Machine and Operating System Organization

Raju Pandey
pandey@cs.ucdavis.edu
Department of Computer Sciences
University of California, Davis

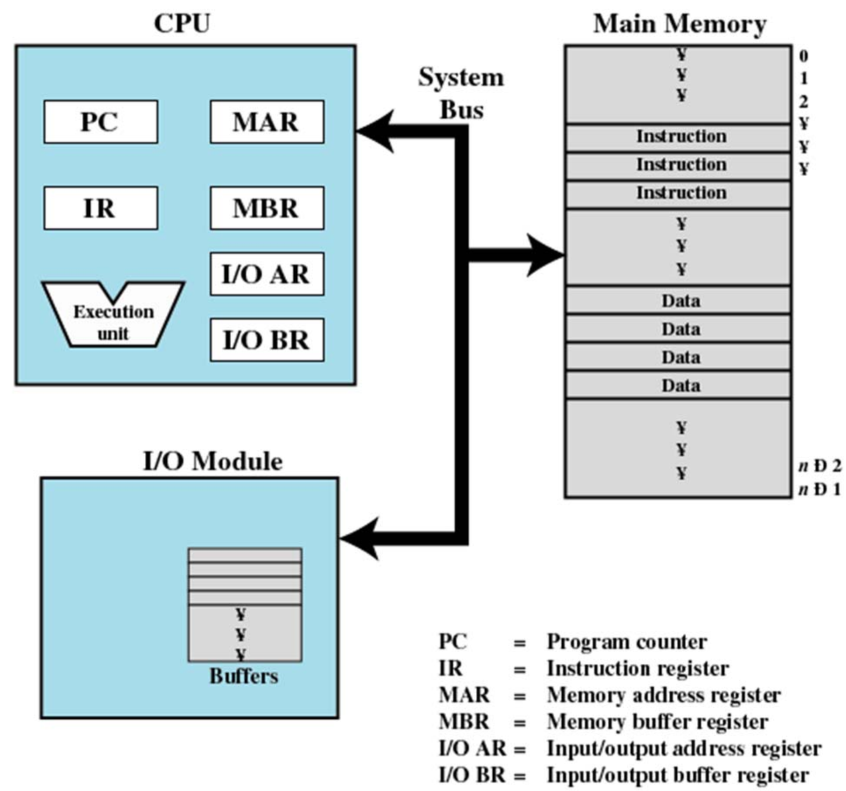
Overview

- Organization of Computing Systems
- Organization of operating systems
  - Software Engineering view:
    - How is operating system software organized?
    - What are implications of specific organization
    - Case studies
  - Abstraction view:
    - How does end user see operating system?
    - How does control transfer between applications and operating systems
    - Design issues
- Thanks: This lecture notes is based on several OS books and other OS classes
  - Silbersatz. Et. al
  - Stallings
  - Bic and Shaw
  - G. Nutt
  - Free BSD book
  - Professor Felix Wu's notes for ECS 150
  - U. Washington (451: Professors Gribble, Lazowska, Levy and Zahorjan)

Part I: Computing System Organization
Background material

Basic Elements

- Processor
- Main Memory
  - volatile
  - referred to as real memory or primary memory
- I/O modules
  - secondary memory devices
  - communications equipment
  - terminals
- System bus
  - communication among processors, memory, and I/O modules



Processor Registers

- **User-visible registers**: enable programmer to minimize main-memory references by optimizing register use
  - *Data*
  - *Address*: Index, Segment pointer, Stack pointer
- **Control and status register:** Used by processor to control operating of the processor
  - Used by privileged operating-system routines to control the execution of programs
  - Program Counter (PC)
    - Contains the address of an instruction to be fetched
  - Instruction Register (IR)
    - Contains the instruction most recently fetched
  - Program Status Word (PSW)
    - Condition codes: Bits set by the processor hardware as a result of operations. For instance, Positive result, Negative result, Zero, Overflow
    - Interrupt enable/disable
    - Supervisor/user mode

