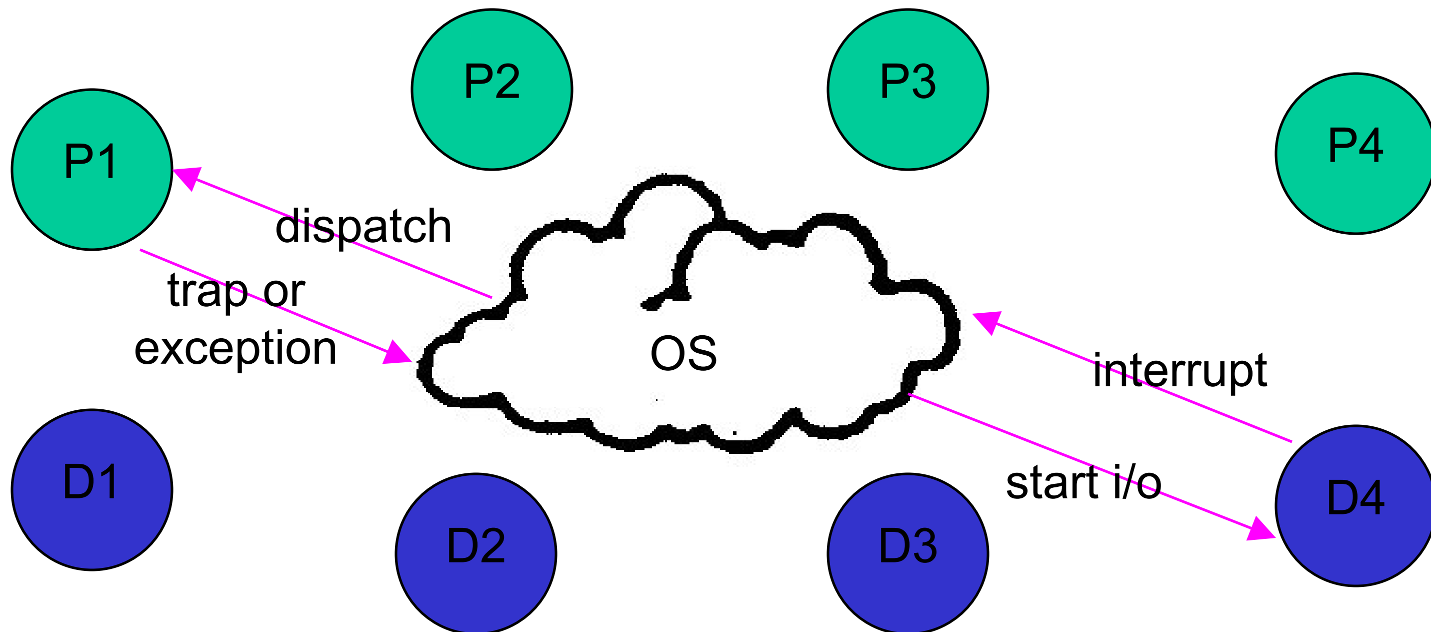
CSE 451: Operating Systems
Spring 2017

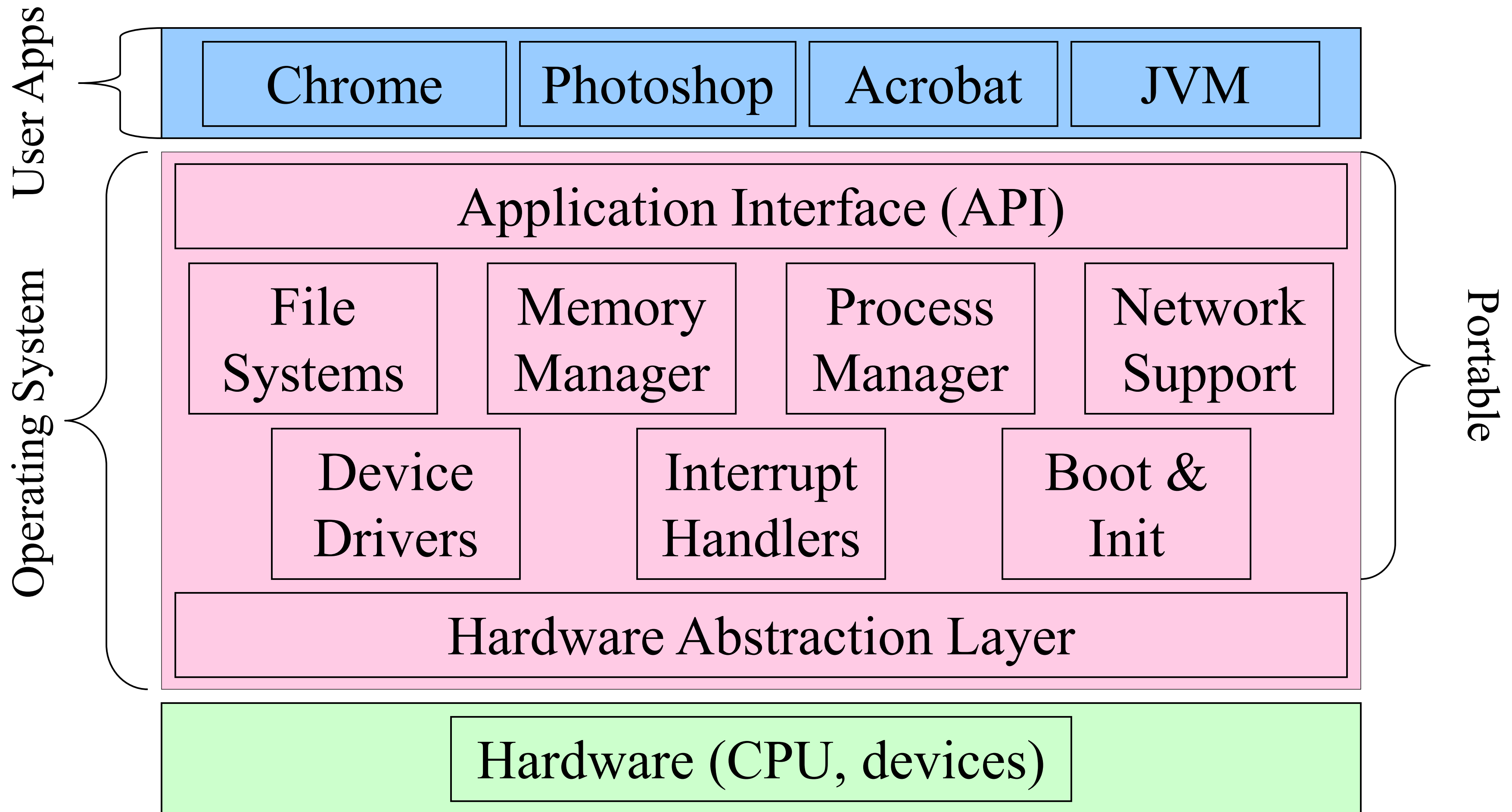
Module 3
Operating System
Components and Structure

