Operating Systems (INFR09047)

2019/2020 Semester 2

Operating Systems Structure

abarbala@inf.ed.ac.uk

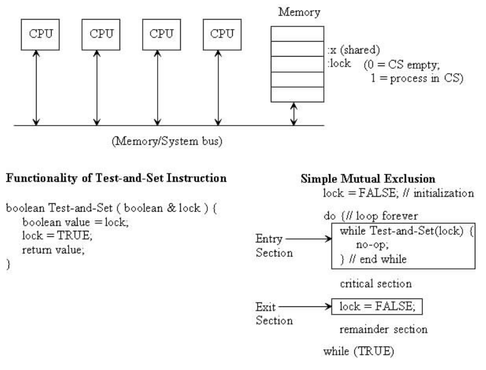
Chapter 2

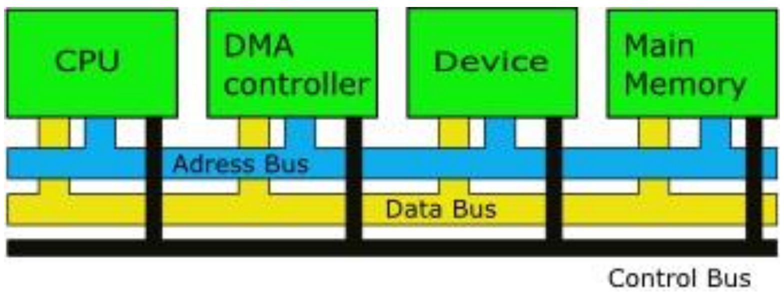
Overview

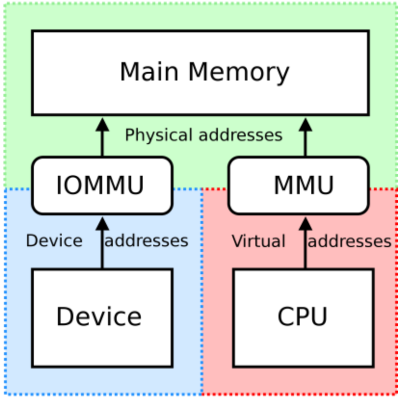
- **Architecture impact**
- **Application-Operating System interaction**
- *Operating System structure*

Hardware Architecture Affects (is Affected by) the OS

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

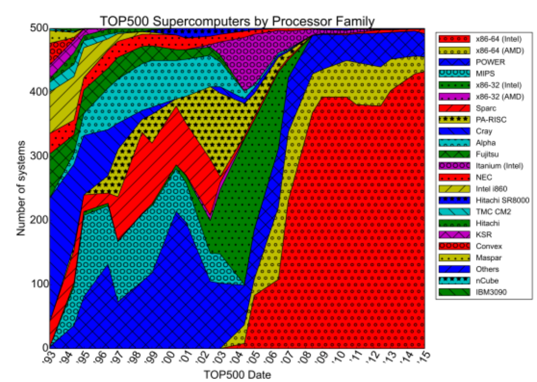






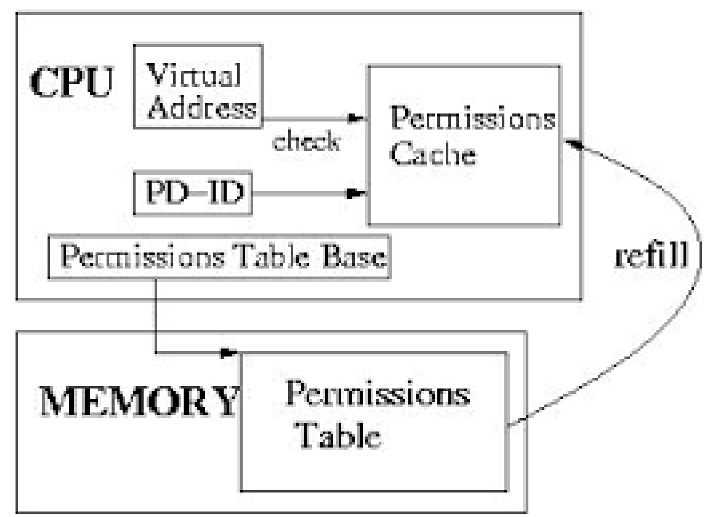
Hardware Architecture Support for the OS

- 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



Hardware Architectural Features Affecting OS

- At the very beginning hardware/software co-design
- Features built primarily to support OS
  - timer (clock) operation
  - memory protection
  - I/O control operations
  - interrupts
  - protected mode(s) of execution
    - kernel vs. user mode
    - privileged instructions
    - system calls
  - virtualization

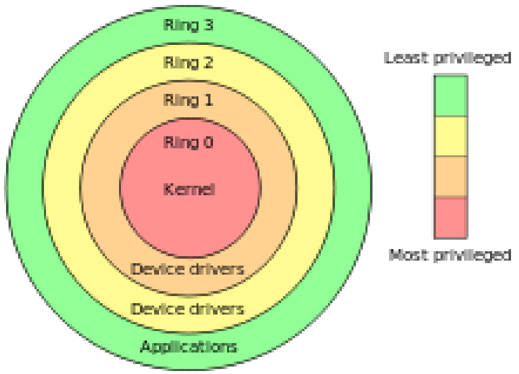


Privileged instructions

- Some instructions are restricted to the OS
  - known as **privileged** instructions
- Only the OS can
  - directly access some classes of I/O devices
  - manipulate memory state management
    - page table pointers, TLB loads, etc.
  - manipulate special ‘mode bits’
    - interrupt priority level
- **Restrictions provide safety and security**