Instruction Execution

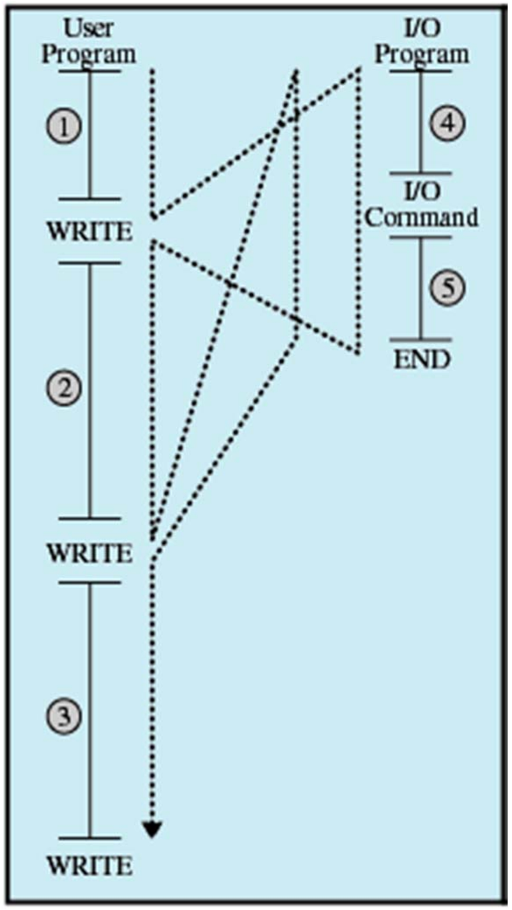
- Fetch: CPU fetches the instruction from memory
  - Program counter (PC) holds address of the instruction to be fetched next
  - Program counter is incremented after each fetch
  - Fetched instruction is placed in the instruction register
- Decode
  - Categories of IR
    - Processor-memory
      - Transfer data between processor and memory
    - Processor-I/O
      - Data transferred to or from a peripheral device
    - Data processing
      - Arithmetic or logic operation on data
    - Control
      - Alter sequence of execution
- Execute: Processor executes each instruction

Interrupts

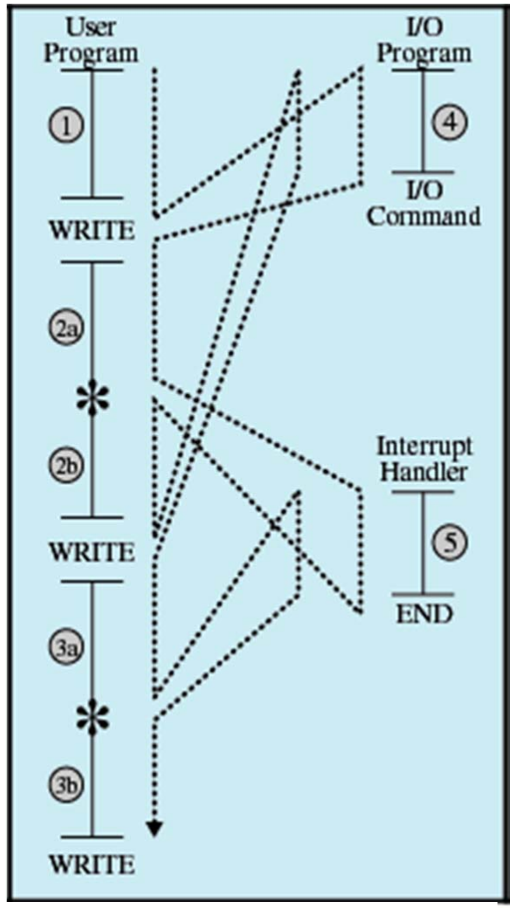
- Interrupt the normal sequencing of the processor
- Most I/O devices are slower than the processor
  - Processor must pause to wait for device

Program	Generated by some condition that occurs as a result of an instruction execution, such as arithmetic overflow, division by zero, attempt to execute an illegal machine instruction, and reference outside a user's allowed memory space.
Timer	Generated by a timer within the processor. This allows the operating system to perform certain functions on a regular basis.
I/O	Generated by an I/O controller, to signal normal completion of an operation or to signal a variety of error conditions.
Hardware failure	Generated by a failure, such as power failure or memory parity error.

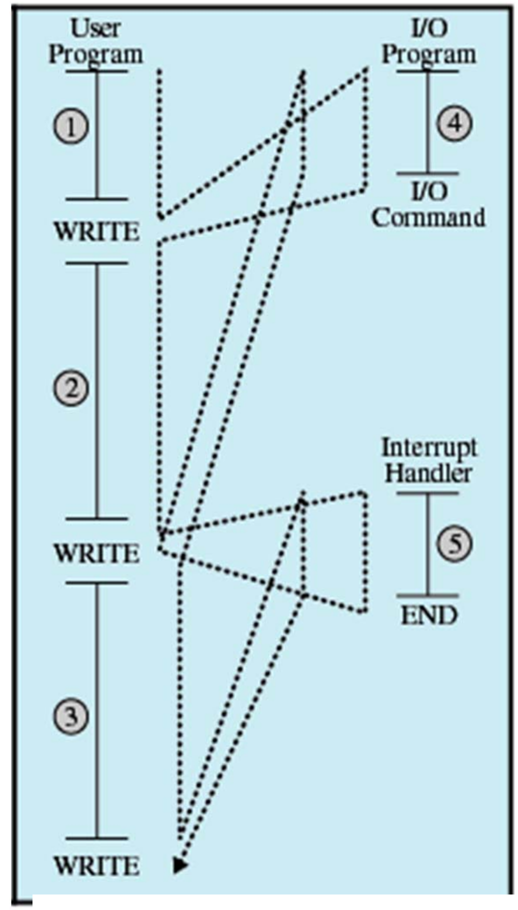
Program Flow of Control



(Without Interrupts)



