Operating Systems

Operating System Structure

Lecture 2

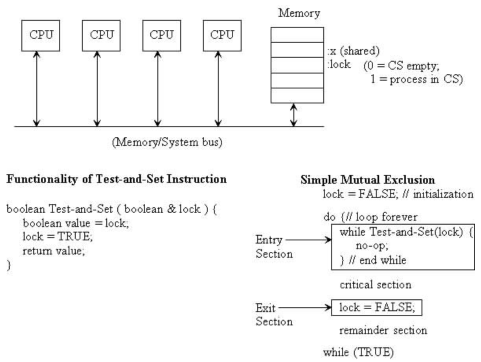
Michael O'Boyle

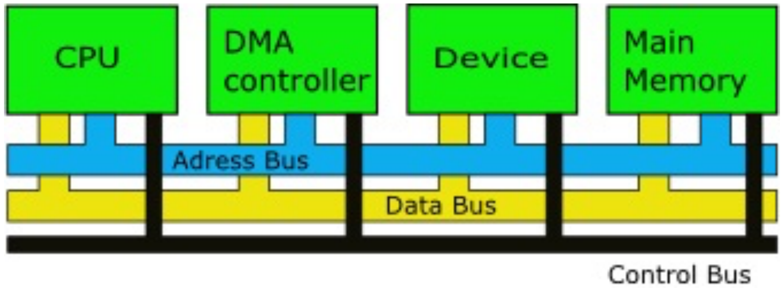
Overview

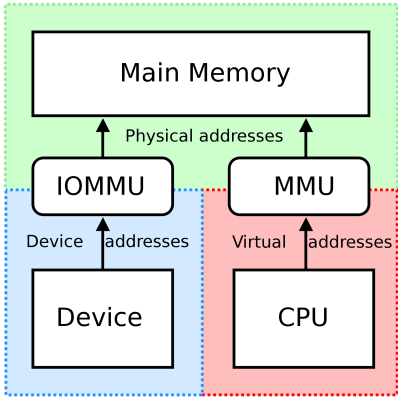
- Architecture impact
- User operating interaction
  - User vs kernel
  - Syscall
- Operating System structure
  - Layers
  - Examples

Lower-level architecture affects (is affected by) the OS

- The operating system supports sharing and protection
  - multiple applications can run concurrently, sharing resources
  - a buggy or malicious application cannot disrupt other applications or the system
- There are many approaches to achieving this
- The architecture determines which approaches are viable (reasonably efficient, or even possible)
  - includes instruction set (synchronization, I/O, …)
  - also hardware components like MMU or DMA controllers



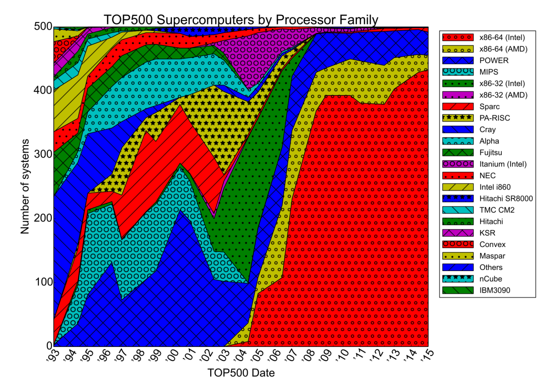




Architecture support for the OS

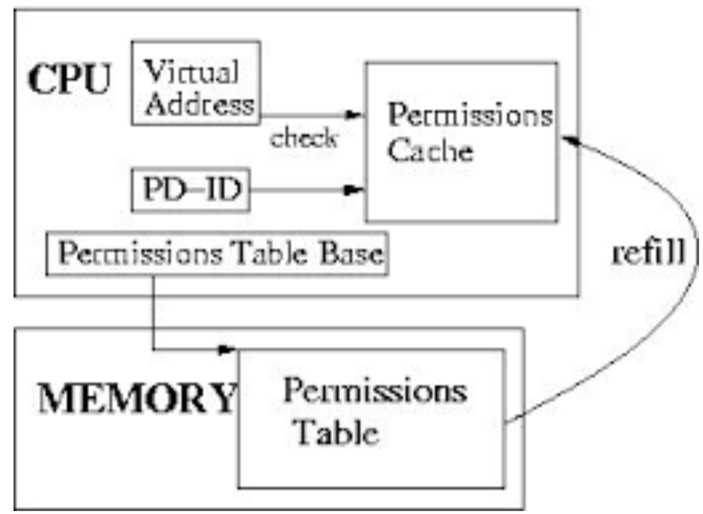
- Architectural support can simplify OS tasks
  - e.g.: early PC operating systems (DOS, MacOS) lacked support for virtual memory, in part because at that time PCs lacked necessary hardware support

- Until recently, Intel-based PCs still lacked support for 64-bit addressing
  - has been available for a decade on other platforms: MIPS, Alpha, IBM, etc…
  - Changed driven by AMD's 64-bit architecture



Architectural features affecting OS's

- These features were built primarily to support OS's:
  - timer (clock) operation
  - synchronization instructions
    - e.g., atomic test-and-set
  - memory protection
  - I/O control operations
  - interrupts and exceptions
  - protected modes of execution
    - kernel vs. user mode
  - privileged instructions
  - system calls
    - Including software interrupts
  - virtualization architectures
- ASPLOS



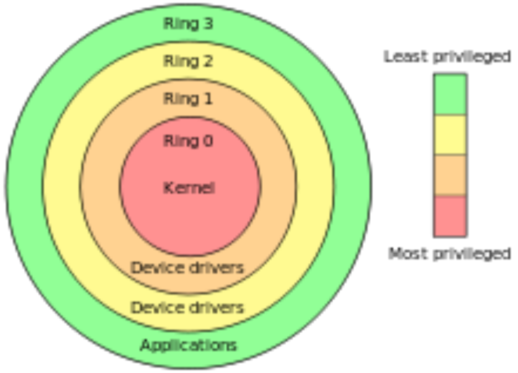
Privileged instructions

- Some instructions are restricted to the OS
  - known as privileged instructions
- Only the OS can:
  - directly access I/O devices (disks, network cards)
  - manipulate memory state management
    - page table pointers, TLB loads, etc.
  - manipulate special 'mode bits'
    - interrupt priority level