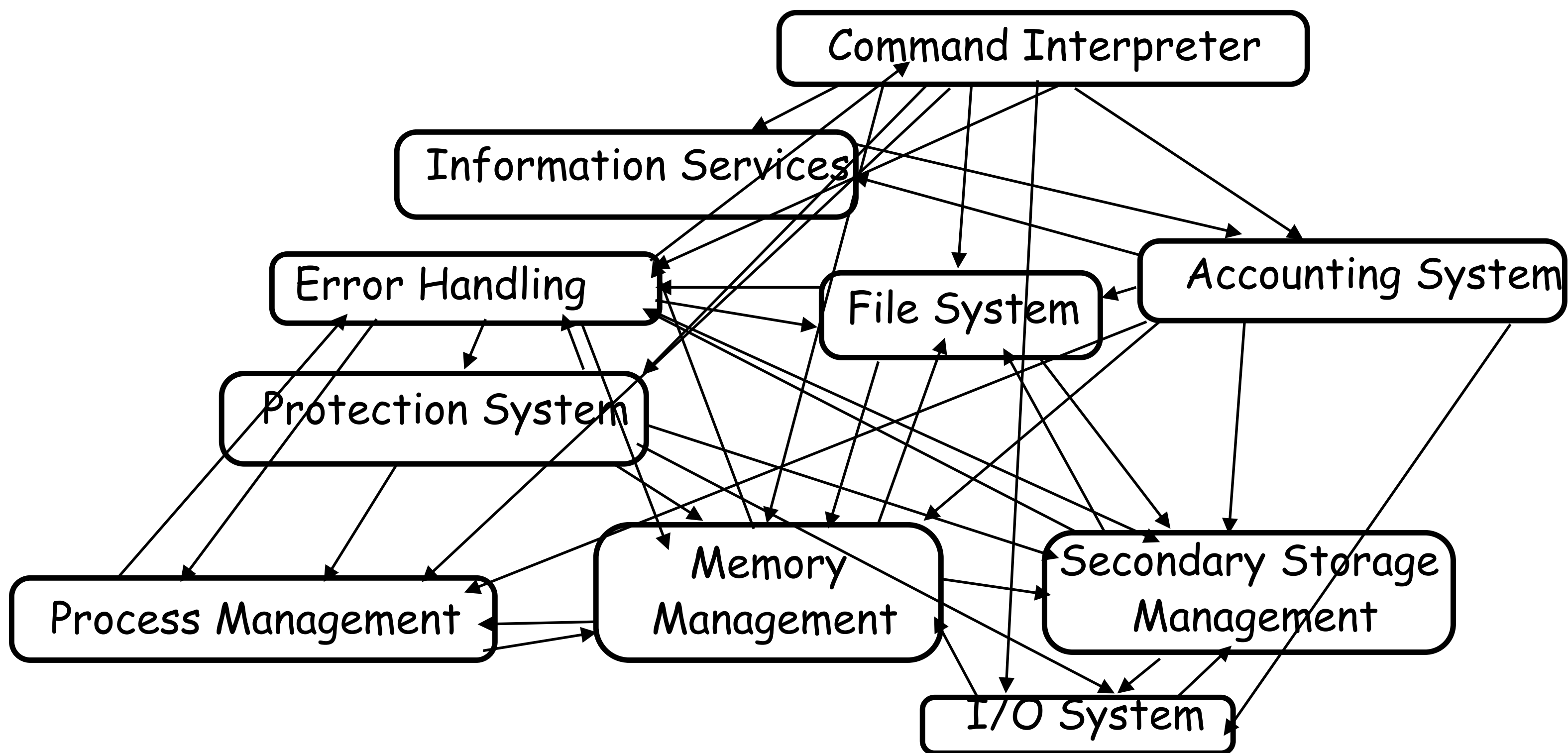
John Zahorjan

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




User Apps
Chrome
Photoshop
Acrobat
JVM
Operating System
Application Interface (API)
File Systems
Memory Manager
Process Manager
Network Support
Device Drivers
Interrupt Handlers
Boot & Init
Hardware Abstraction Layer
Portable
Hardware (CPU, devices)