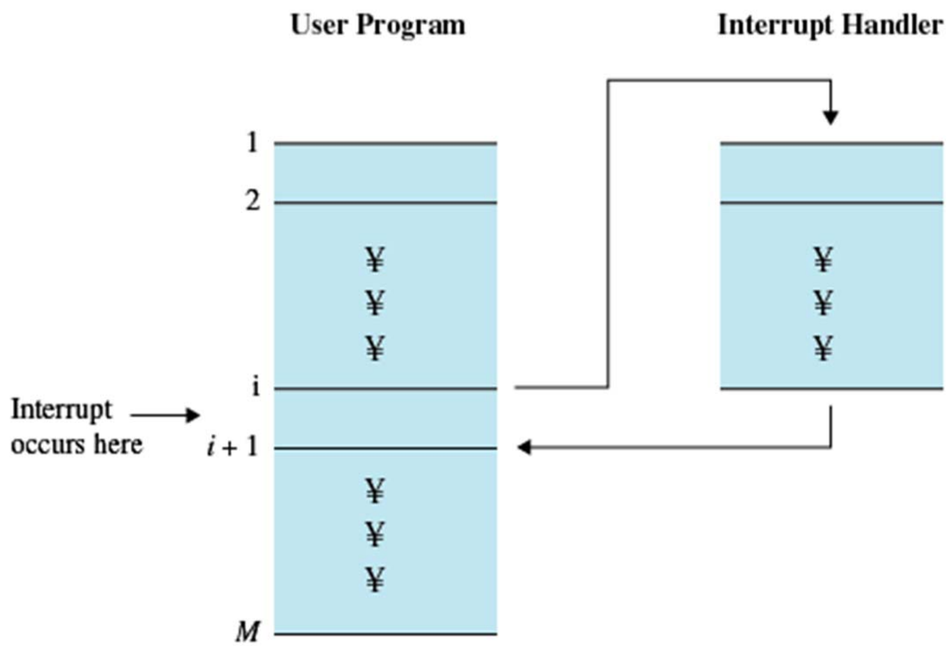
(Interrupts; Short I/O Wait)



Interrupts; Long I/O Wait)

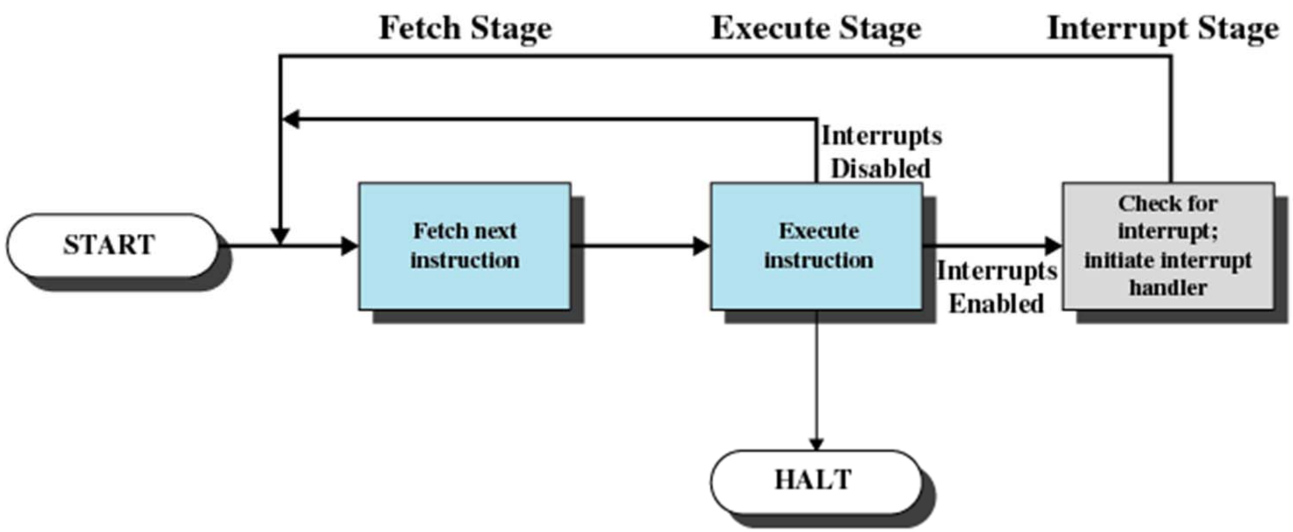
Interrupts and processing of interrupts

- Interrupts: Suspends the normal sequence of execution
- Interrupt handler: respond to specific interrupts
  - Program to service a particular I/O device
  - Generally part of the operating system