OS protection

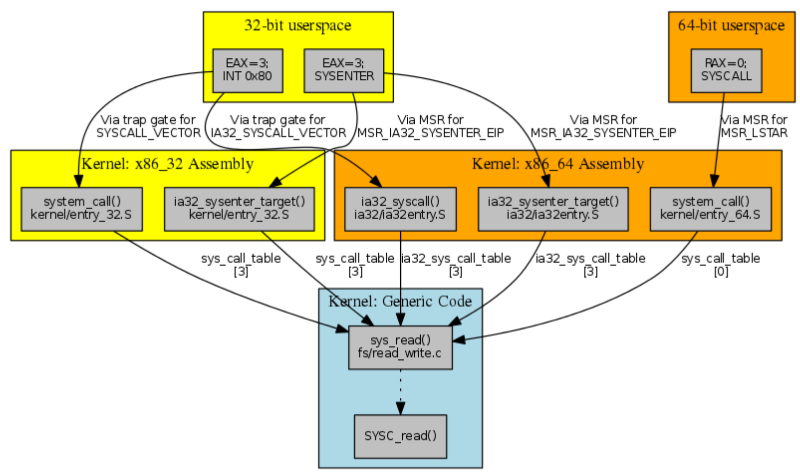
- How does the processor know if a privileged instruction can be executed?
  - Architecture must support **at least two modes** of operation
    - **kernel** mode
    - **user** mode
  - x86 supports 4 protection modes (rings)



  - **Mode** is set by status bit in a protected processor register
    - User programs execute in user mode
    - OS kernel executes in kernel (privileged, supervisor) mode
- Privileged instructions can only be executed in kernel mode
  - When code running in **user mode** attempts to execute a privileged instruction the “Privileged Instruction” exception is triggered

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 – i.e., ask the OS to do that for them
  - OS defines a set of system calls
  - User-mode program executes system call instruction
- **Syscall instruction**
  - “protected procedure call”



Syscall

- The **syscall** instruction, **atomically,** on a single CPU/core
  - 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 arg is a **syscall number**, indicating what OS function to call
    - 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 back to user mode

Kernel Crossing Illustrated

Firefox: read(int fileDescriptor, void *buffer, int numBytes)

Save user PC
PC = trap handler address
Enter kernel mode

user mode

kernel mode

trap handler

PC = saved PC
Enter user mode

Save app state
Verify syscall number
Find sys_read( ) handler in vector table

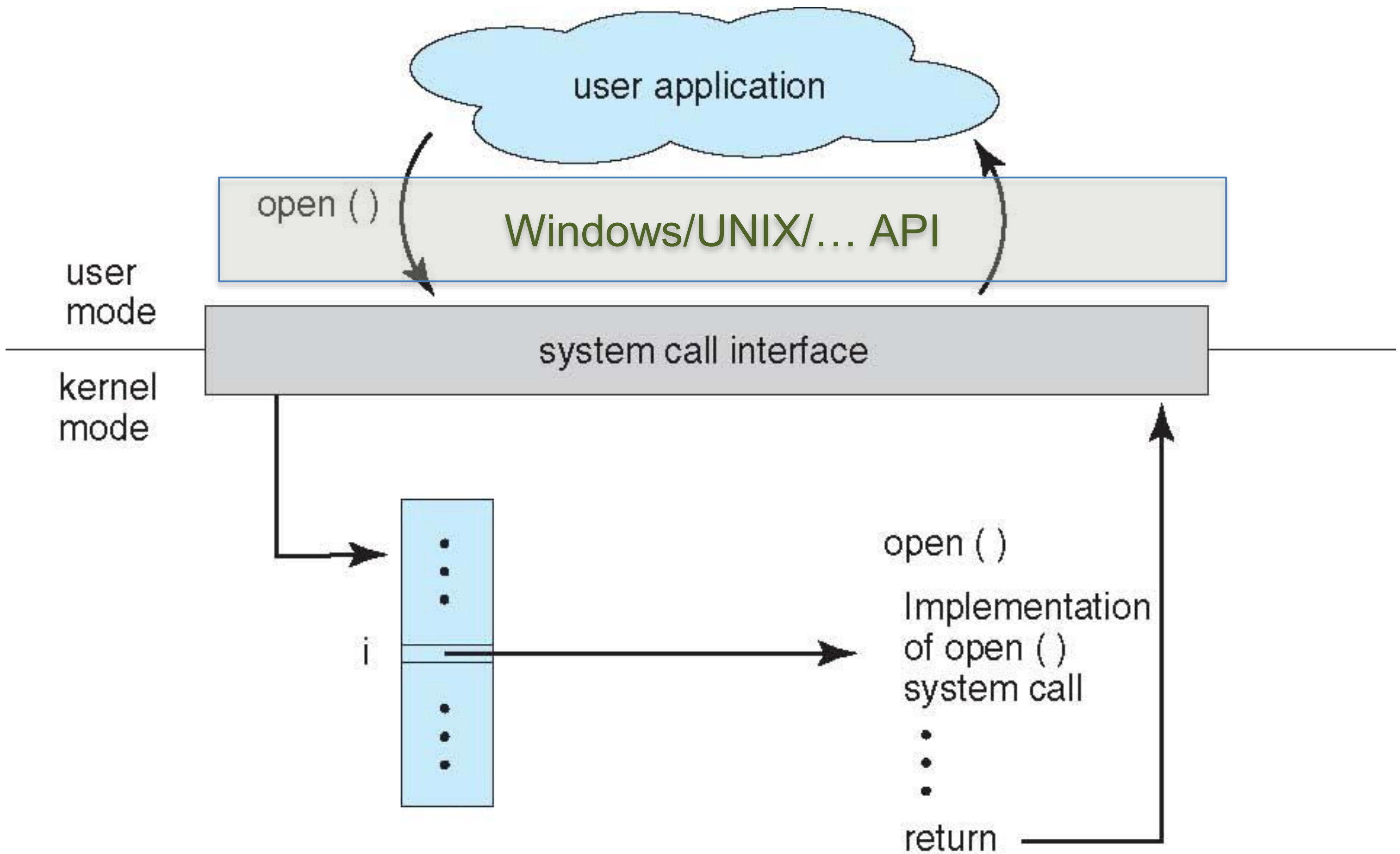
sys_read( ) kernel routine

Verify args
Initiate read
Choose next process to run
Setup return values
Restore app state