Command Interpreter
Information Services
Error Handling
Accounting System
File System
Protection System
Memory Management
Secondary Storage Management
Process Management
I/O System

Part I: Major OS components

1. processes/threads
2. memory
3. I/O
4. secondary storage
5. file systems
6. protection
7. shells (command interpreter, or OS UI)
8. windowing system
9. networking

1. Process management

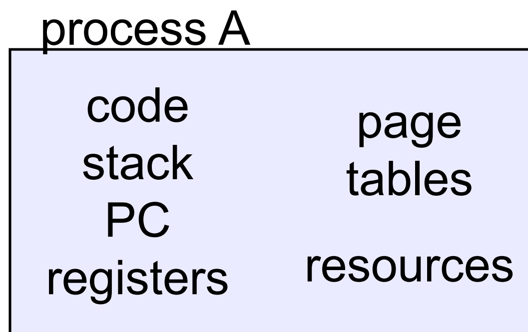
- An OS executes many kinds of activities:
  - users' programs
  - background jobs or scripts
  - system programs
    - print managers, name servers, file servers, network daemons, …
- Each of these activities is encapsulated in a process
  - a process is a running program
  - a process has an execution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

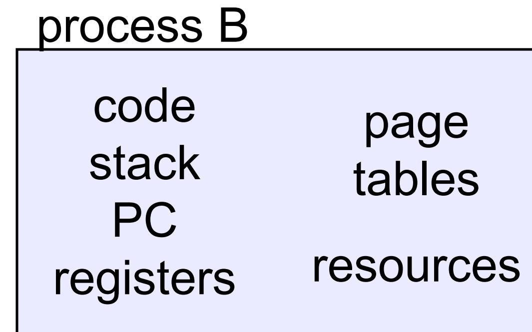
Processes vs. Threads

- 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.

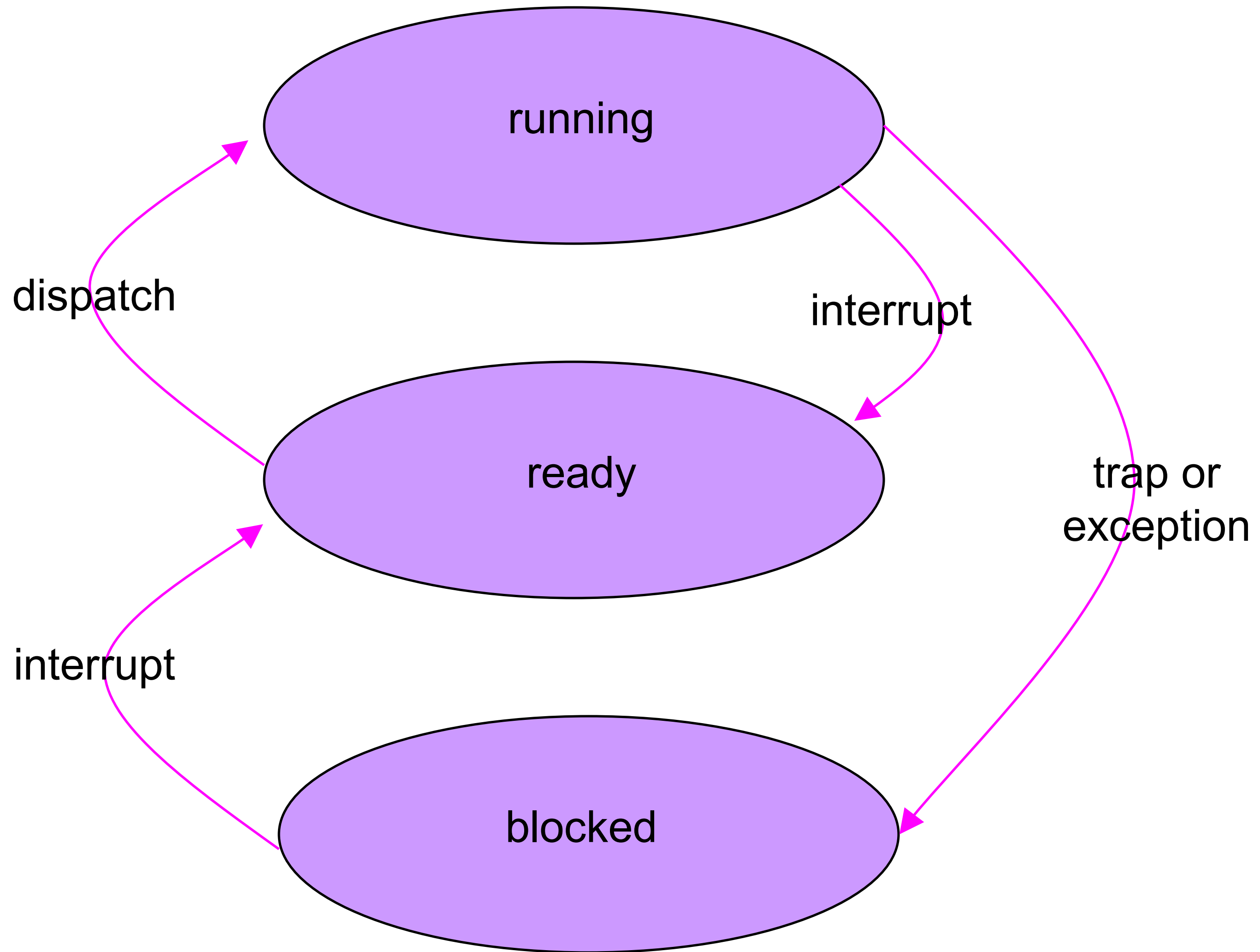
Program / processor / process

- 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



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)
  - inter-process communication (kill, pipe, mmap, …)
  - inter-process synchronization (wait, flock, sem_open, ...)

