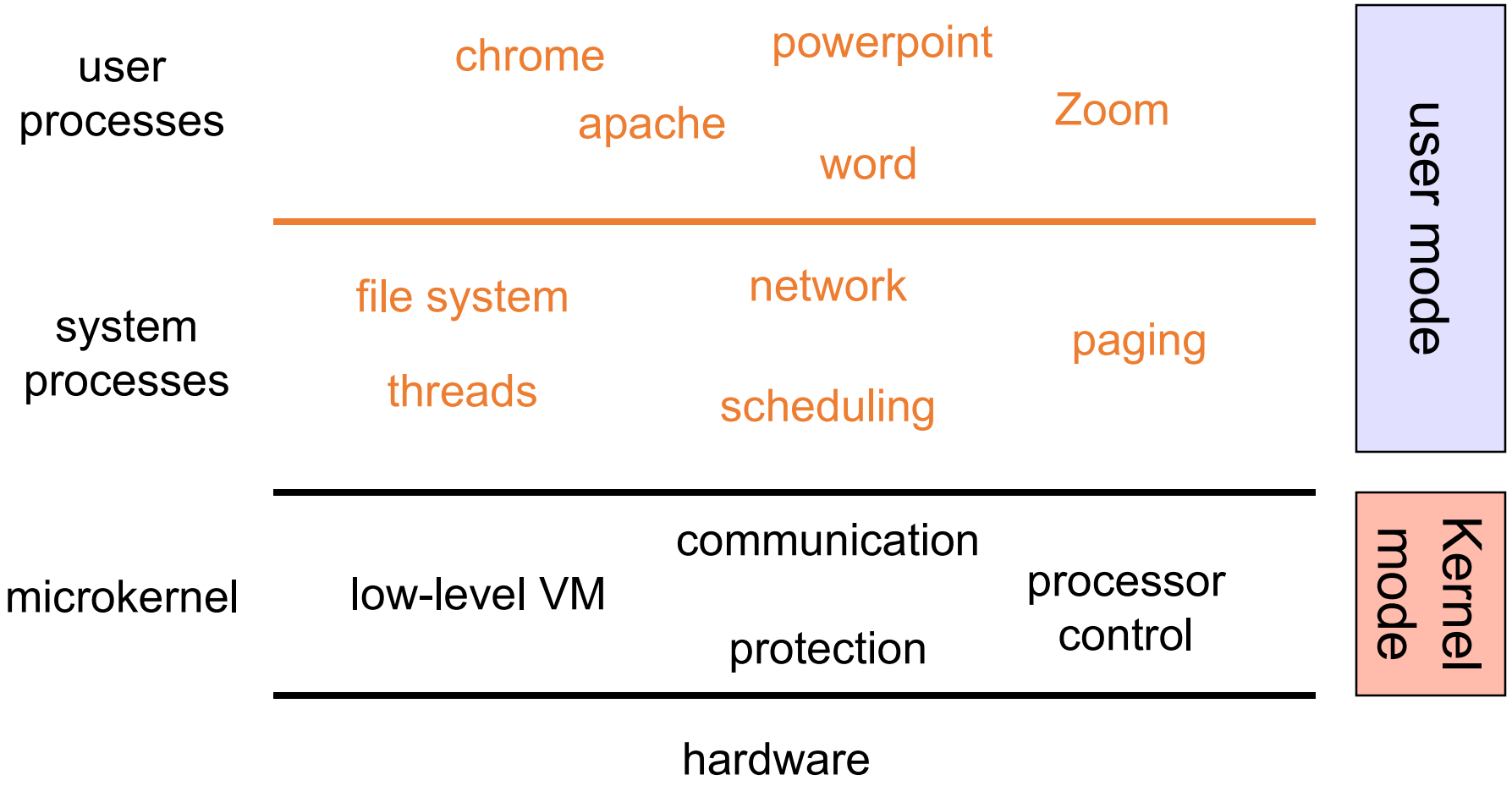
Alternative to Monolithic: Microkernels

- Goal:
  - minimize what goes into the kernel
  - implement everything else that traditionally goes in an OS as user-level processes