http://syscalls.kernelgrok.com/

ERET instruction

API – System Call – OS Relationship

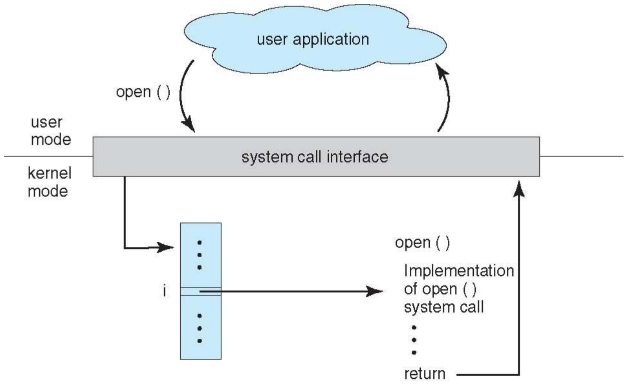


Examples of Windows and Unix System Calls

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

System Call vs Subroutine Call

- Syscall **is not subroutine call**, with the caller specifying the next PC
  - Caller knows where the subroutines are located in memory
  - Subroutines trust each other
  - All subroutines share memory
- 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



OS Services

- All entries to the OS occur via the mechanism just shown
  - Acquiring privileged mode and branching to the trap handler are **inseparable**

- **Terminology**
  - **Exception**: synchronous; unexpected problem with code
  - **Syscall**: synchronous; intended transition to OS
  - **Interrupt**: asynchronous; caused by an external device

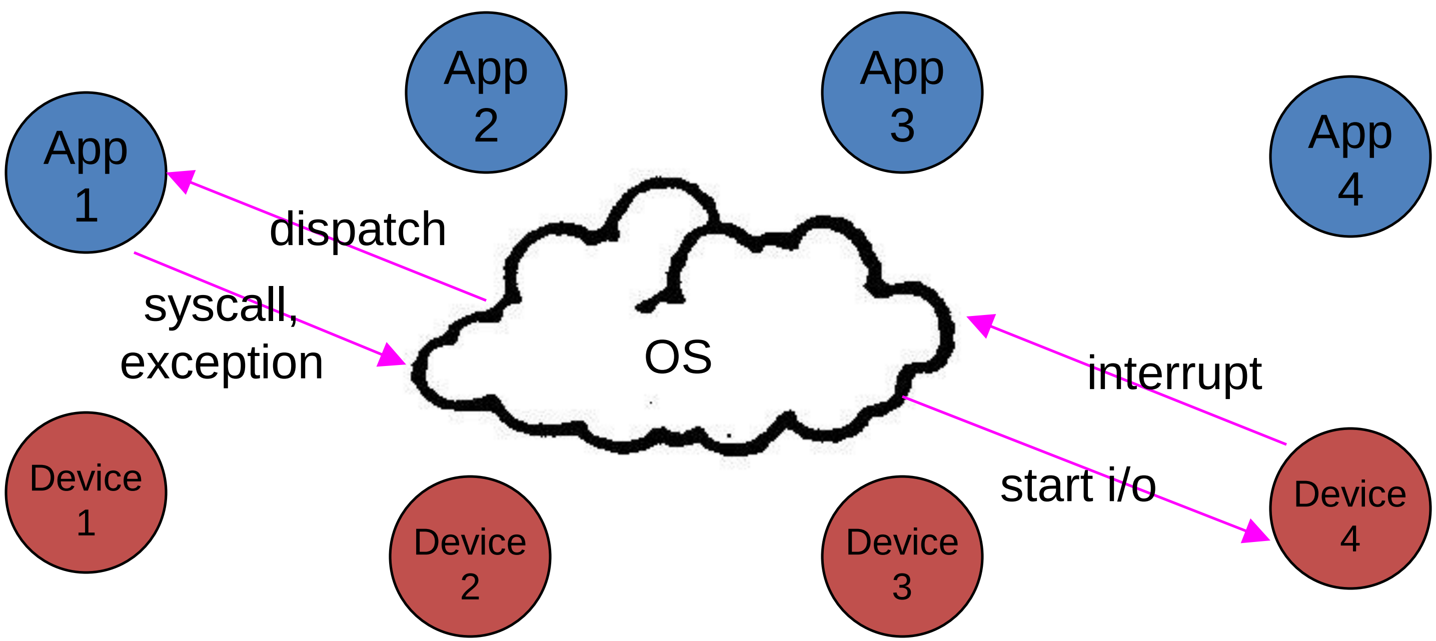
- Privileged instructions and resources sharing are the basis for almost everything OS-related
  - memory protection, protected I/O, limiting user resource consumption, etc.