- Restrictions provide safety and security

OS protection

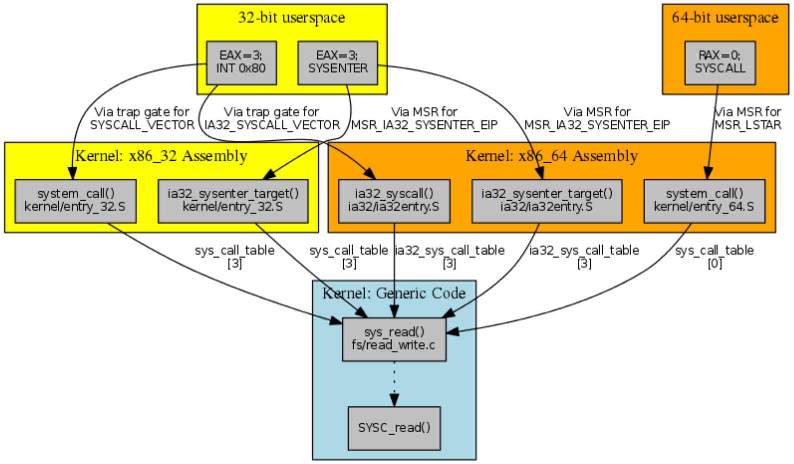
- So how does the processor know if a privileged instruction should be executed?
  - the architecture must support at least two modes of operation: kernel mode and user mode
    - x86 support 4 protection modes



  - mode is set by status bit in a protected processor register
    - user programs execute in user mode
    - OS executes in kernel (privileged) mode (OS == kernel)
- Privileged instructions can only be executed in kernel (privileged) mode
  - if code running in user mode attempts to execute a privileged instruction the Illegal excecutin trap

Crossing protection boundaries

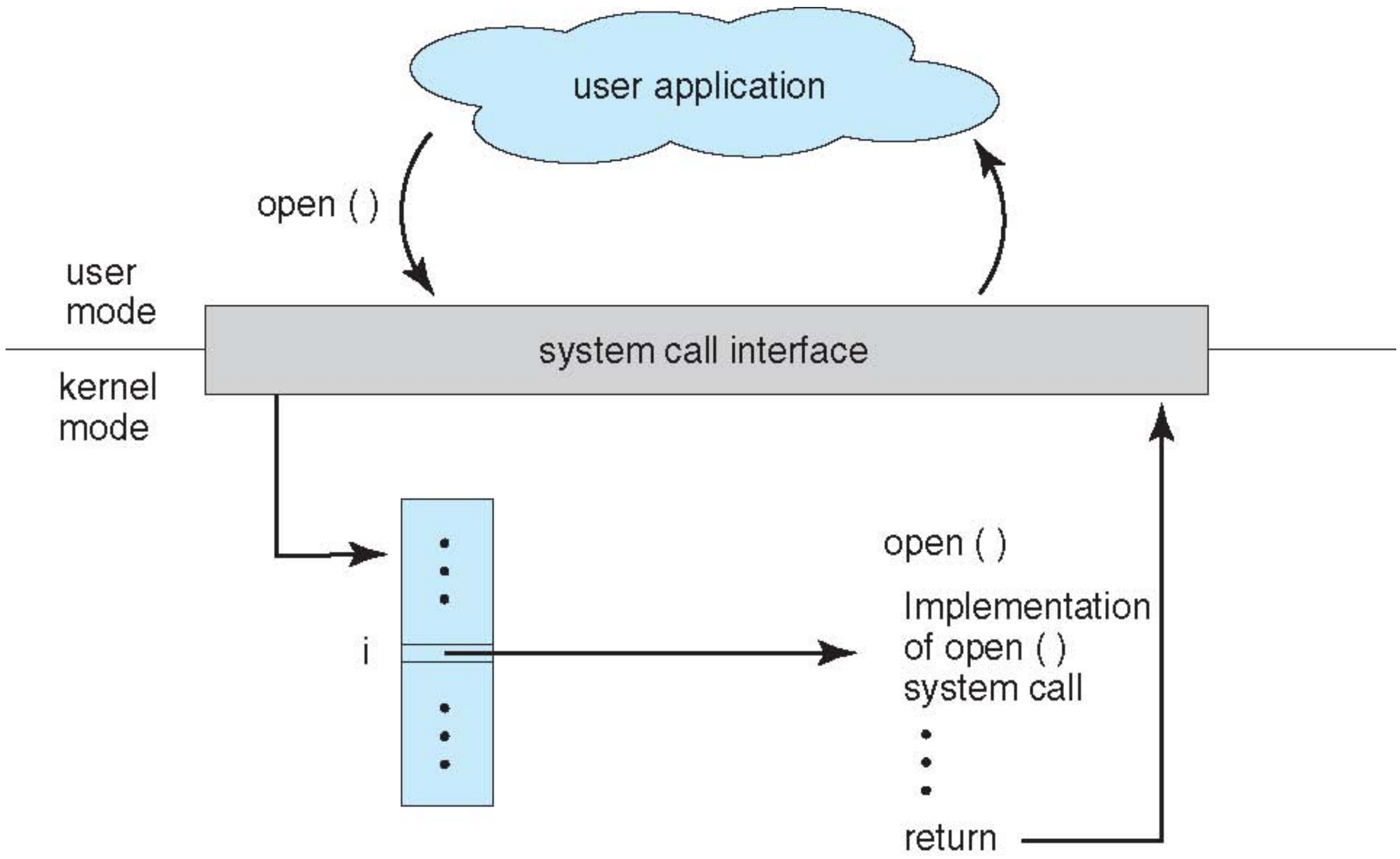
- So how do user programs do something privileged?
  - e.g., how can you write to a disk if you can't execute an I/O instructions?
- User programs must call an OS procedure – that is ask the OS to do it for them
  - OS defines a set of system calls
  - User-mode program executes system call instruction
- Syscall instruction
  - Like a protected procedure call