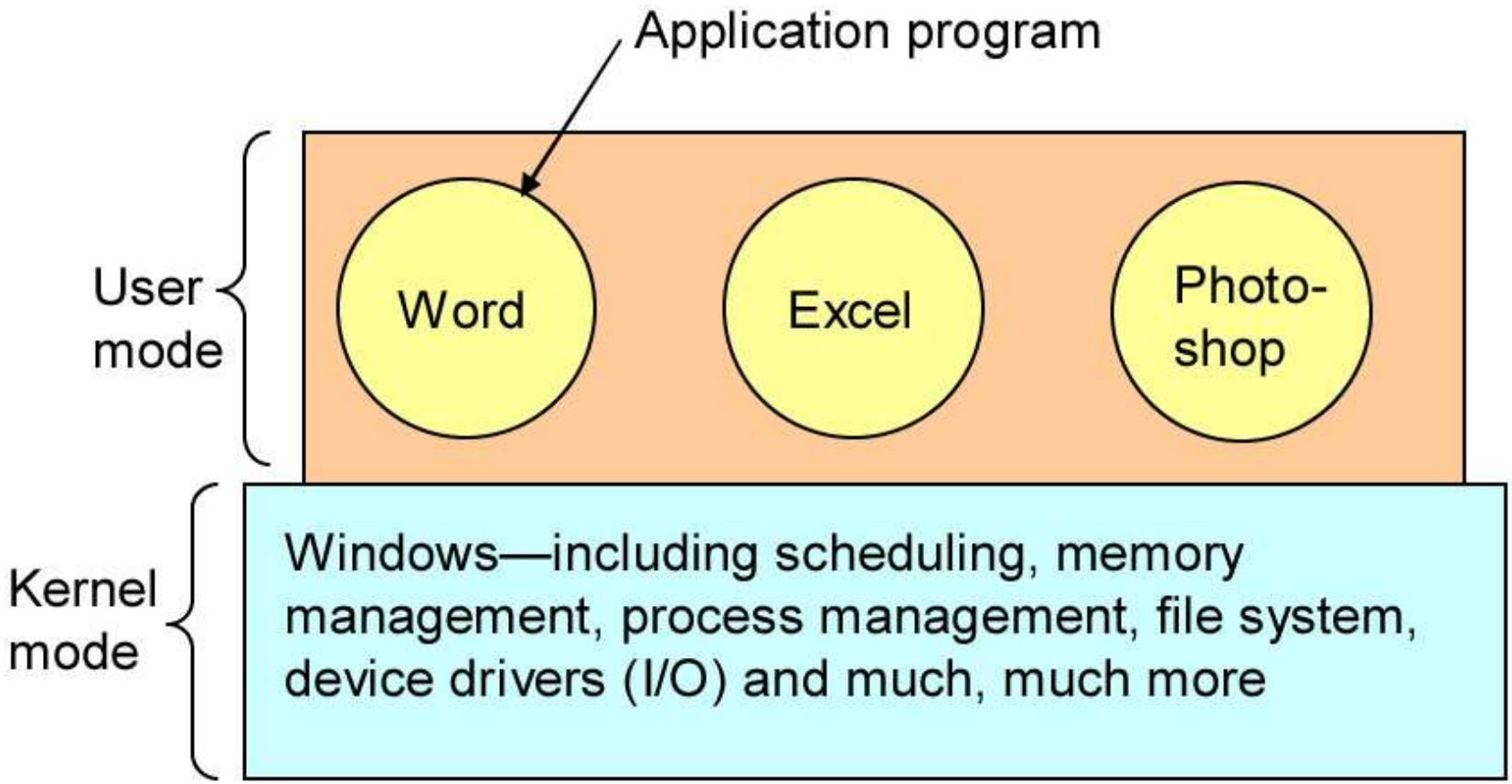
- This results in:
  - better reliability (isolation between components, less code running at full privilege)
  - ease of extension and customization
  - poor performance (user/kernel boundary crossings)

- First microkernel system was Hydra (CMU, 1970)
  - Follow-ons: Mach (CMU), Chorus (French UNIX-like OS), OS X (Apple)