2. Memory management

- Primary memory is the directly accessed storage for the CPU
  - programs must be resident in memory to execute
  - memory access is fast
  - but memory doesn't survive power failures

- OS must:
  - allocate memory space for processes
  - deallocate space when needed by rest of system
  - maintain mappings from virtual memory to physical
    - page tables

    } Mechanism

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

    } Policy

3. I/O

- A big chunk of the OS kernel deals with I/O
  - hundreds of thousands of lines in Windows, Unix, etc.
- 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: SCSI 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

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, …)
  - handle scheduling of disk operations, error handling, and often management of space on disks
- Usually independent of file system
  - device => raw storage
  - file system => layer of abstraction providing structured storage

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, …
  - user code operates on files/directories, not on disk blocks
- FS defines operations on objects, like creat, read, write, stat

“File system”

- The term “file system” has at least three common meanings
  - The generic notion of providing a more convenient abtraction layered on some storage device
  - A particular software implementation of that generic idea, e.g., NTFS or FAT or ext4
  - A self-contained, and so physically portable, bunch of bits on some storage device
    - 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

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 mechanisms motivations:
  - "I'm not perfect" -- help to detect and contain unintentional errors
  - "There are adversaries" -- preventing malicious abuses

7. Command Interpreter (shell)

- 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, or from the mouse (GUIs)
  - allows users to launch and control new programs
- On some systems, command interpreter may be a standard part of the OS (e.g., MS DOS, Apple II, JOS)
- On others, it's just non-privileged code that provides an interface to the user
  - e.g., bash/csh/tcsh/zsh on UNIX

© 2017 Gribble, Lazowska, Levy, Zahorjan, Zbikowski

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

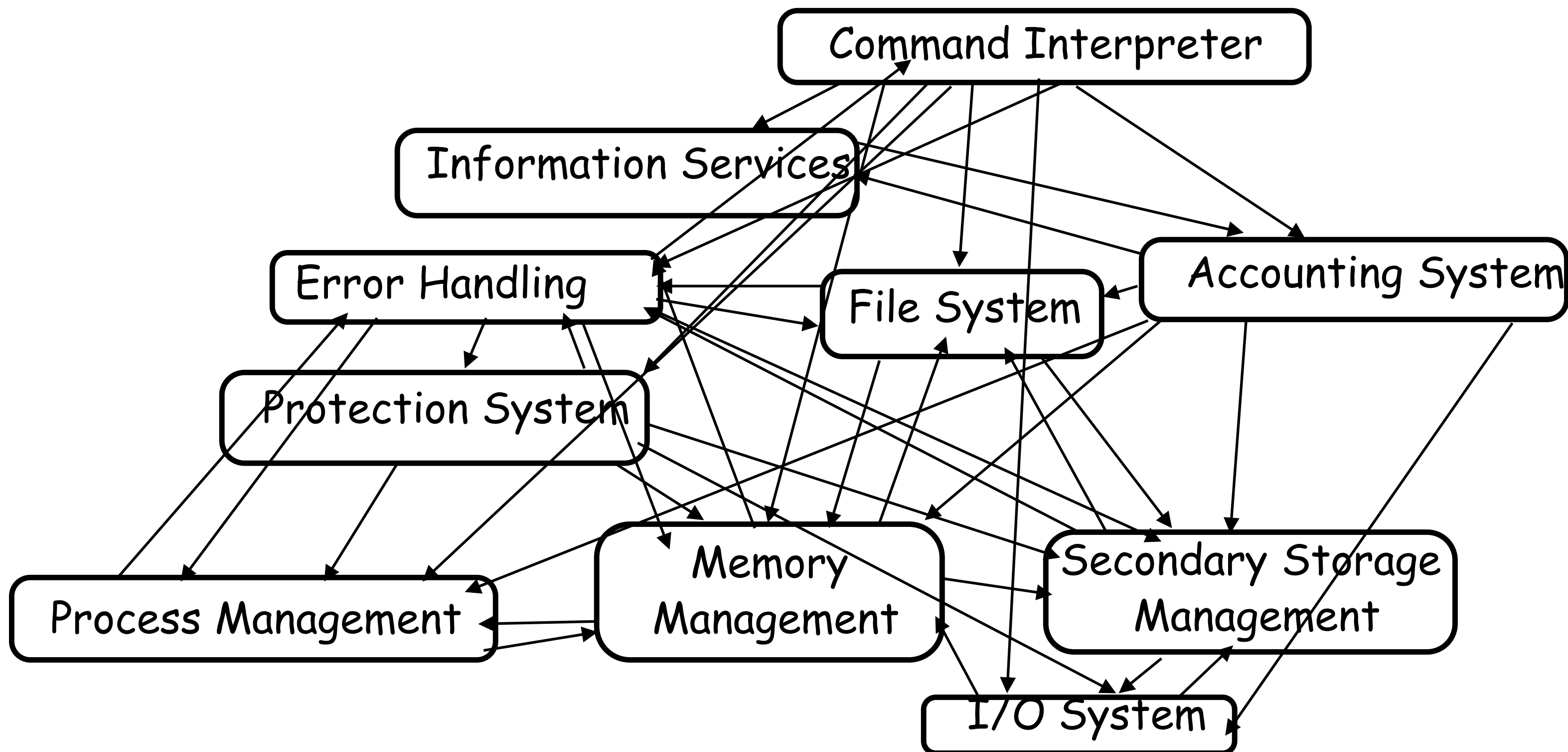
- The Internet moves bits from one machine to another machine
  - An IP address basically names a machine
    - e.g., 128.208.1.137 is attu1.cs.washington.edu
- When bytes are sent “to a machine,” who receives them?
  - The operating system
- But I want to ask the web server process on the machine for 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)?