Overview

- *Architecture impact*
- *Application-Operating System interaction*
- **Operating System structure**

OS Structure

- OS mediates access and abstracts away ugliness
- OS sits between **applications** and the **hardware**
  - Applications **(App)** request services
    - **Explicitly** via syscalls
    - **Implicitly** via exceptions
  - Devices **(Device)** 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
  - **User** goals: convenient to use, easy to learn, reliable, safe, and fast
  - **System** goals: easy to design, implement, and maintain, as well as flexible, reliable, error-free, and efficient
- Affected by choice of hardware, type of system

Operating System Design and Implementation

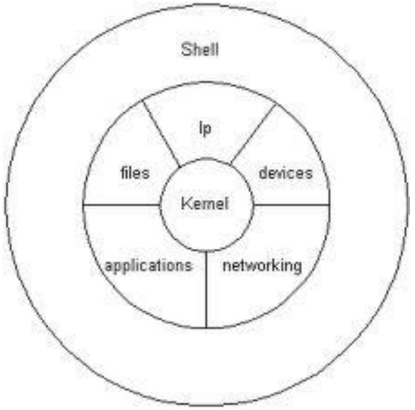
- Important **principles to separate**
- **Policy**: ***What*** will be done? (Decision)
- **Mechanism**: ***How*** to do it?

- Separation allows maximum flexibility
  - Policies are likely to change across places or over time
  - A general mechanism can support a wide range of policies

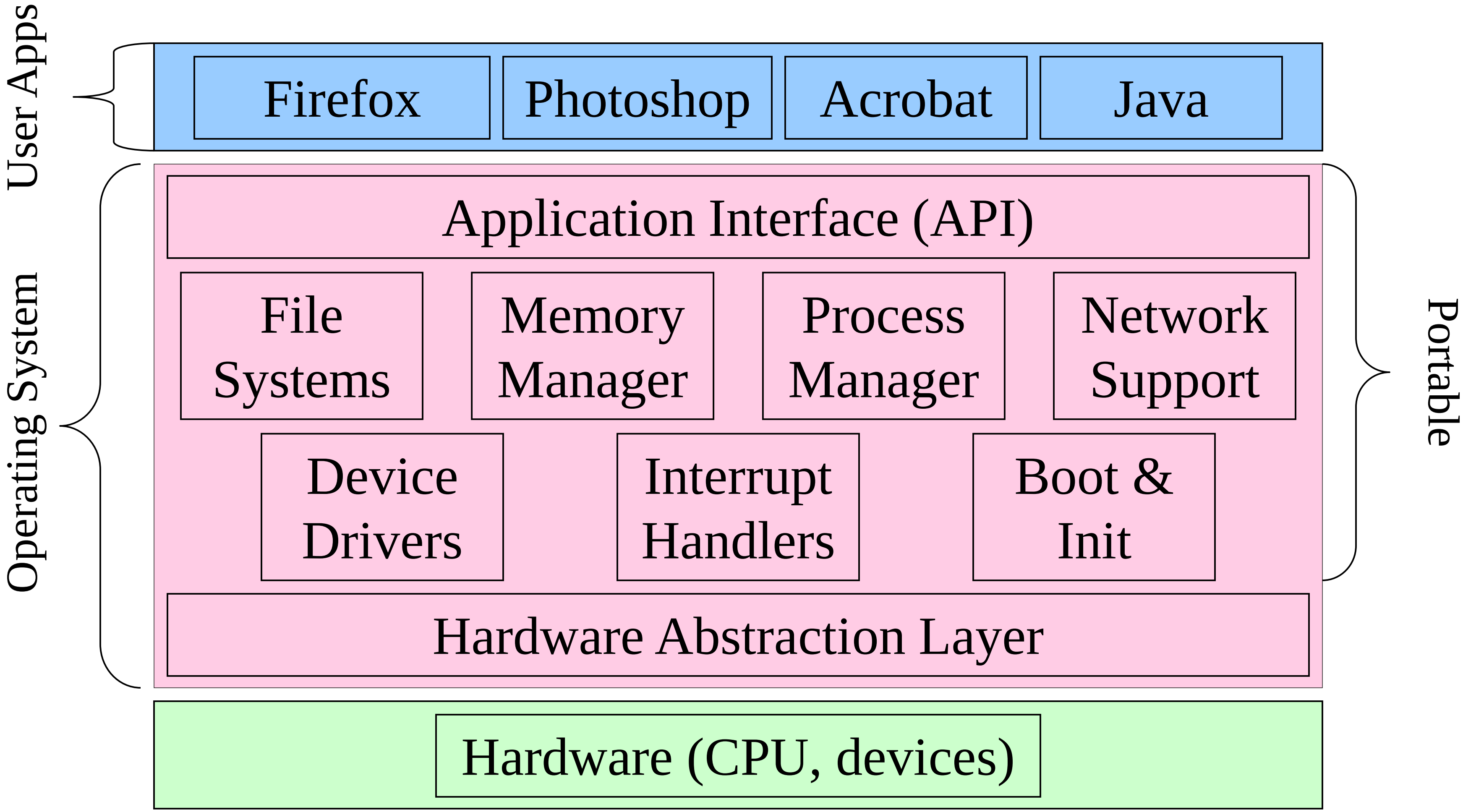
- Microkernel OSes are based on such principle
  - A core kernel implements the mechanisms
  - Policies are implemented outside the core kernel
    - Easily modifiable