Microkernel structure illustrated

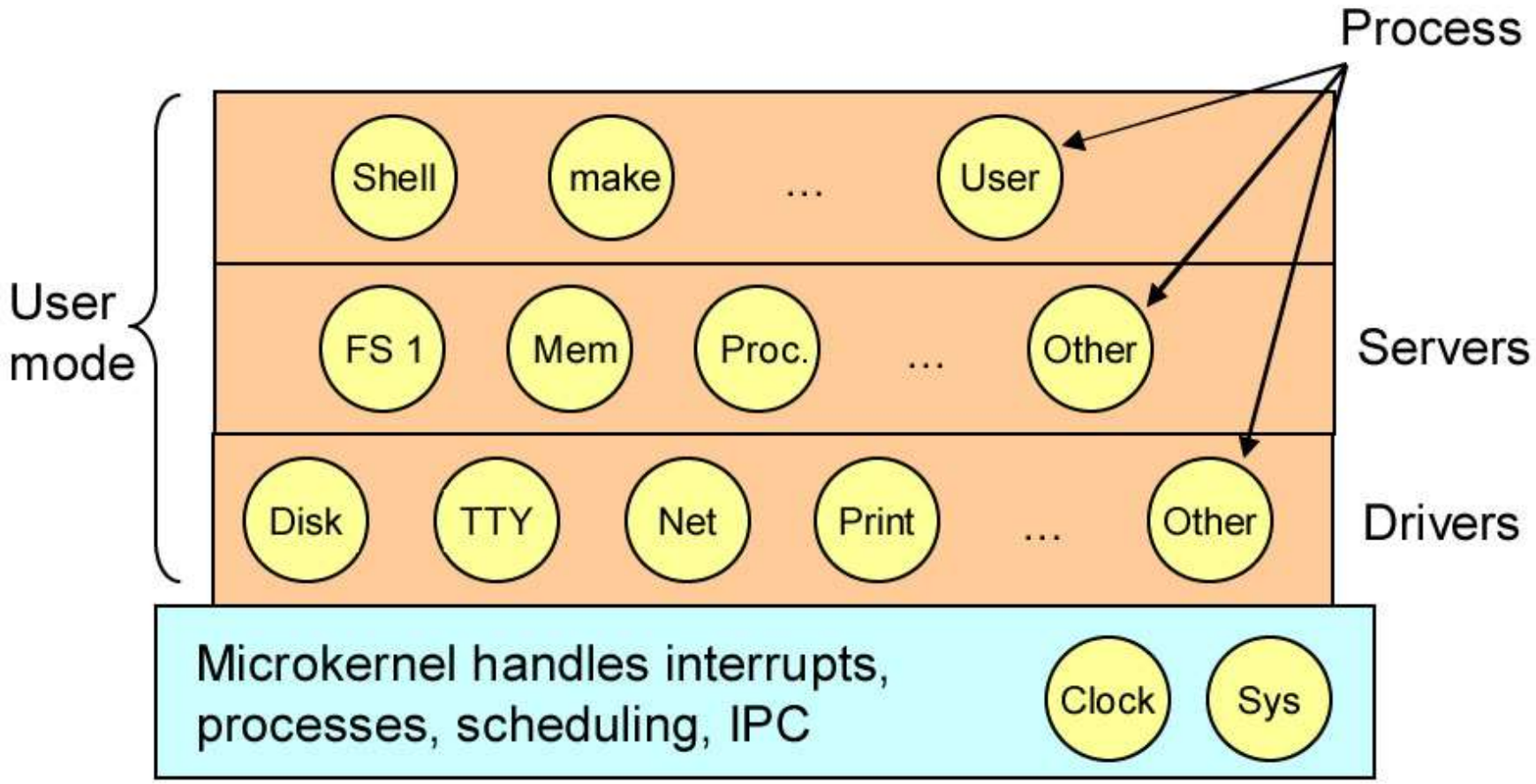


EXAMPLE: WINDOWS



From Andy Tanenbaum

ARCHITECTURE OF MINIX 3

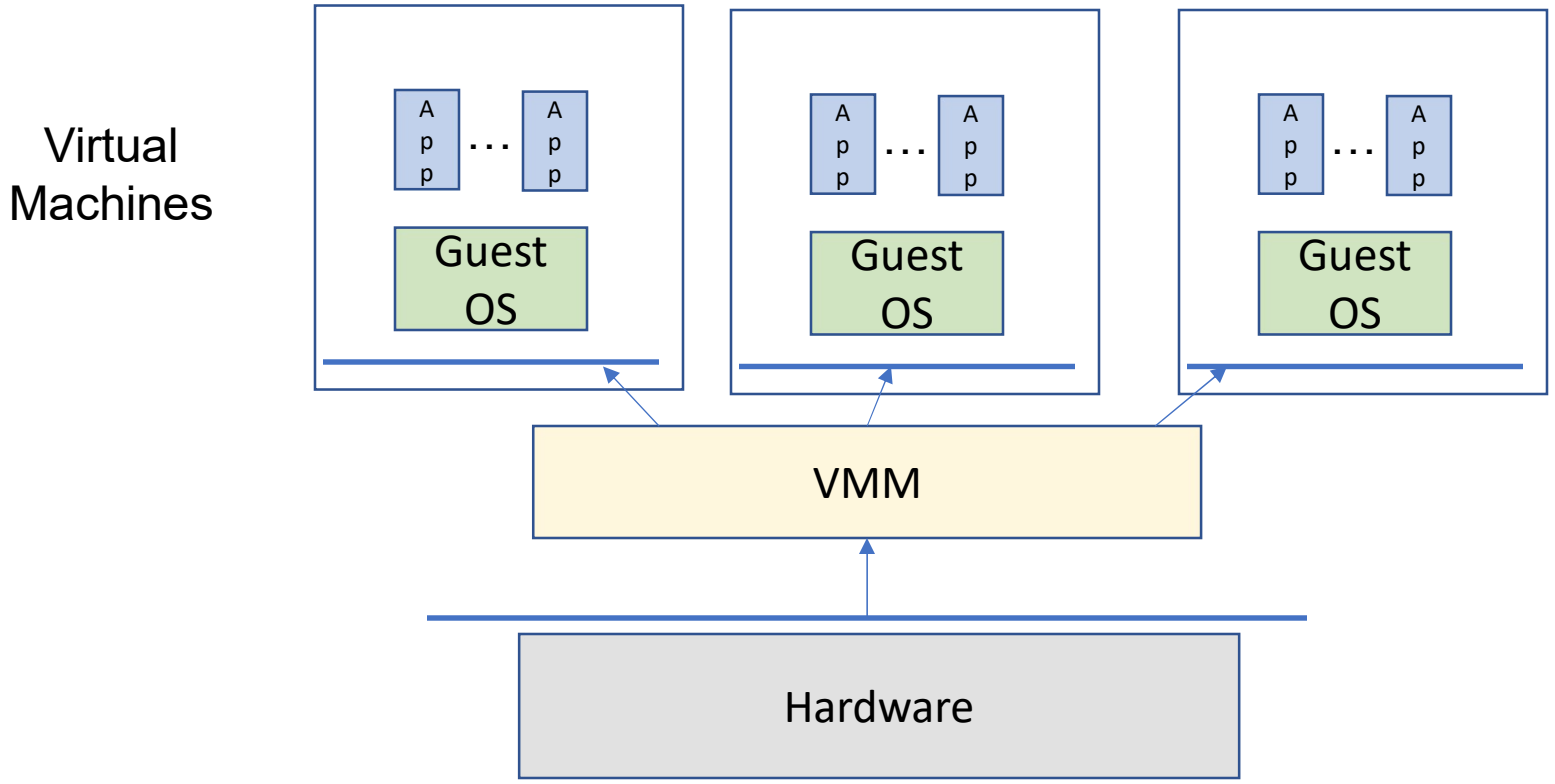


IPC == Inter-Process Communication

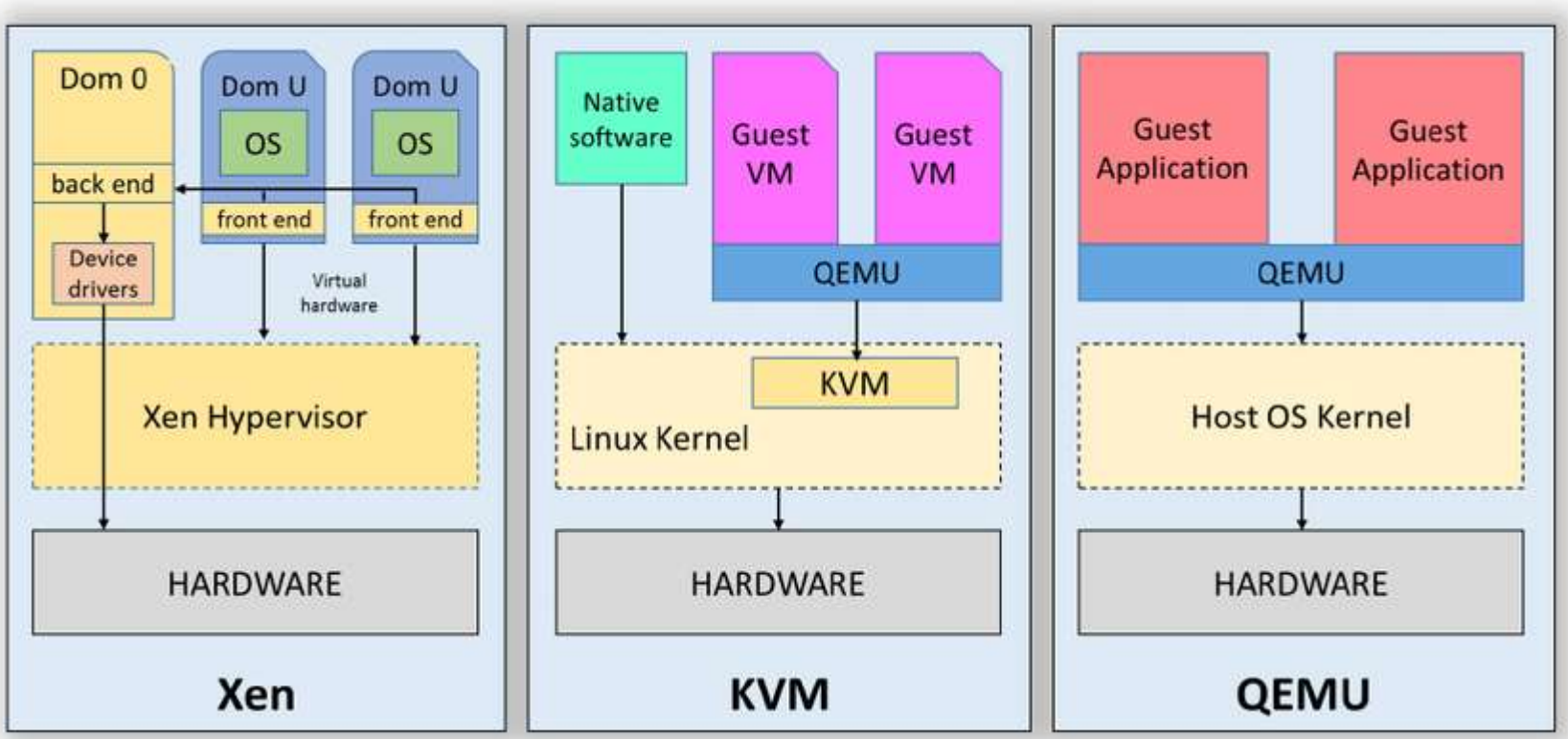
6

From Andy Tanenbaum

Another Kind of "Layering": Virtual Machine Monitors



Modern Virtual Machine Monitors



Exokernel

- Basic idea is for the kernel to present an abstraction of the hardware to user level
  - That abstraction doesn't have to have the same API as the actual hardware
- User-level processes operate on hardware via the abstraction/exokernel
- The exokernel validates that the operations requested are legal
- The exokernel's abstractions guarantee that user level code can operate only on the portions of actual physical resources they've been allocated
- Result?
  - Very cheap communication between user code and "OS code" as most of the OS is running at user level

Lecture Question Answers

- What are some important components of an OS?
  - process/thread management; memory/address space management; device IO; file systems; syscall interface / shells / windowing system; networking;
- “Program” vs. “Process” vs. “Thread”