Part II: OS structure

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

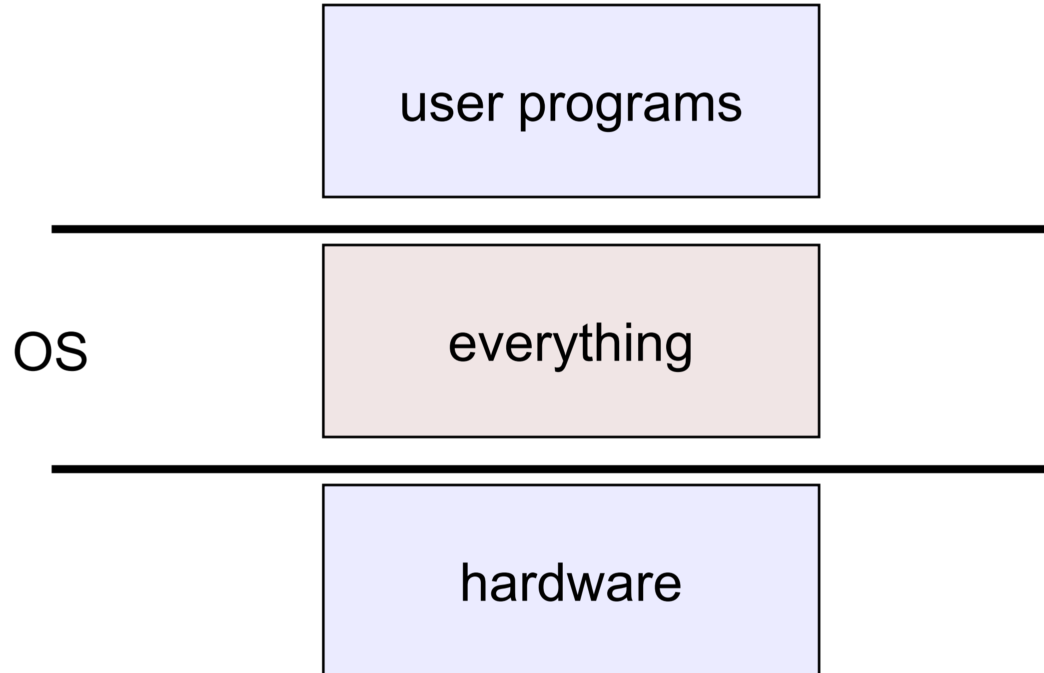


OS structure

- 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, …
- 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:
  - cost of module interactions is low (procedure call)
- Disadvantages:
  - hard to understand
  - hard to modify
  - unreliable (no isolation between system modules)
  - hard to maintain
- What is the alternative?
  - find a way to organize the OS in order to simplify its design and implementation

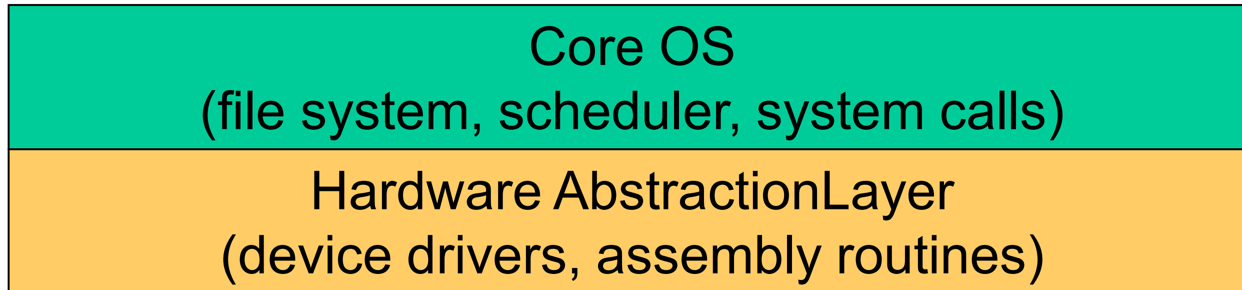
Layering

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

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
  - strict layering isn't flexible enough
- Poor performance
  - each layer crossing has overhead associated with it
- Disjunction between model and reality
  - systems modeled as layers, but not really built that way