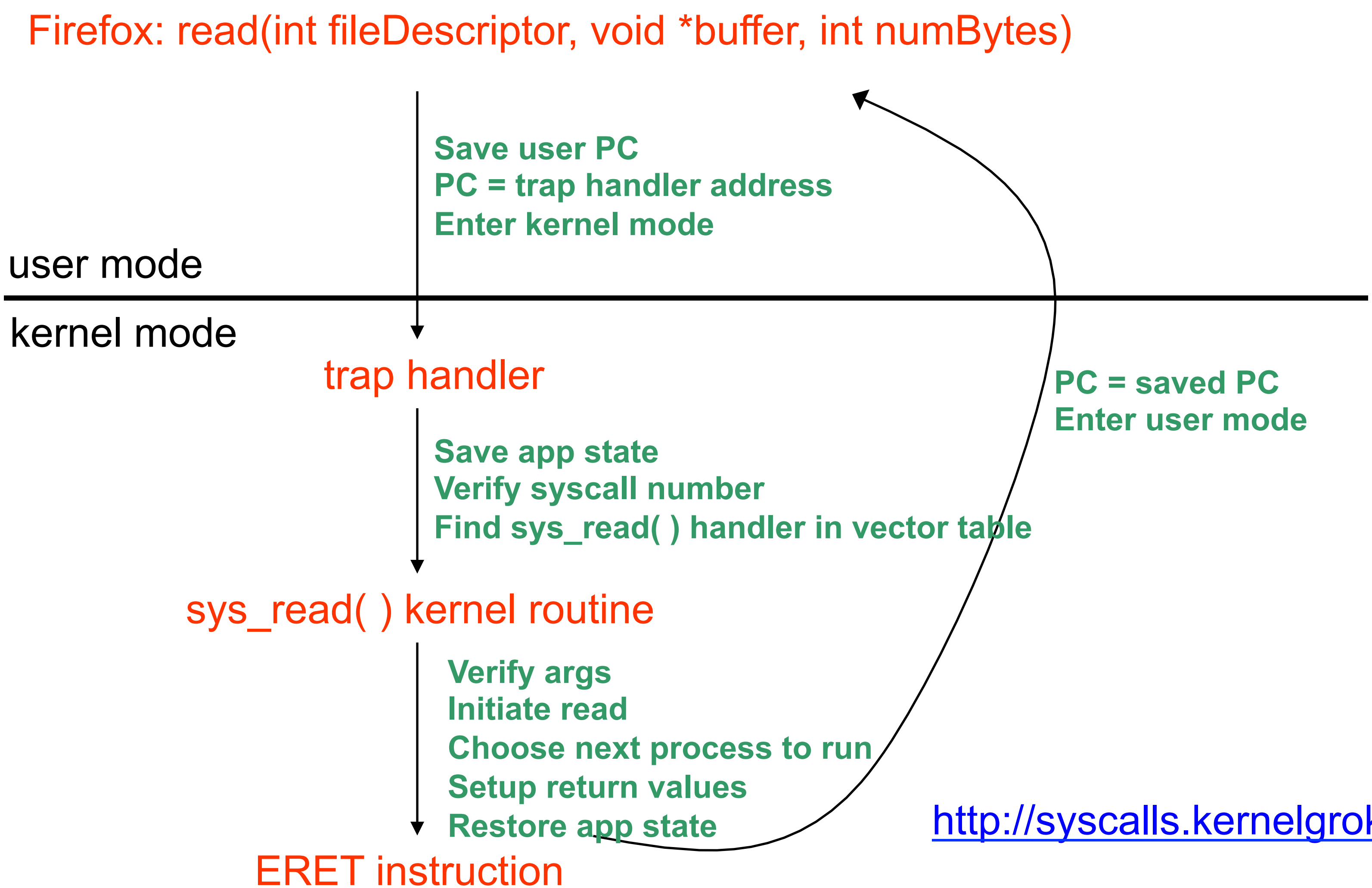
Syscall

- The syscall instruction atomically:
  - Saves the current PC
  - Sets the execution mode to privileged
  - Sets the PC to a handler address
- Similar to a procedure call
  - Caller puts arguments in a place callee expects (registers or stack)
    - One of the args is a syscall number, indicating which OS function to invoke
  - Callee (OS) saves caller's state (registers, other control state) so it can use the CPU
  - OS function code runs
    - OS must verify caller's arguments (e.g., pointers)
  - OS returns using a special instruction
    - Automatically sets PC to return address and sets execution mode to user

API – System Call – OS Relationship



A kernel crossing illustrated



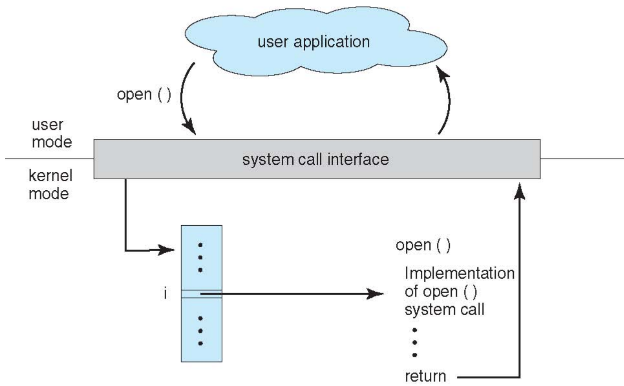
http://syscalls.kernelgrok.com/

Examples of Windows and Unix System Calls

	Windows	Unix
Process Control	CreateProcess() ExitProcess() WaitForSingleObject()	fork() exit() wait()
File Manipulation	CreateFile() ReadFile() WriteFile() CloseHandle()	open() read() write() close()
Device Manipulation	SetConsoleMode() ReadConsole() WriteConsole()	ioctl() read() write()
Information Maintenance	GetCurrentProcessID() SetTimer() Sleep()	getpid() alarm() sleep()
Communication	CreatePipe() CreateFileMapping() MapViewOfFile()	pipe() shmget() mmap()
Protection	SetFileSecurity() InitlializeSecurityDescriptor() SetSecurityDescriptorGroup()	chmod() umask() chown()

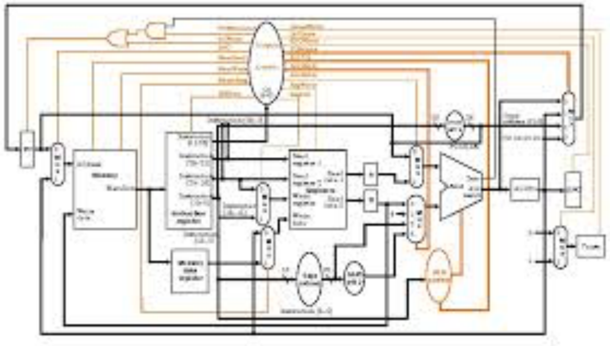
System call issues

- A syscall is not subroutine call, with the caller specifying the next PC.
  - the caller knows where the subroutines are located in memory; therefore they can be target of attack.
- The kernel saves state?
  - Prevents overwriting of values
- The kernel verify arguments
  - Prevents buggy code crashing system
- Referring to kernel objects as arguments
  - Data copied between user buffer and kernel buffer