Major OS Services

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

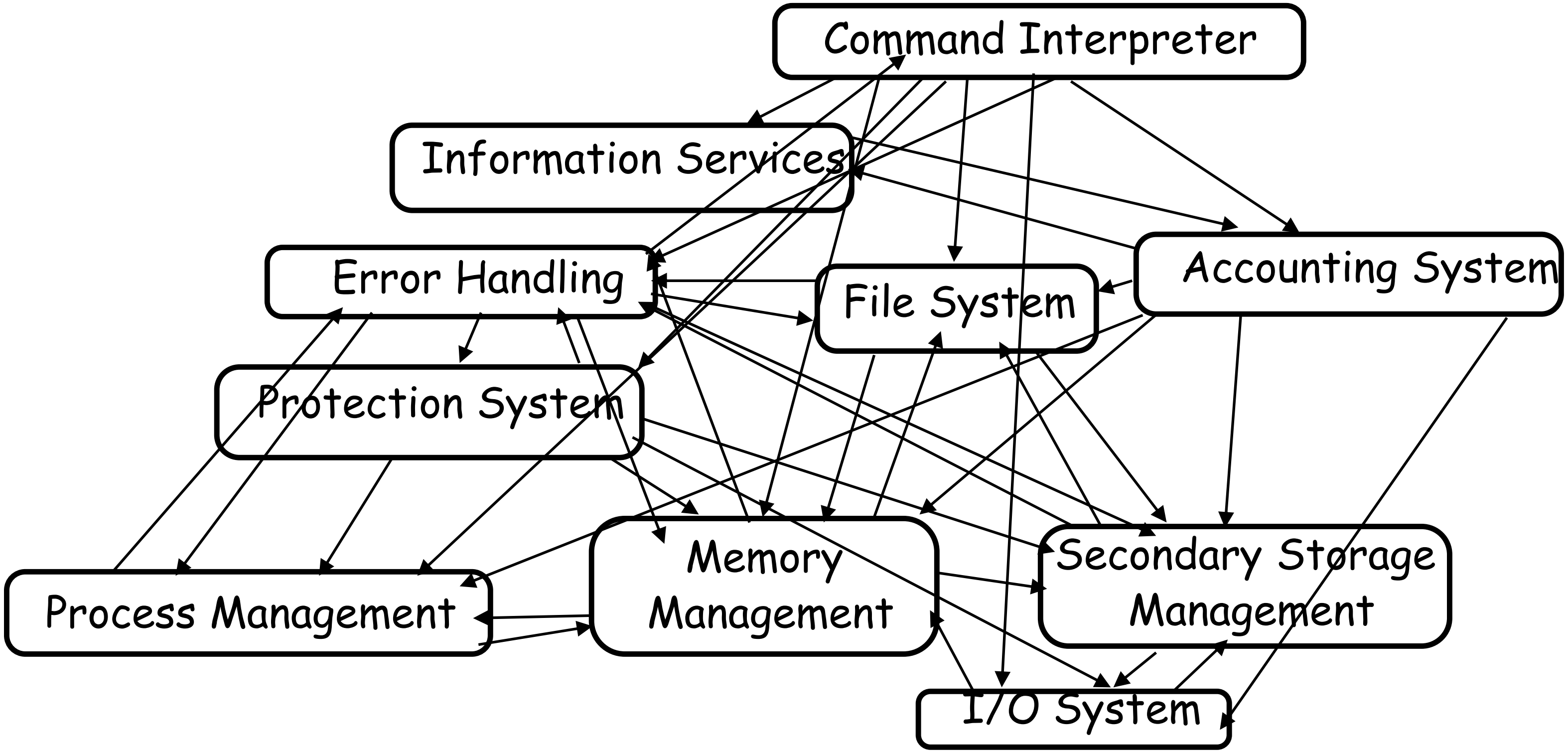


Software Layers and OS Services



OS Structure #1

- It's not always clear how to stitch OS **services** together



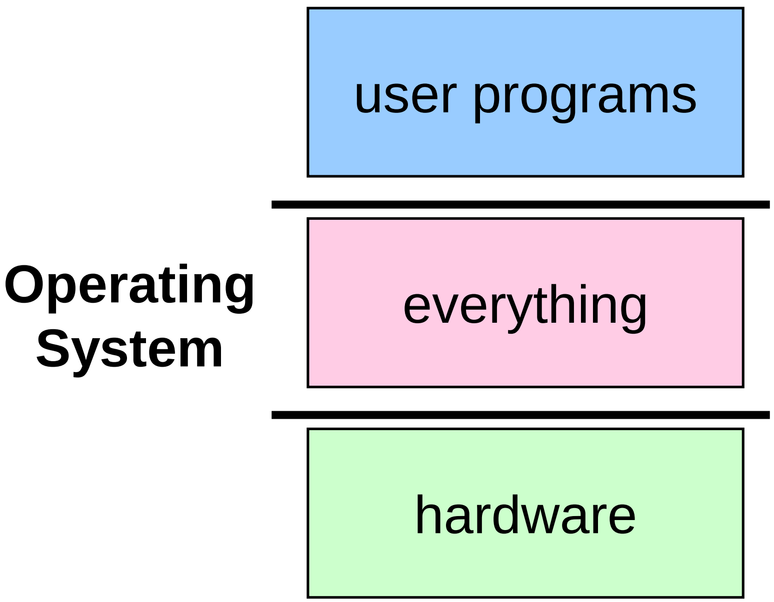
OS Structure #2

- Major issues
  - how do we organize all these?
  - what are all of the code modules, and where do they exist?
  - how do they cooperate?