Hardware Abstraction Layer

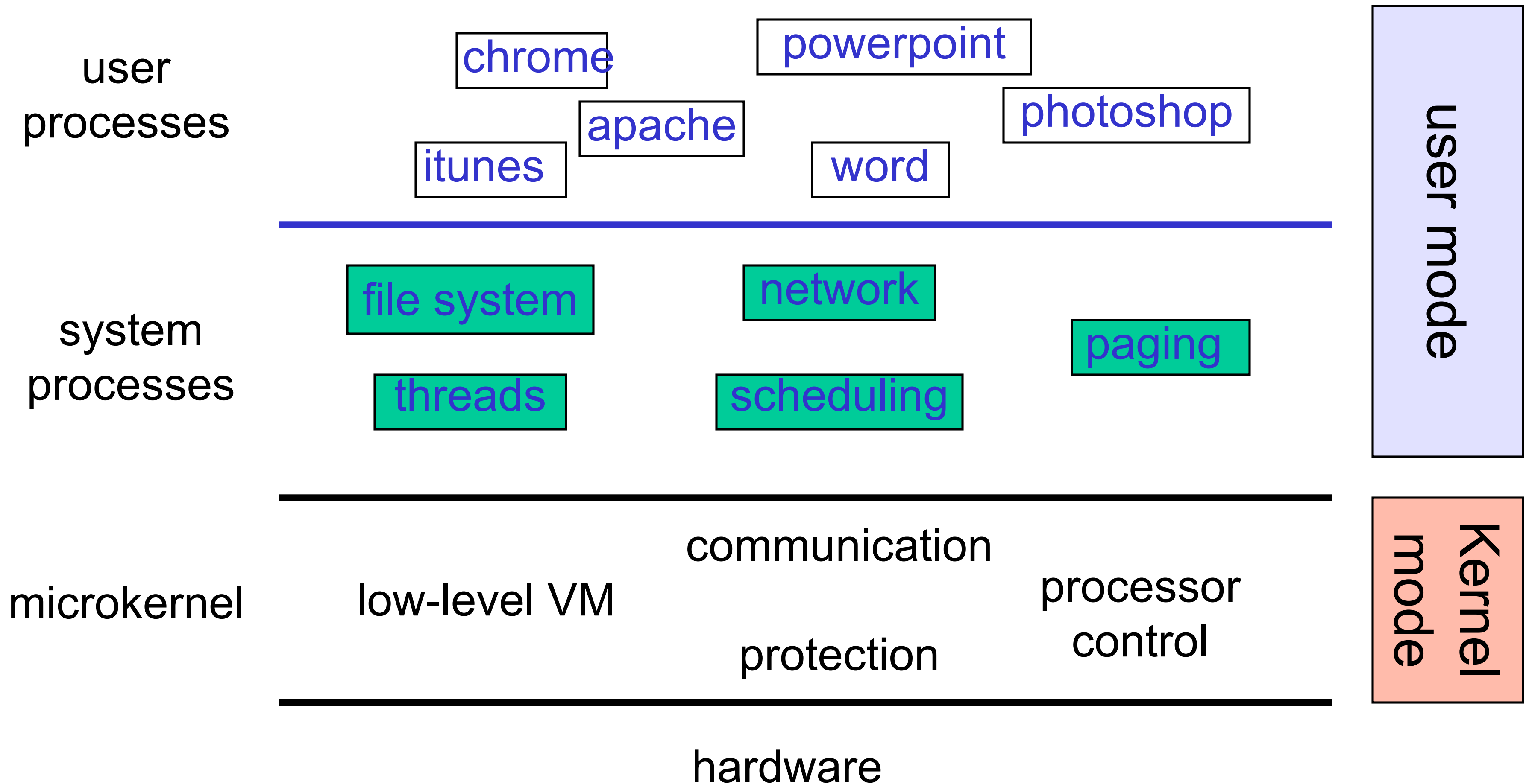


- An example of layering in modern operating systems
- Goal: separates hardware-specific routines from the "core" OS
  - Provides portability
  - Improves readability