Exception Handling and Protection

- *All* entries to the OS occur via the mechanism just shown
  - Acquiring privileged mode and branching to the trap handler are inseparable
- Terminology:
  - **Interrupt**: asynchronous; caused by an external device
  - **Exception**: synchronous; unexpected problem with instruction
  - **Trap**: synchronous; intended transition to OS due to an instruction
- Privileged instructions and resources are the basis for most everything: memory protection, protected I/O, limiting user resource consumption, …

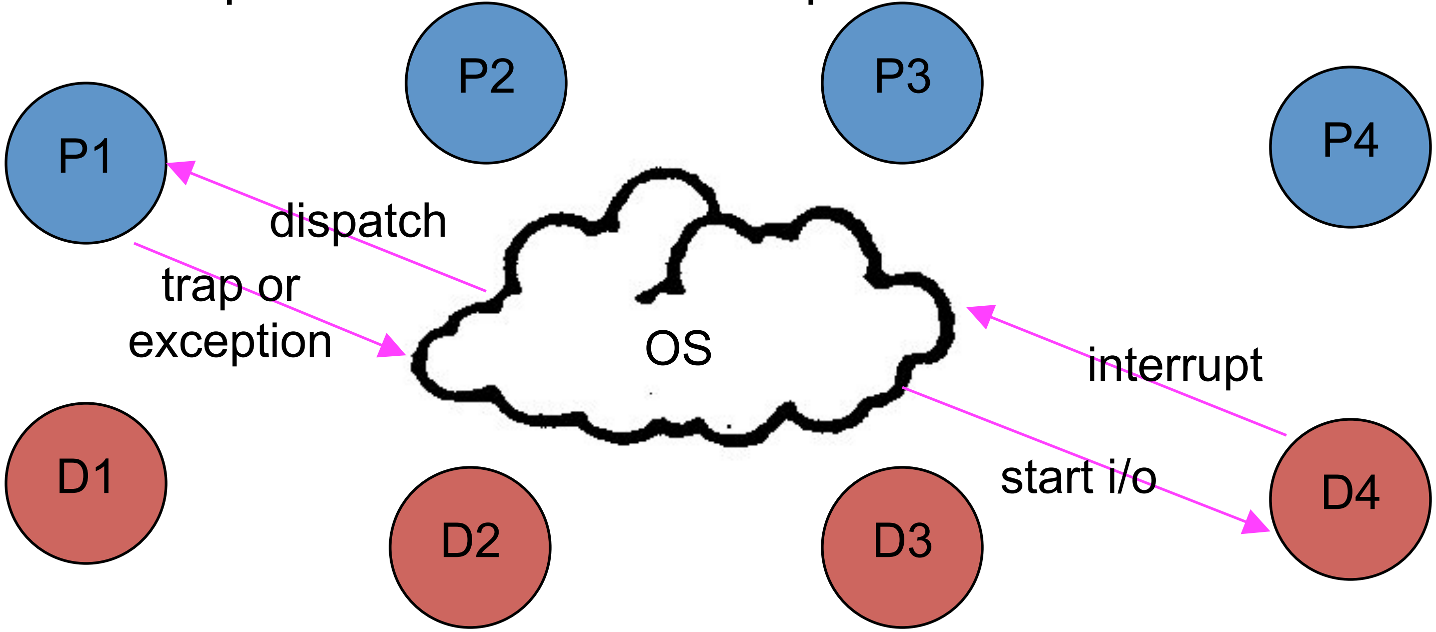


Overview

- *Architecture impact*
- *User operating interaction*
  - *User vs kernel*
  - *Syscall*
- **Operating System structure**
  - Layers
  - Examples

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



Operating System Design and Implementation

- Design and Implementation of OS not “solvable”, but some approaches have proven successful
- Internal structure of different Operating Systems can vary widely
- Start the design by defining goals and specifications
- Affected by choice of hardware, type of system
- **User** goals and **System** goals
  - User goals – operating system should be convenient to use, easy to learn, reliable, safe, and fast
  - System goals – operating system should be easy to design, implement, and maintain, as well as flexible, reliable, error-free, and efficient

Operating System Design and Implementation

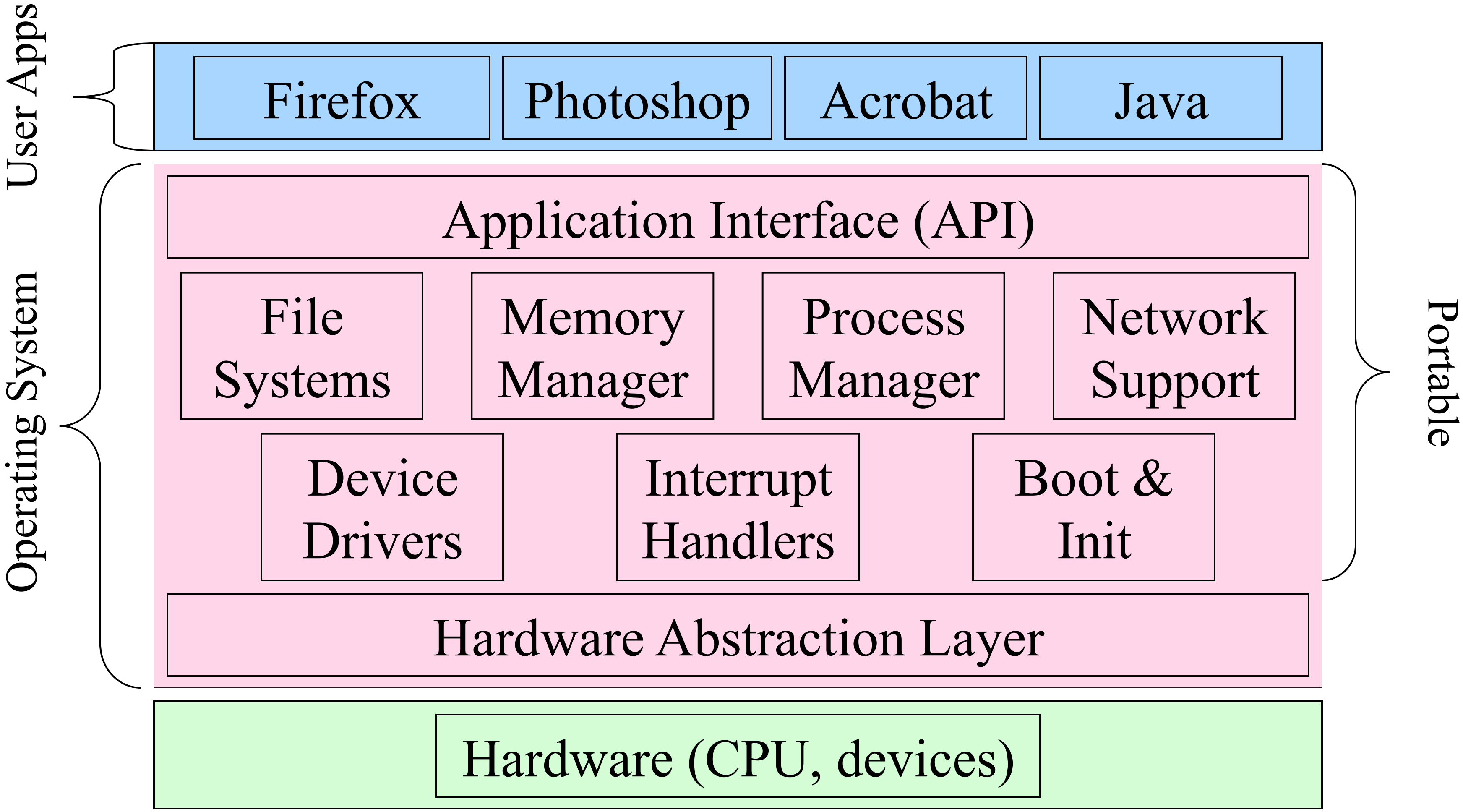
- Important principle to separate
  Policy: ***What*** will be done?
  Mechanism: ***How*** to do it?