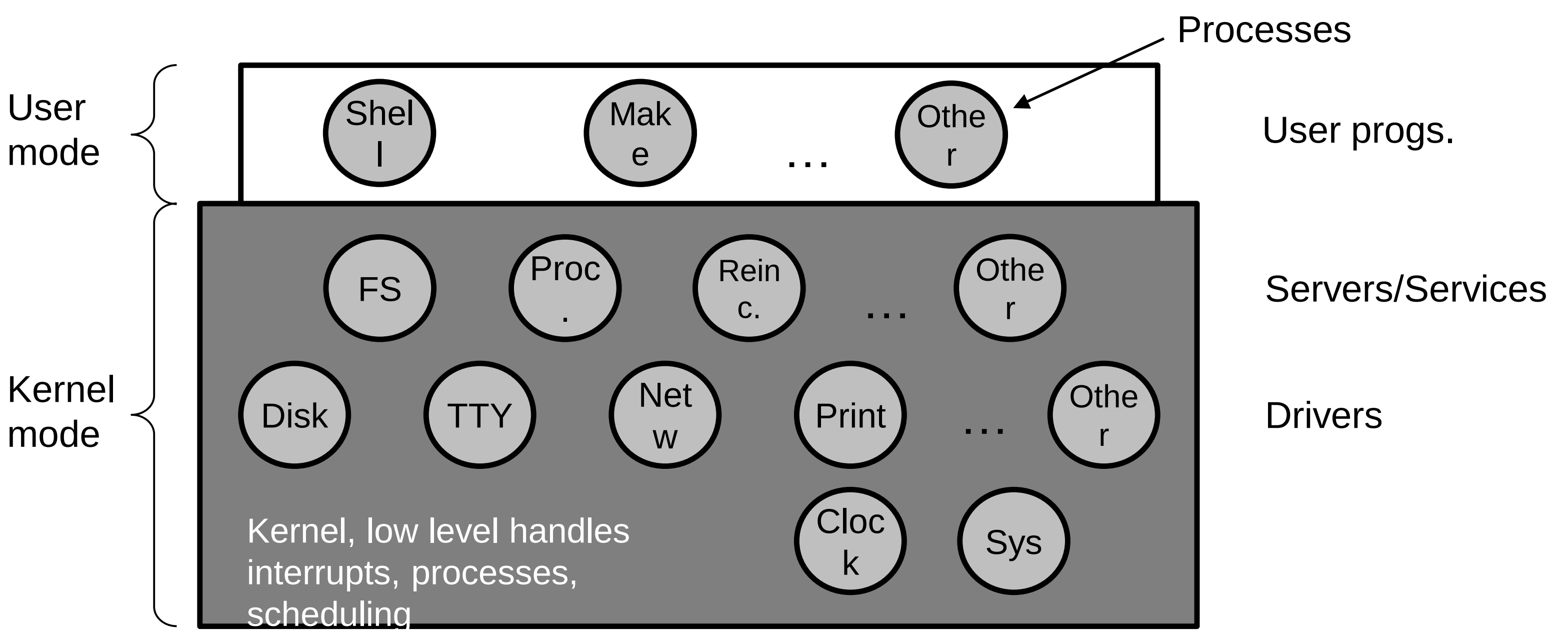
- Massive software engineering and design problem
  - design a large, complex program that
    - performs well
    - is reliable
    - is extensible
    - is backwards compatible
    - etc.

Monolithic **OS Design** #1

- Likely the earliest OS organization
- **UNIX** was built as monolithic
  - **Linux** is built as monolithic



Monolithic Example: Linux

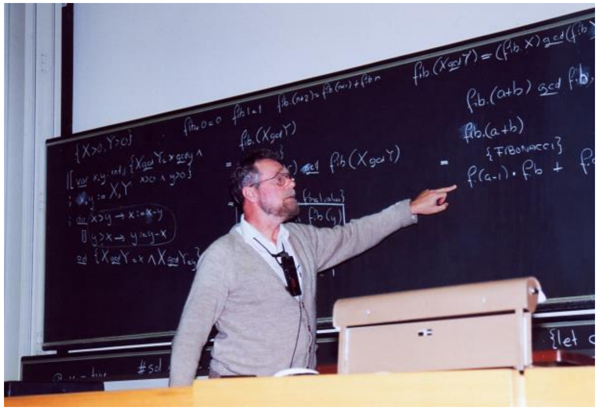


Monolithic **OS Design** #2

- Major **advantage**
  - **cost** of subsystems 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 OS subsystems to simplify its design and implementation

Layered **OS Design**

- 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

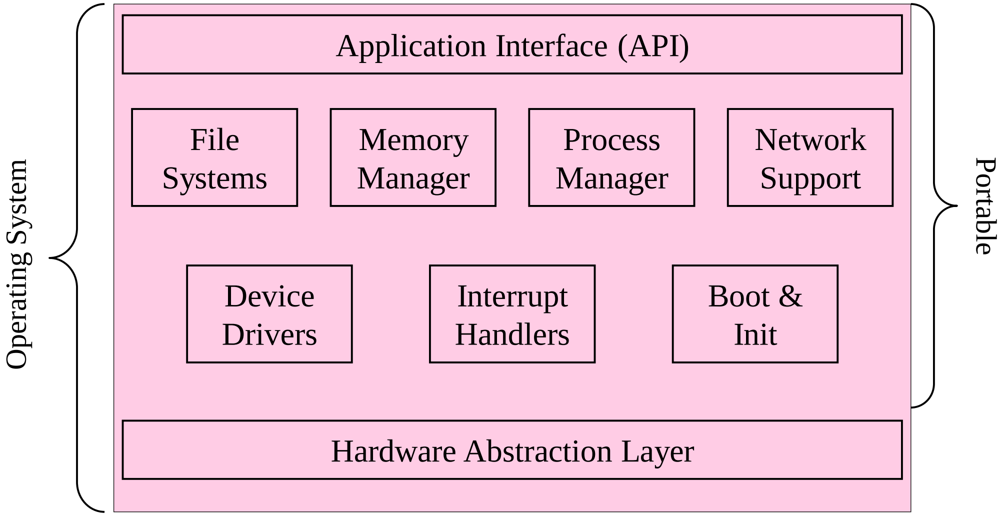


Problems with layering

- **Imposes** hierarchical structure
  - but real systems are more complex
    - File system requires virtual memory services
    - Virtual memory 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