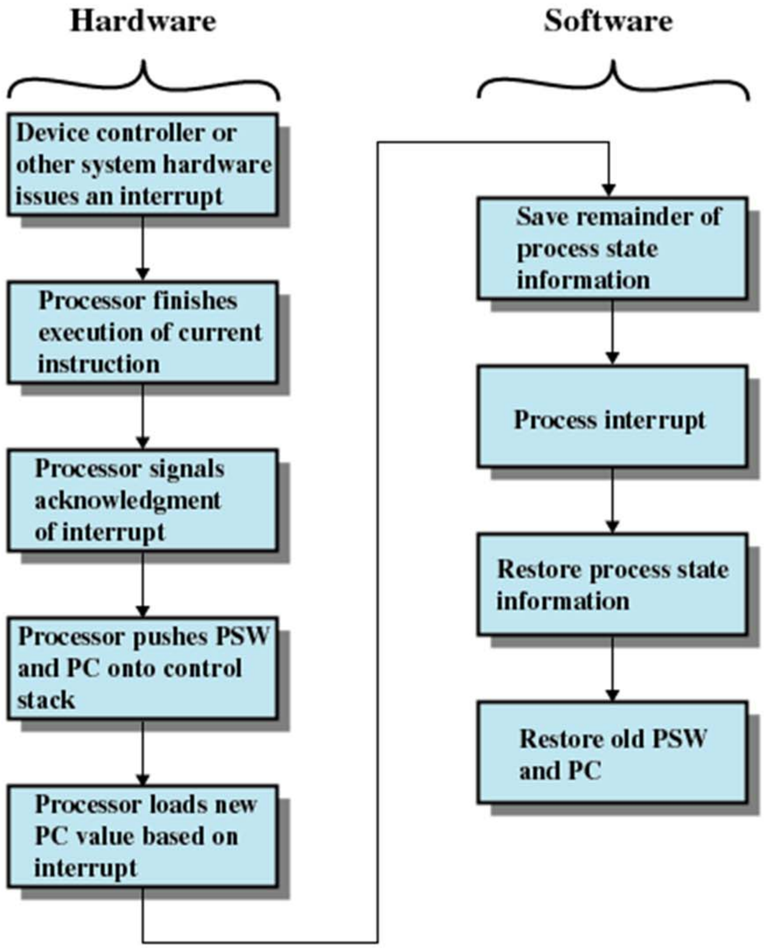


Interrupt Cycle

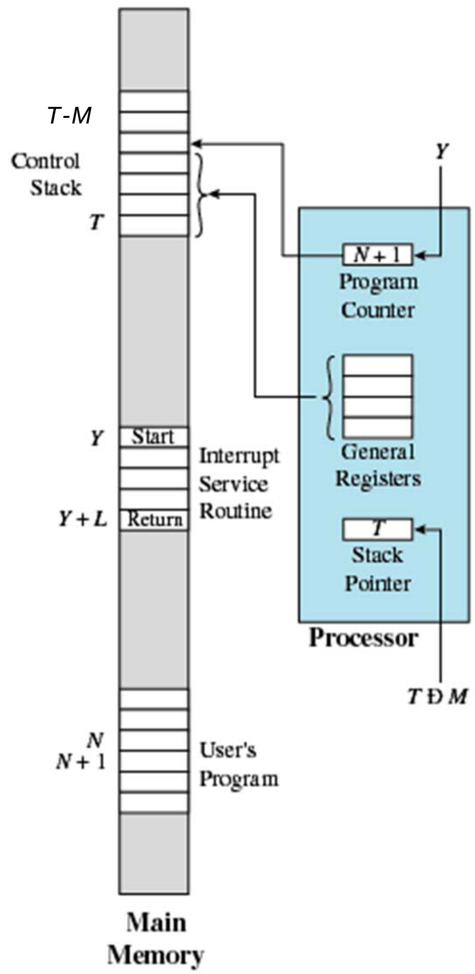
- Processor checks for interrupts
- If no interrupts fetch the next instruction for the current program
- If an interrupt is pending, suspend execution of the current program, and execute the interrupt-handler routine