- Mechanisms determine how to do something, policies decide what will be done

- The separation of policy from mechanism is a very important principle, it allows maximum flexibility if policy decisions are to be changed later (example – timer)

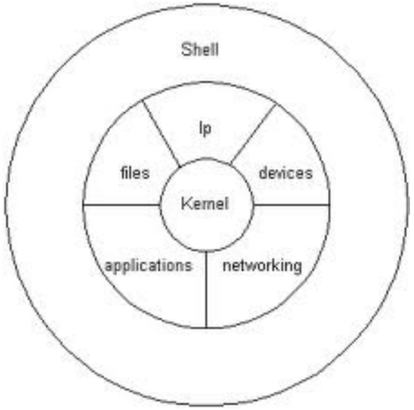
- Specifying and designing an OS is highly creative task of **software engineering**

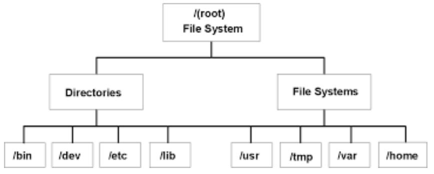
System layers



Major OS components

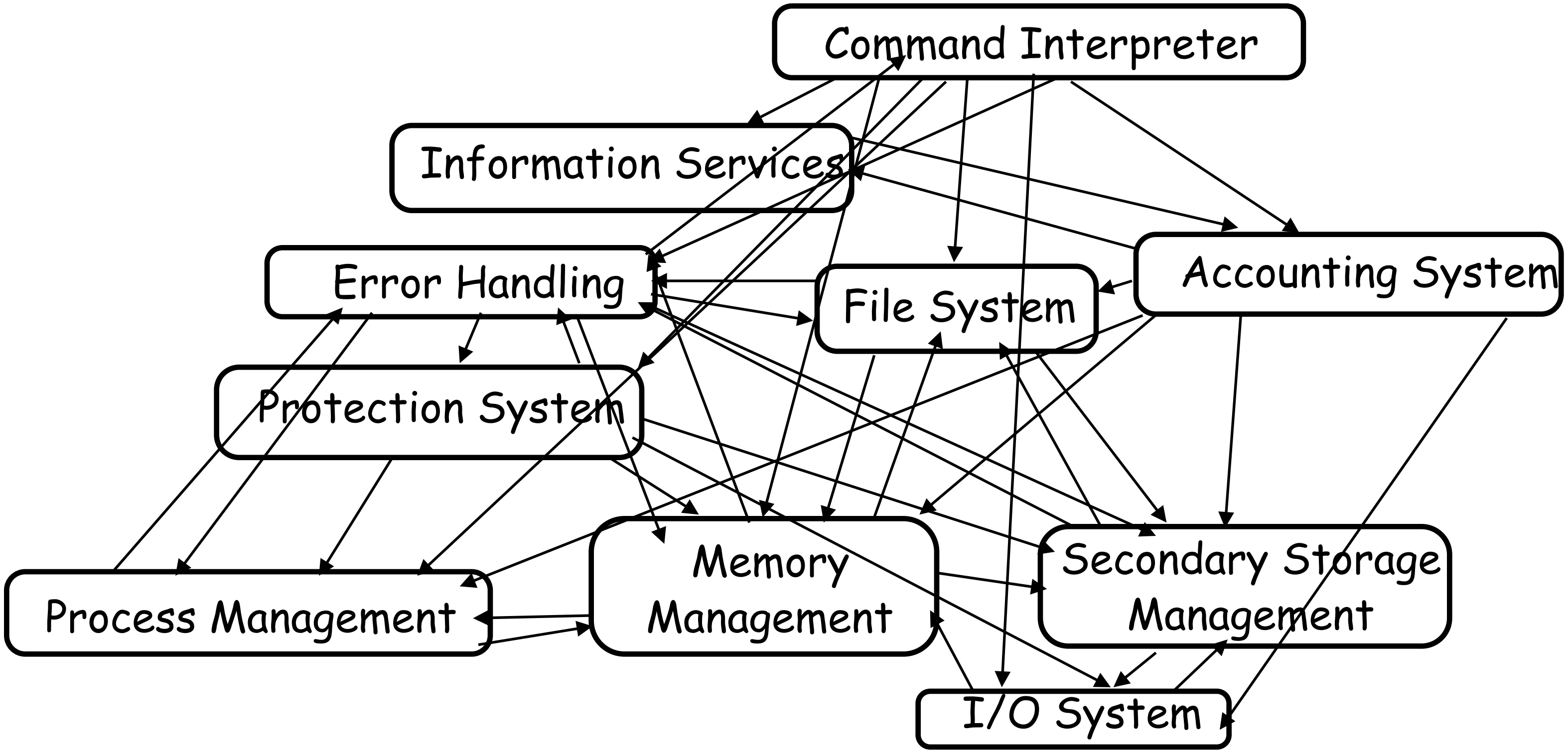
- processes
- memory
- I/O
- secondary storage
- file systems
- protection
- shells (command interpreter, or OS UI)
- GUI
- Networking