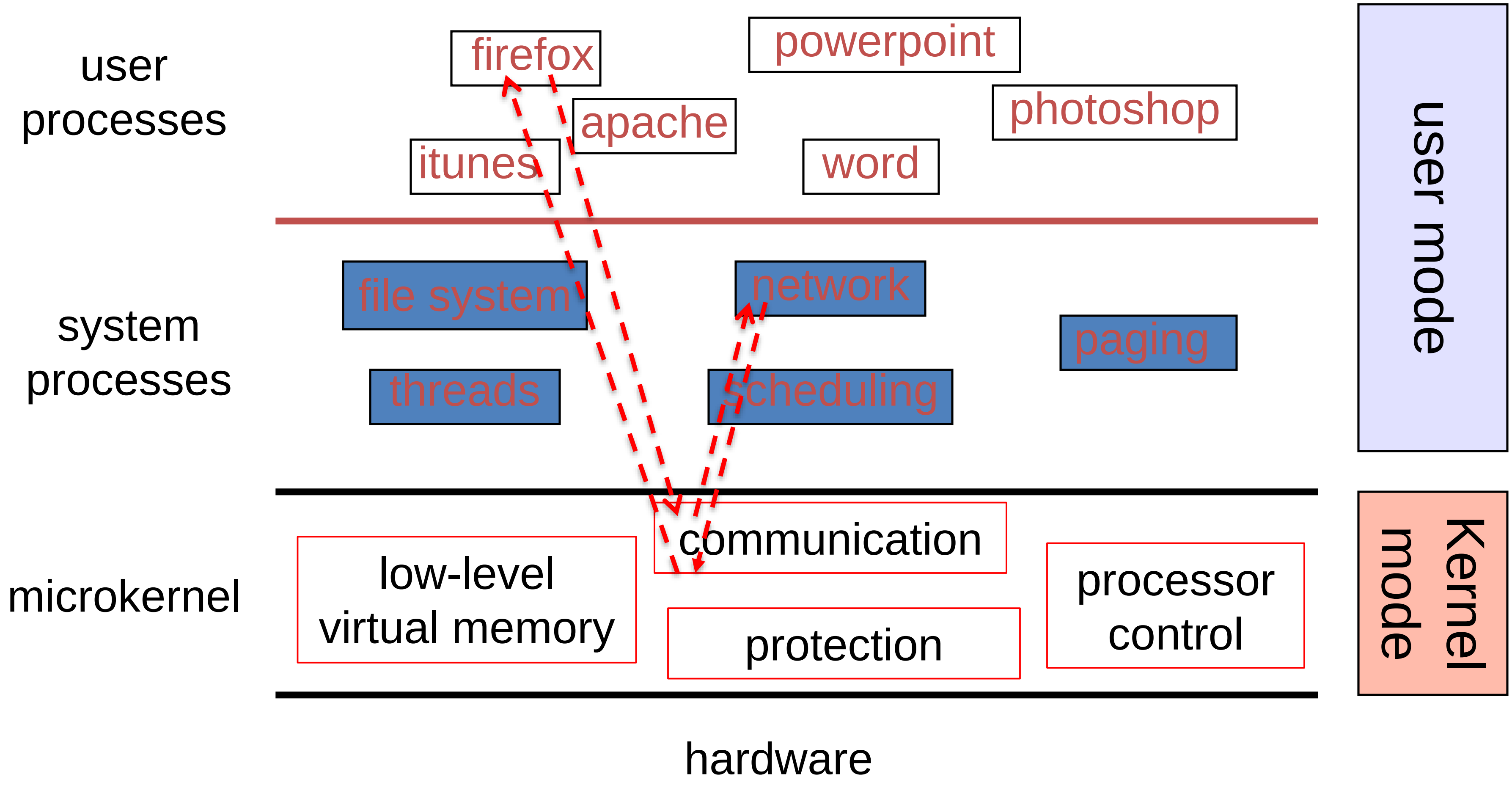
- An **example** of layering in modern operating systems
  - Windows, etc.
- Goal: separates hardware-specific routines from the **core kernel** of the OS
  - Provides portability
  - Improves readability



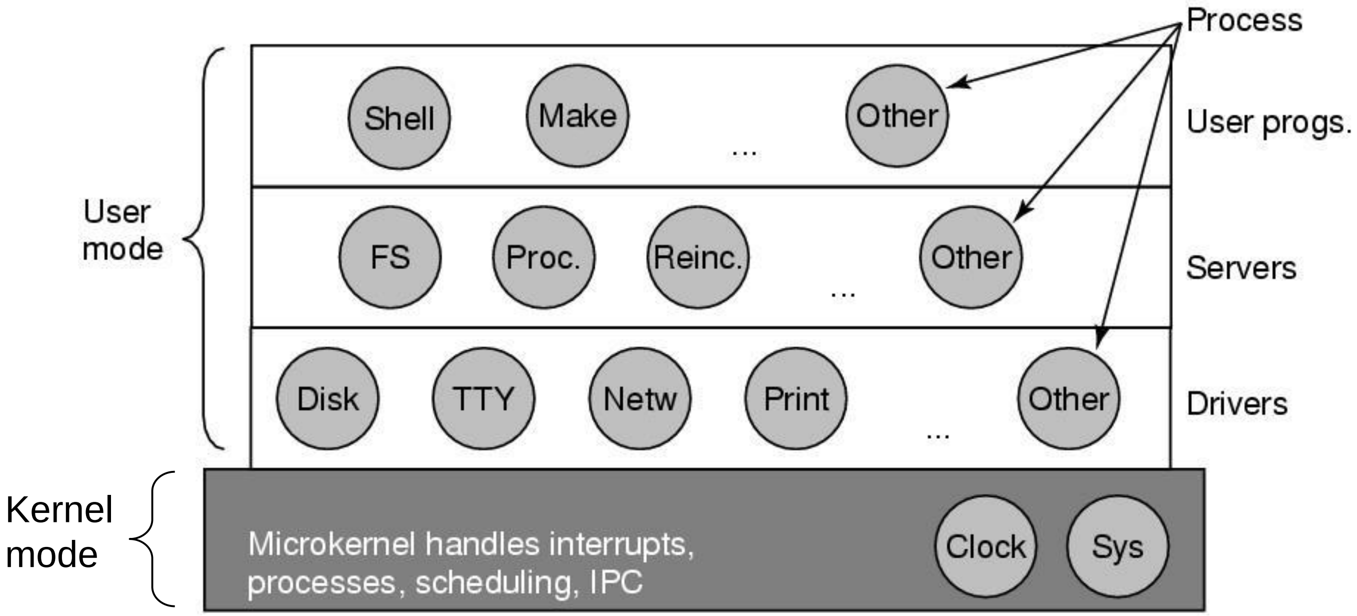
Microkernel **OS Design**

- 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 (services)
- 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), MINIX (UNIX-like OS from Amsterdam)