  - Program is source code; process is running code; thread is an independent sequence of instruction execution within a process
- What are the (execution) states of a process?
  - Running, runnable, and blocked
  - What does it mean to be “blocked”?
    - That not only are you not running, you’re not eligible to run until something more happens
- What is the relationship between processes and address spaces?
  - There is usually a 1-1 relationship between processes and address spaces
- What’s special about device drivers?
  - They’re written by third parties, but execute in privileged mode

Lecture Question Answers

- What's the difference between a disk and a file system?
  - A disk is an array of blocks; a file system abstracts that to directories, files, and the like
- What is the "scope" of a namespace and why does it matter?
  - The region over which a single text string name will refer to the same object
  - When my code communicates with some other code, we need a way to name objects. The scope of a name limits with who I can communicate using it.
- If there were no adversaries would I need the OS to provide protection?
  - Yes, because my code has bugs and I want to protect myself from myself
- What does "OS structure" mean and why do we care?
  - It's just the universal issue of how do I engineer a large body of code so that I can get it right and maintain with minimal effort, while at the same time getting good performance from it. Because the OS is used by every application, it's correctness and performance permeate everything.

Lecture Question Answers

- What alternatives to monolithic have been tried?
  - Layered; microkernel; variations on systems that export hardware interface or otherwise achieve similar goals
- Why would you build a software layer that presents an interface identical to a hardware layer?
  - You might want to provide virtual machines in this way
- What is the real goal, and can you get it some other way?
  - You might claim the real goal is assured isolation and containment, and you might guess that there would be other approaches that might be cheaper than exporting the hardware interface (but that we won't see until later in the course)

Summary and Next Module

- Summary
  - OS design has been an evolutionary process of trial and error
  - Successful OS designs have run the spectrum from monolithic, to layered, to micro kernels, to virtual machine monitors
  - The role and design of an OS are still evolving
  - It is impossible to pick one “correct” way to structure an OS
    - (They’re all wrong!)