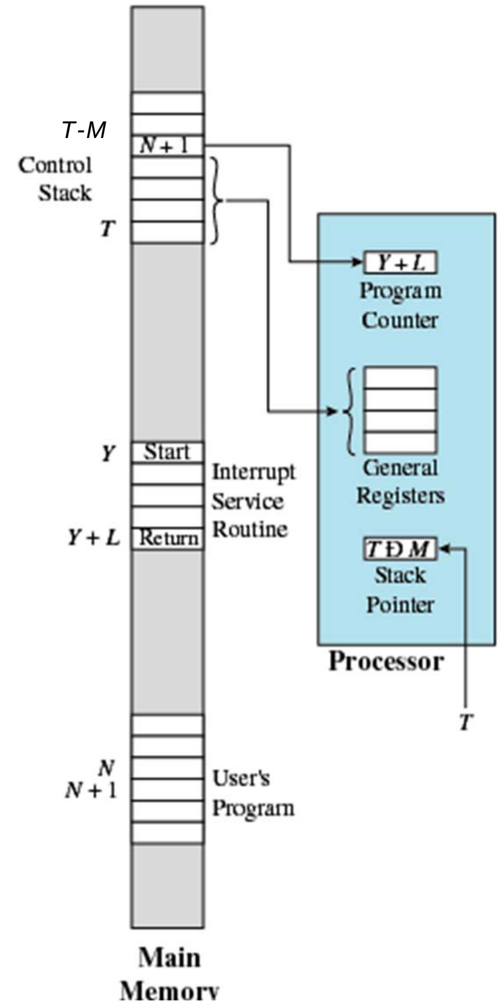
Simple Interrupt Processing



Changes in Memory and Registers for an Interrupt



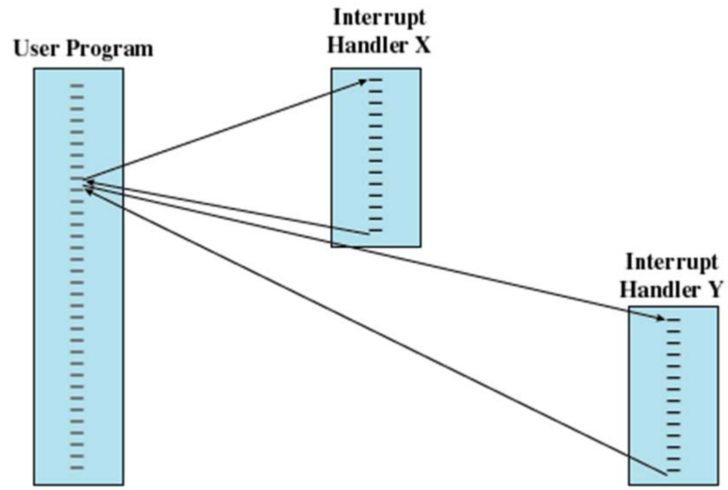
Interrupt occurs after instruction at location N



Return from Interrupt

Multiple Interrupts

- What if interrupt occurs while another interrupt is being serviced?
- One at a time:
  - Disable others -> keep them pending
  - Finish
  - Go back to service other pending
  - Problem: Priority, Time-critical processing?



(a) Sequential interrupt processing

Multiple Interrupts

- Enable higher priority interrupts to go before others

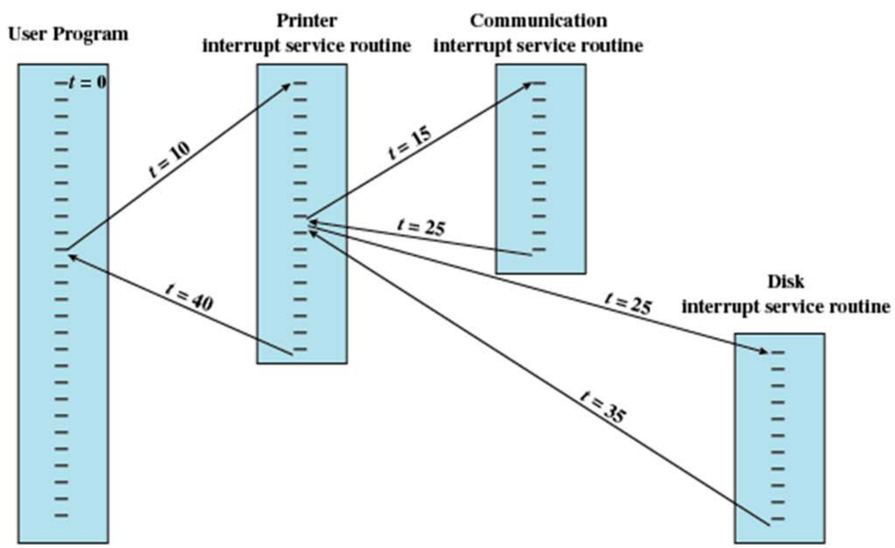
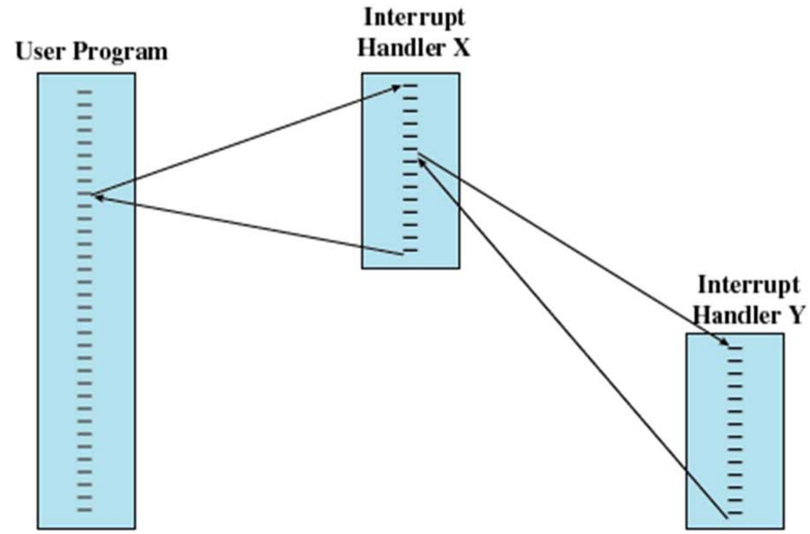