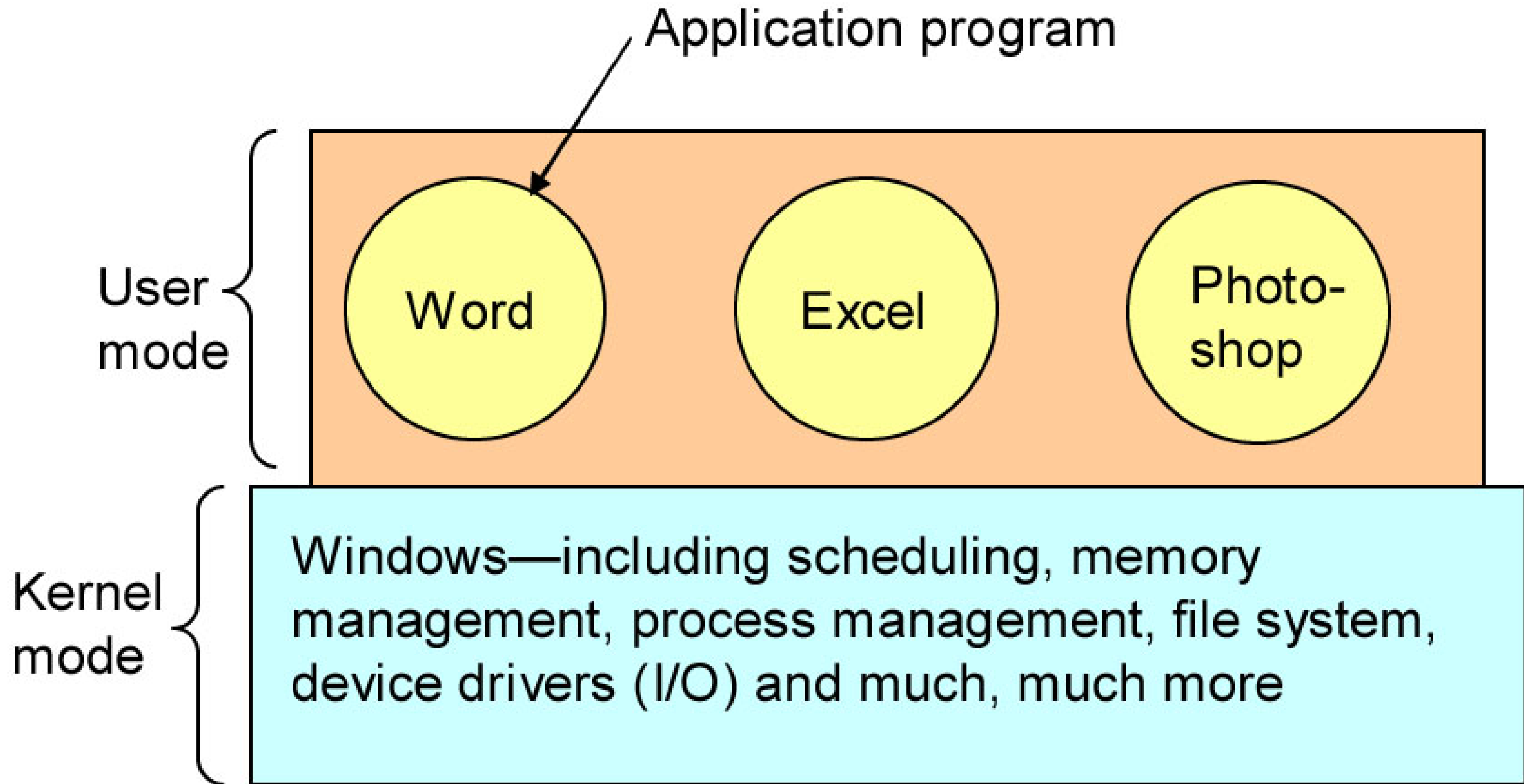
Alternative to Monolithic: Microkernels

- Introduced in the late 80’s, early 90’s
- Goal:
  - minimize what goes into the kernel
  - organize rest of 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