OS structure

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

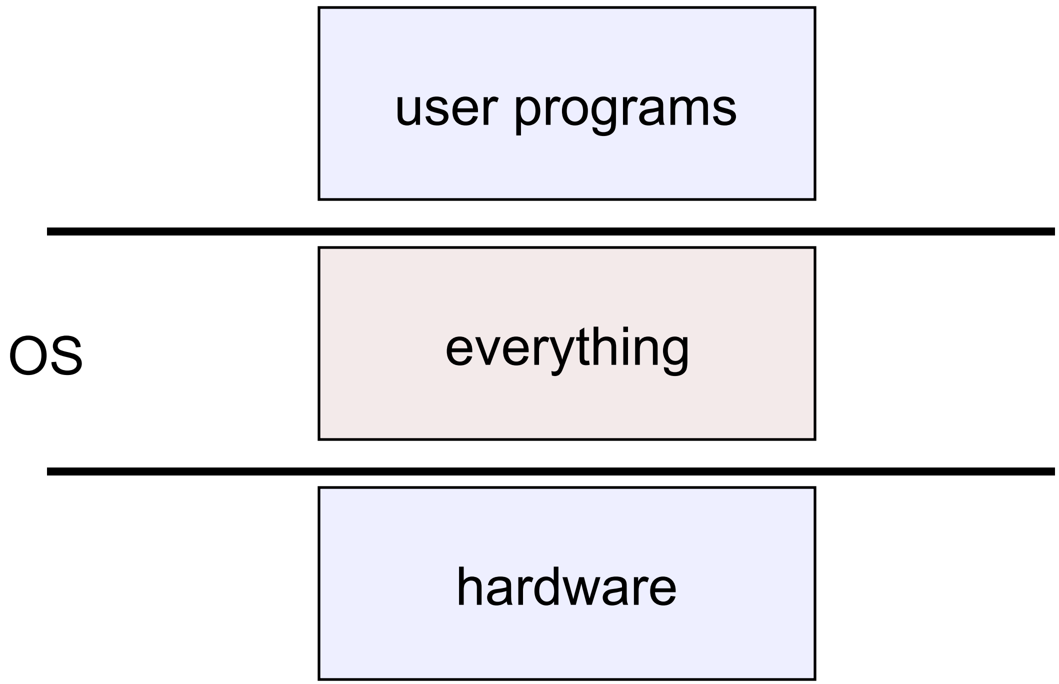


OS structure

- 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:

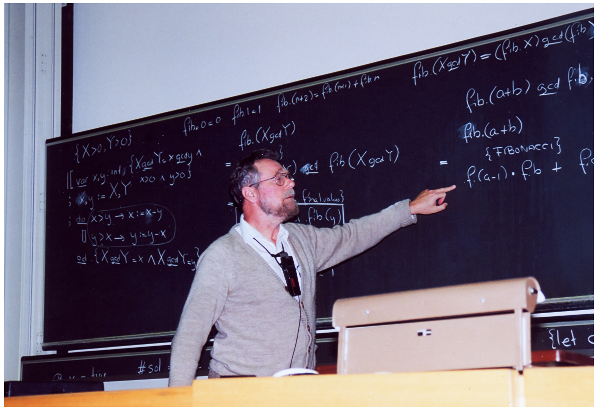


Monolithic design

- 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

- The 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
  - 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