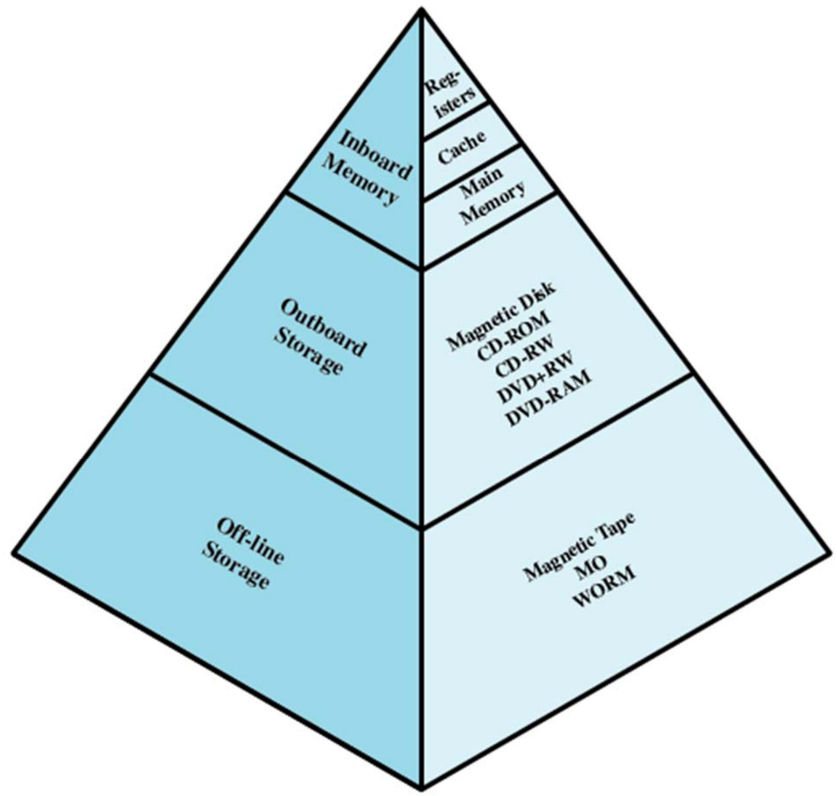
Figure 1.13 Example Time Sequence of Multiple Interrupts



(b) Nested interrupt processing

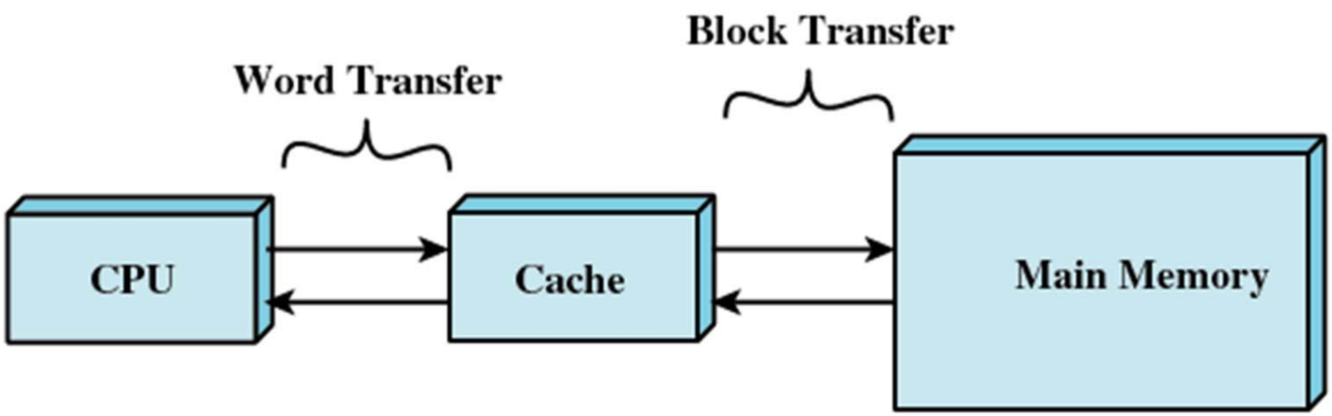
Memory Hierarchy

- Three characteristics of memory
  - Capacity
  - Access Time
  - Cost
- Relationships:
  - Faster access time, greater cost per bit
  - Greater capacity, smaller cost/bit
  - Greater capacity, slower access speed
- Memory hierarchy:
  - Decreasing cost/bit
  - Increasing capacity
  - Increasing access time
  - Decreasing frequency of access of the memory by the processor
    - Locality of reference



Cache Memory

- Invisible to operating system
- Increase the speed of memory
- Processor speed is faster than memory speed
- Exploit the principle of locality:
  - Processor first checks cache
  - If not found in cache, the block of memory containing the needed information is moved