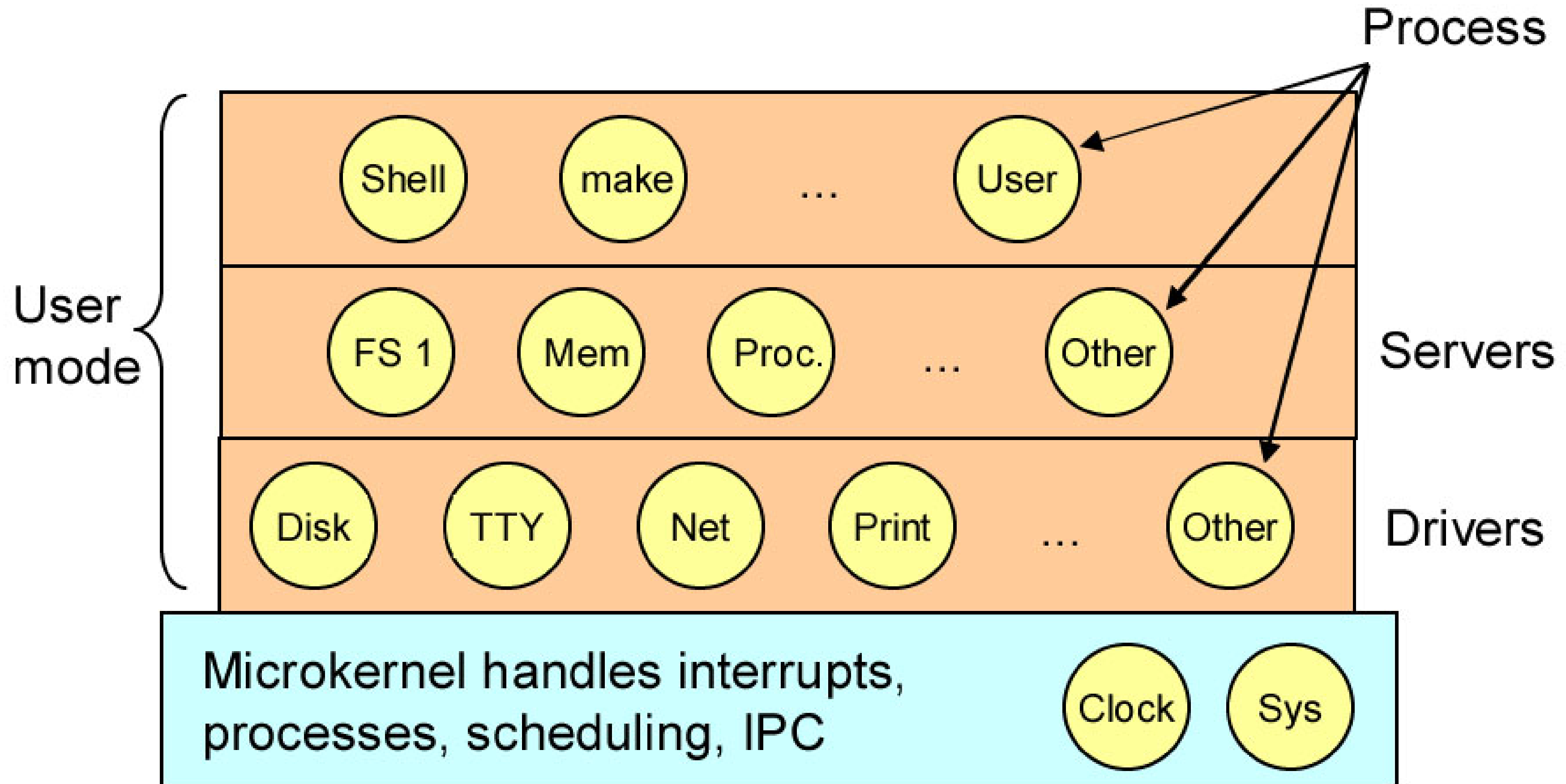


EXAMPLE: WINDOWS



From Andy Tanenbaum

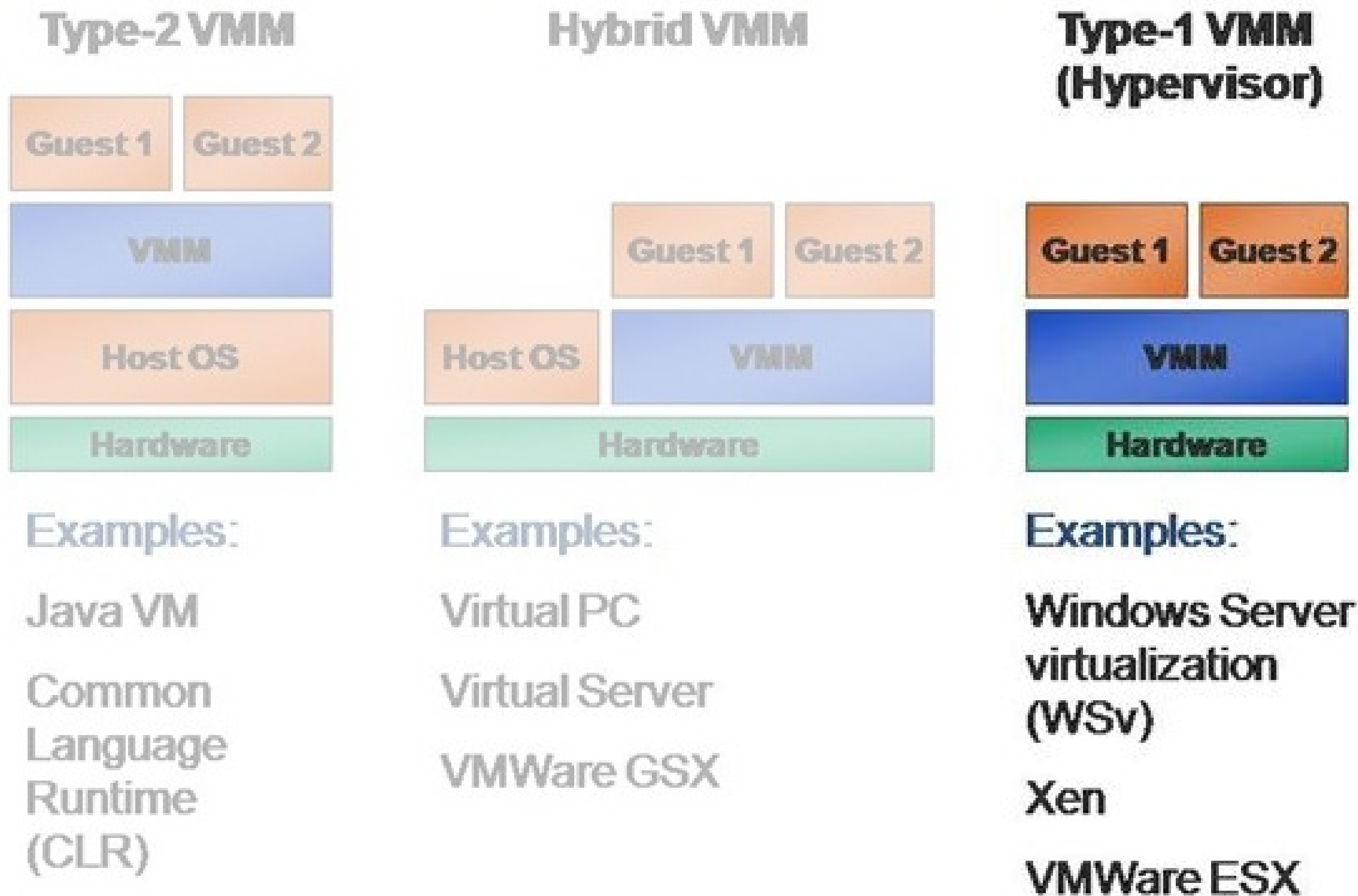
ARCHITECTURE OF MINIX 3



6

From Andy Tanenbaum

Virtual Machine Monitors



- Transparently implement "hardware" in software
- Voilà, you can boot a "guest OS"

From http://port25.technet.com/

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

Summary and Next Module

- Summary
  - OS design has been a evolutionary process of trial and error. Probably more error than success
  - 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