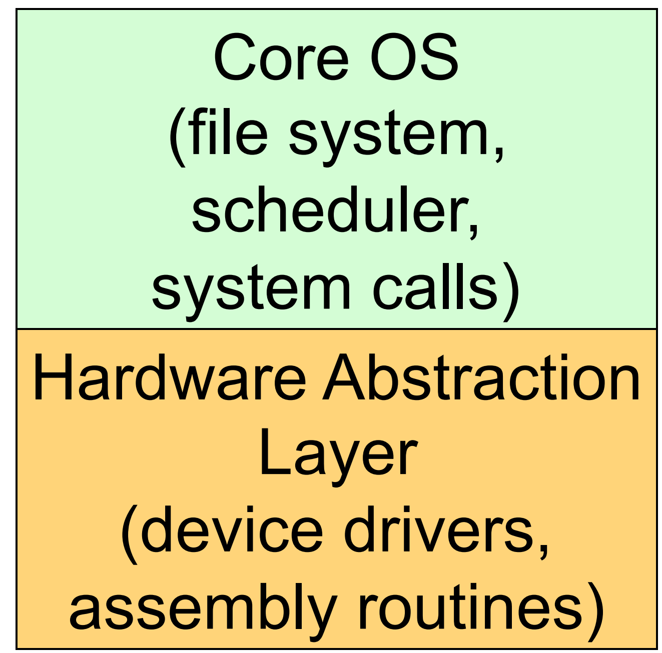


Problems with layering

- 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

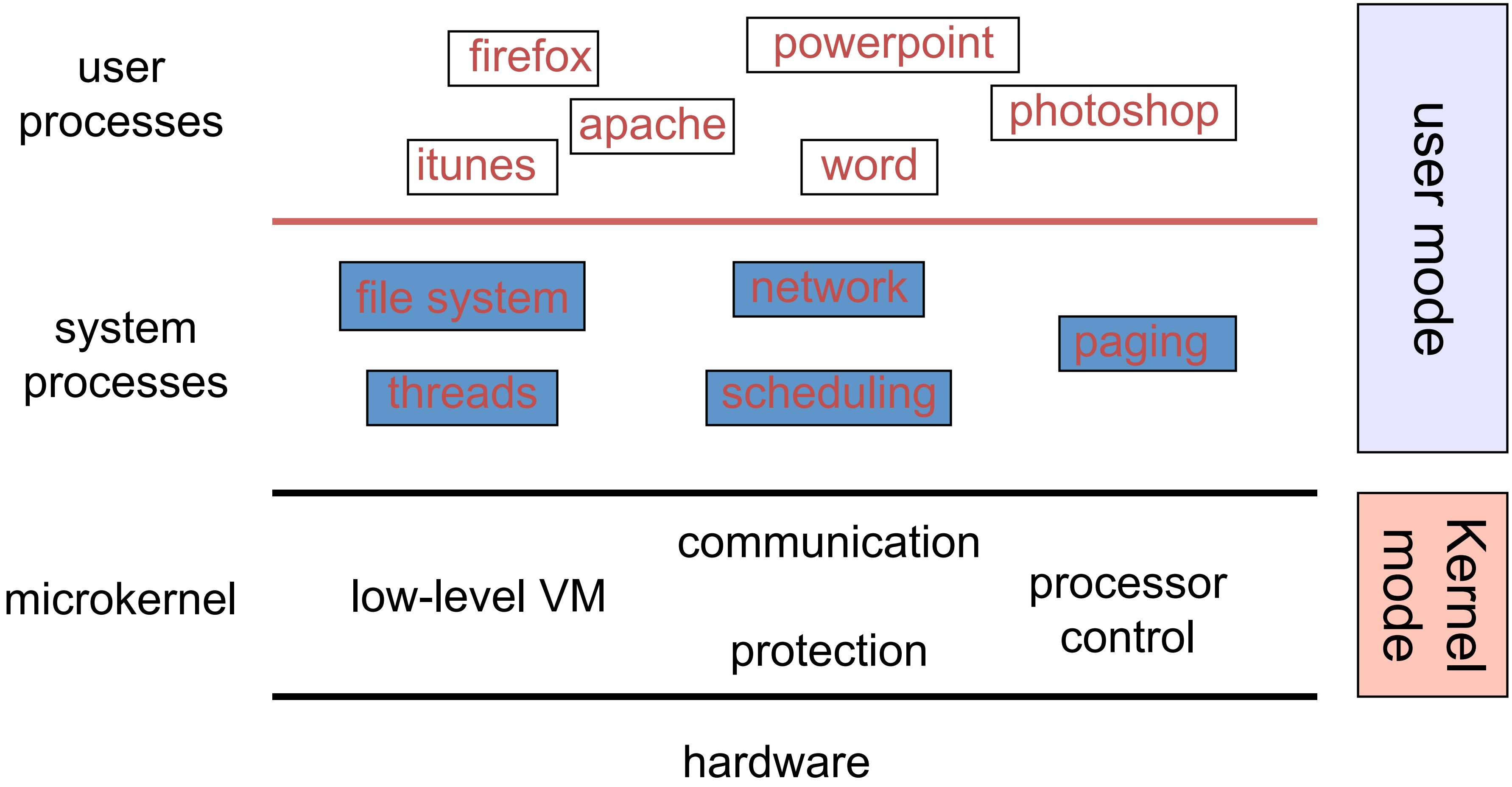
- An example of layering in modern operating systems
- Goal: separates hardware-specific routines from the "core" OS
  - Provides portability
  - Improves readability



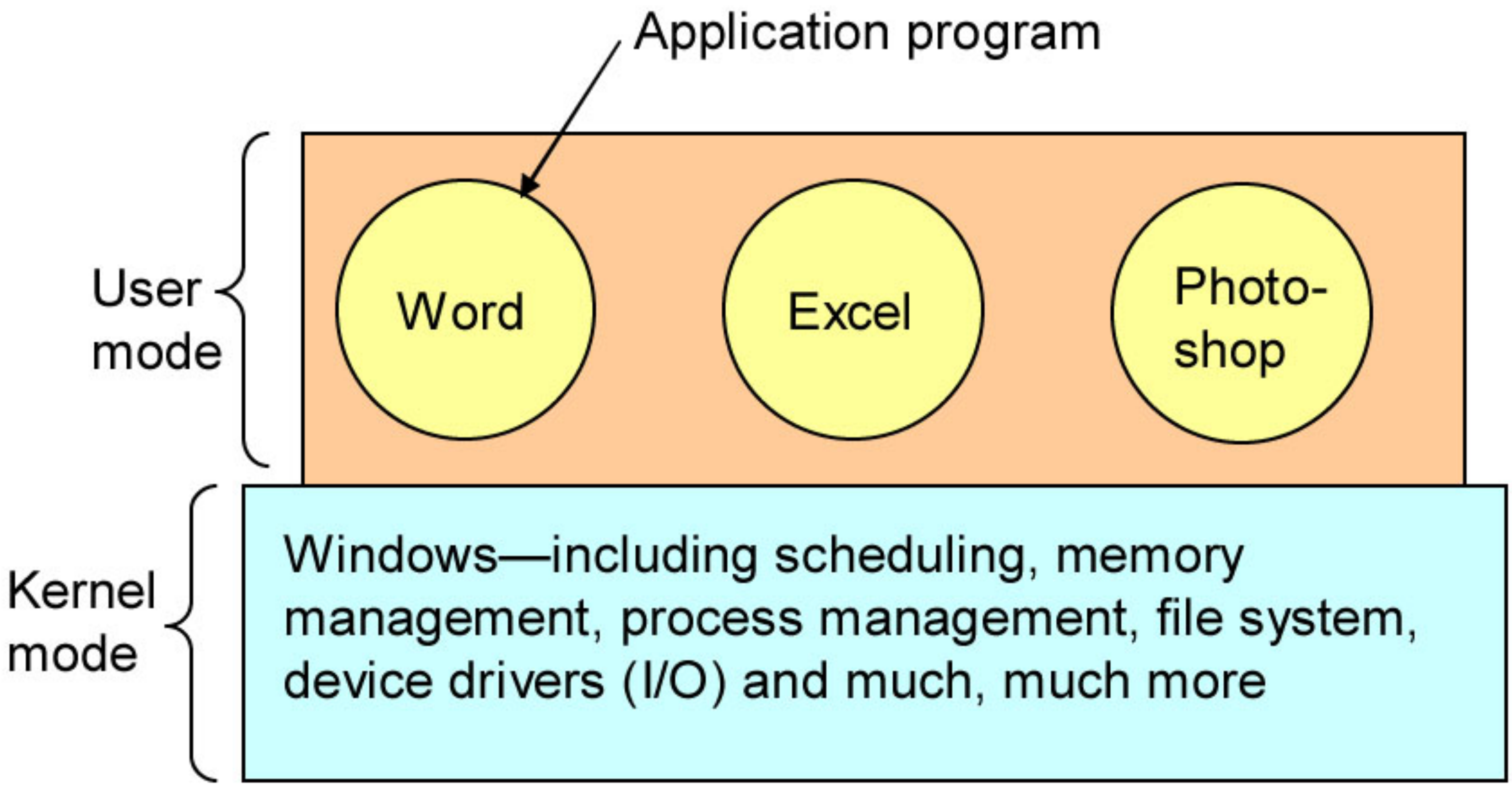
Microkernels

- Popular in the late 80's, early 90's
  – recent resurgence of popularity
- Goal:
  – minimize what goes in kernel
  – organize rest of OS as user-level processes
- This results in:
  – better reliability (isolation between components)
  – 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), in some ways NT (Microsoft)