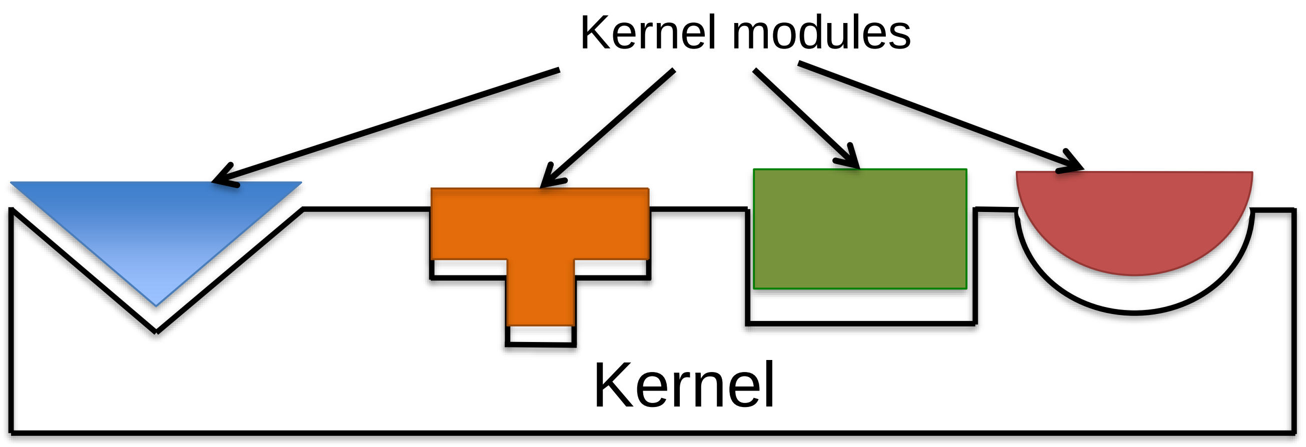
Microkernel Structure Illustrated



Microkernel Example: MINIX



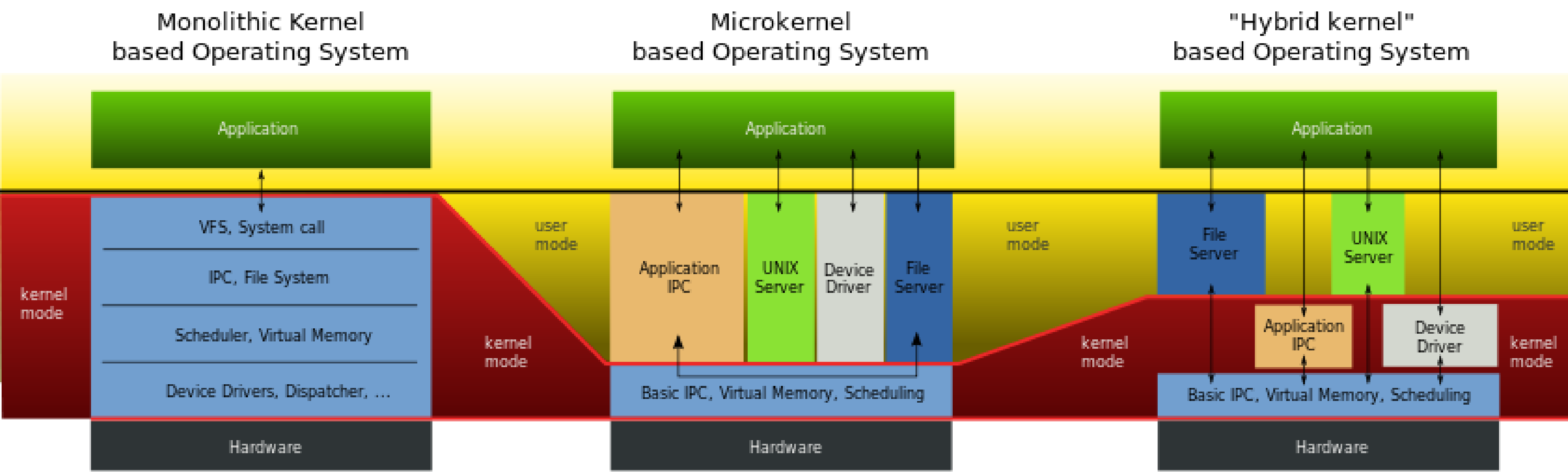
Loadable Kernel Modules



- **Core services** in the kernel, **others** dynamically loaded
- Common in modern implementations
  - **Monolithic**: load the code in kernel space (Solaris, Linux, etc.)
  - **Microkernel**: load the code in user space (any)
- 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

Hybrid **OS Design**

- Many **different** approaches
  - Key idea: exploit the benefits of monolithic and microkernel designs
  - Windows, Xnu/Darwin, DragonFly BSD, …
- Extensibility via kernel modules



Picture Copyright of Wikipedia

Summary

- Fundamental distinction between user and privileged modes supported by most hardware
- 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
- The role and design of an OS are still evolving
- There is no “ideal” OS structure