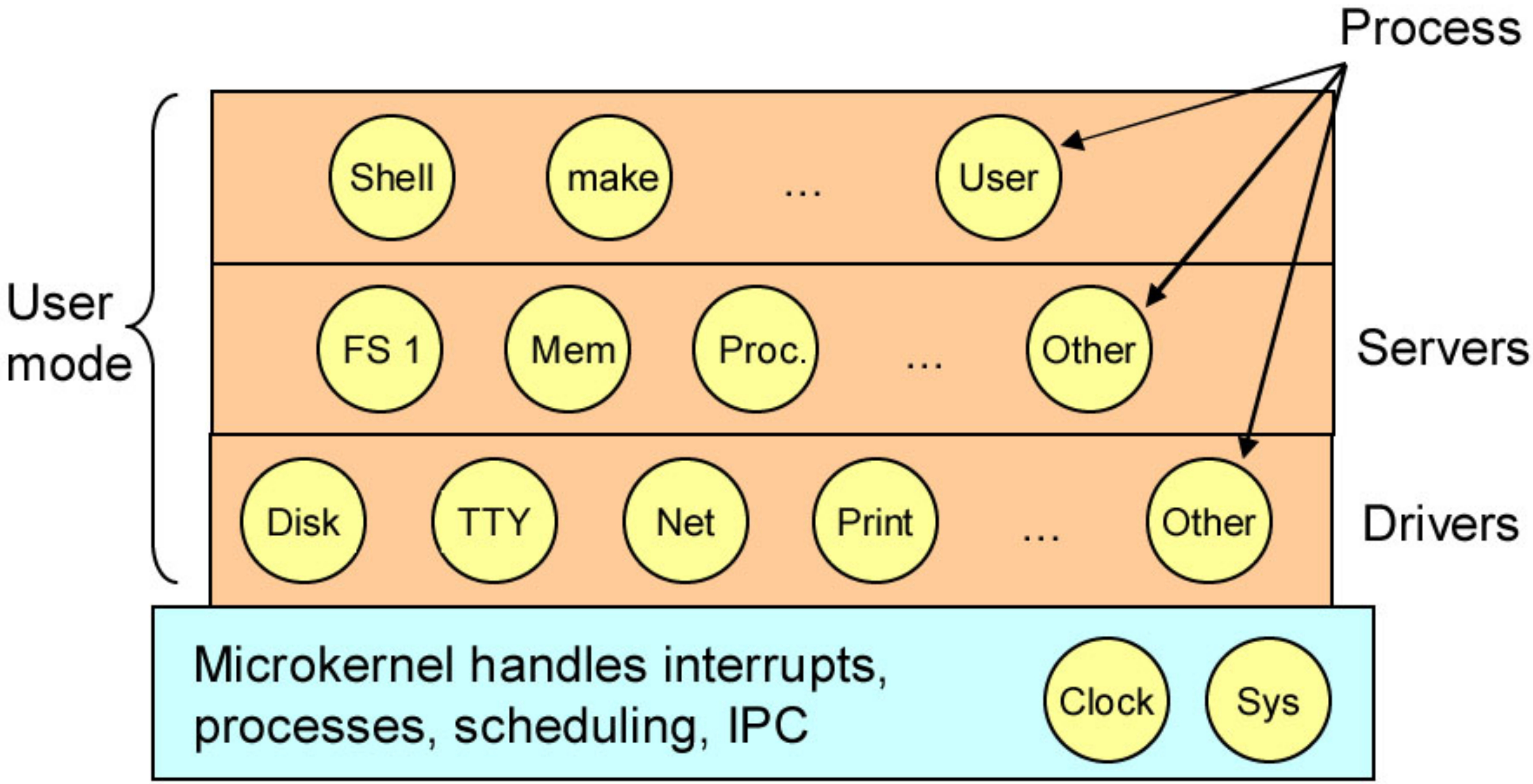
Microkernel structure illustrated



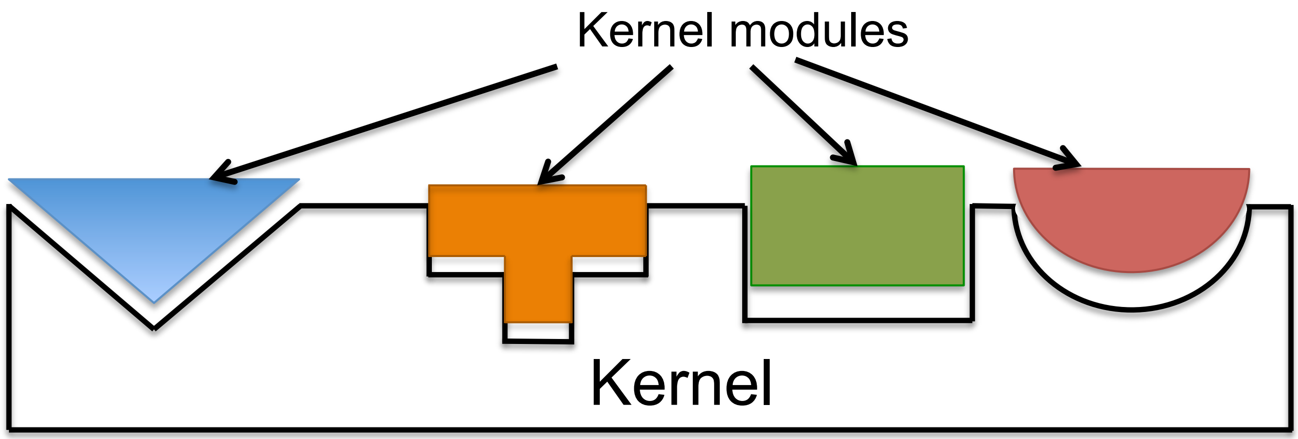
EXAMPLE: WINDOWS



ARCHITECTURE OF MINIX 3



Loadable Kernel Modules



- (Perhaps) the best practice for OS design
- Core services in the kernel and others dynamically loaded
- Common in modern implementations
  - Solaris, Linux, etc.
- Advantages
  - convenient: no need for rebooting for newly added modules
  - efficient: no need for message passing unlike microkernel
  - flexible: any module can call any other module unlike layered model

Summary

- Fundamental distinction between user and priviliged mode supported by most hardware
- OS design has been an 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
- The role and design of an OS are still evolving
- It is impossible to pick one “correct” way to structure an OS