Cache/Main-Memory Structure

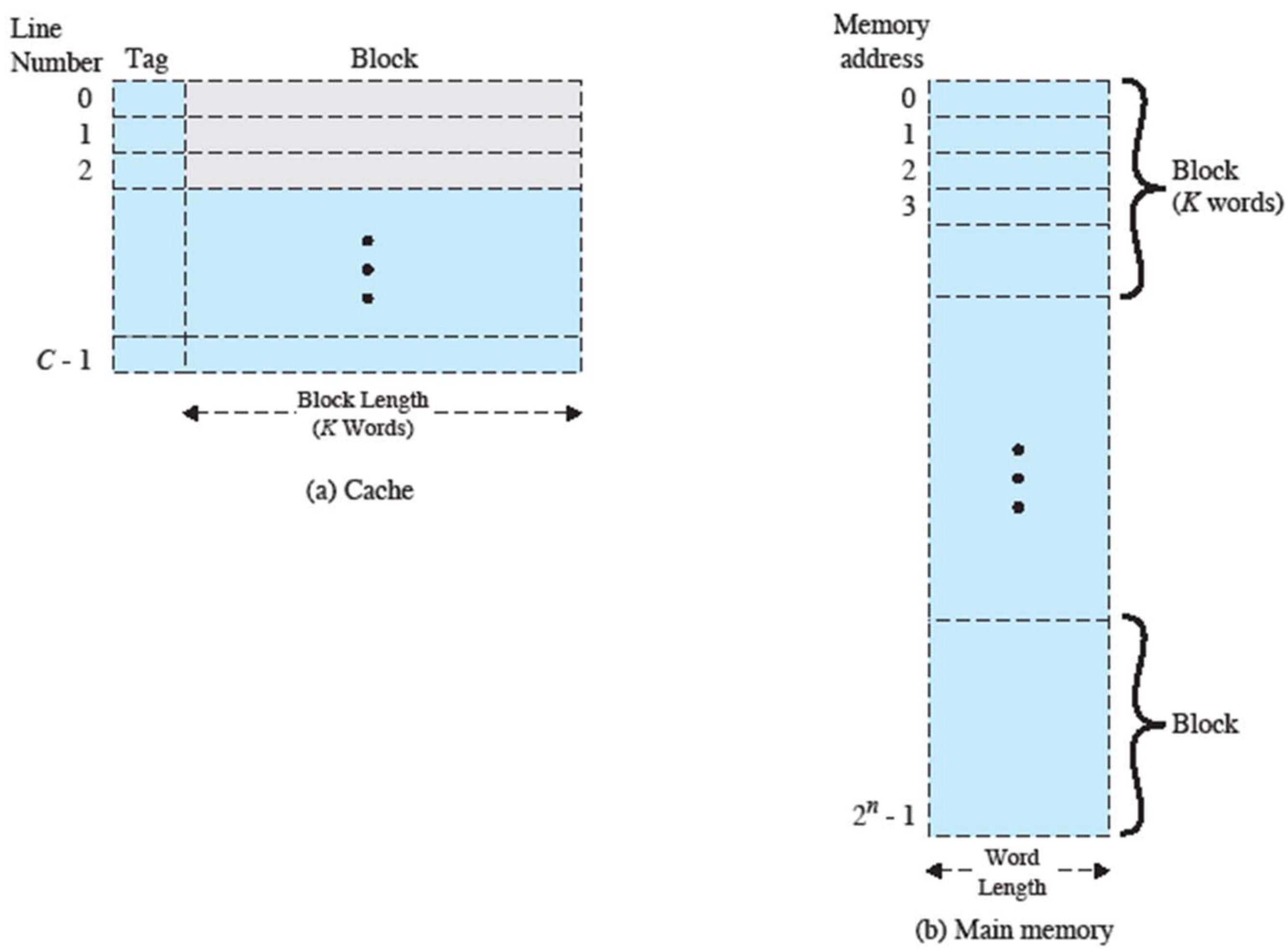


Figure 1.17 Cache/Main-Memory Structure

Cache Read Operation

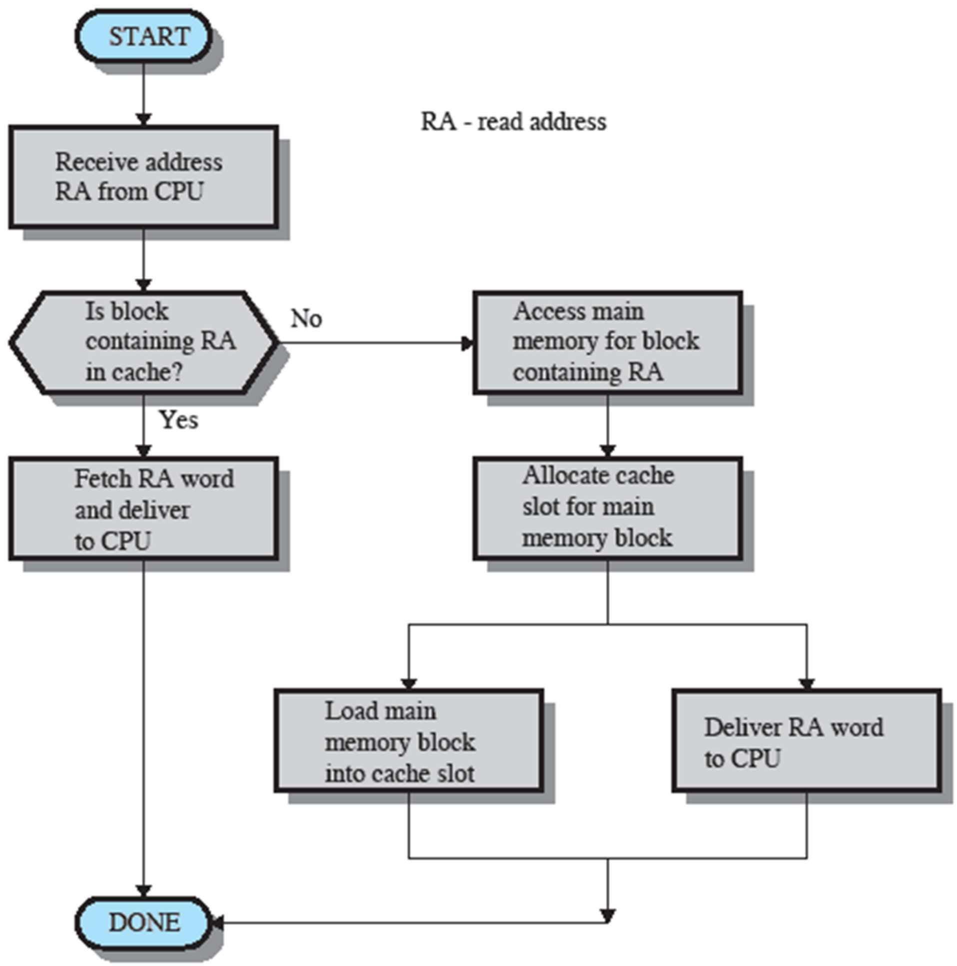


Figure 1.18 Cache Read Operation

Cache Design Issues

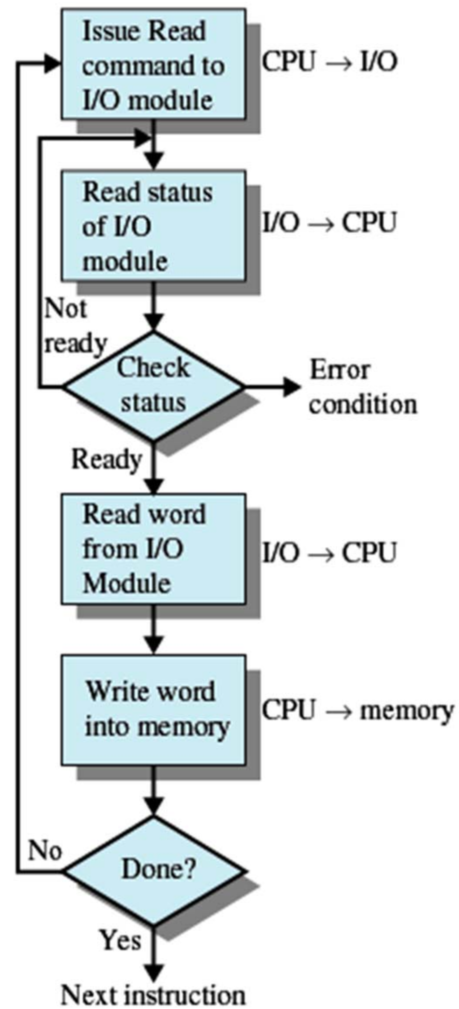
- Cache size
  - Small caches have significant impact on performance
- Block size
  - The unit of data exchanged between cache and main memory
  - Larger block size means more hits
  - But too large reduces chance of reuse.
- Mapping function: Determines which cache location the block will occupy
  - Two constraints:
    - When one block read in, another may need replaced
    - Complexity of mapping function increases circuitry costs for searching
- Replacement algorithm
  - Chooses which block to replace when a new block is to be loaded into the cache.
  - Ideally replacing a block that isn't likely to be needed again
- Impossible to guarantee
- Effective strategy is to replace a block that has been used less than others
  - Least Recently Used (LRU)
- Write policy: Dictates when the memory write operation takes place
- Can occur every time the block is updated
- Can occur when the block is replaced
  - Minimize write operations
  - Leave main memory in an obsolete state

I/O Techniques

- When the processor encounters an instruction relating to I/O,
  - it executes that instruction by issuing a command to the appropriate I/O module.
- Three techniques are possible for I/O operations:
  - Programmed I/O
  - Interrupt-driven I/O
  - Direct memory access (DMA)

Programmed I/O

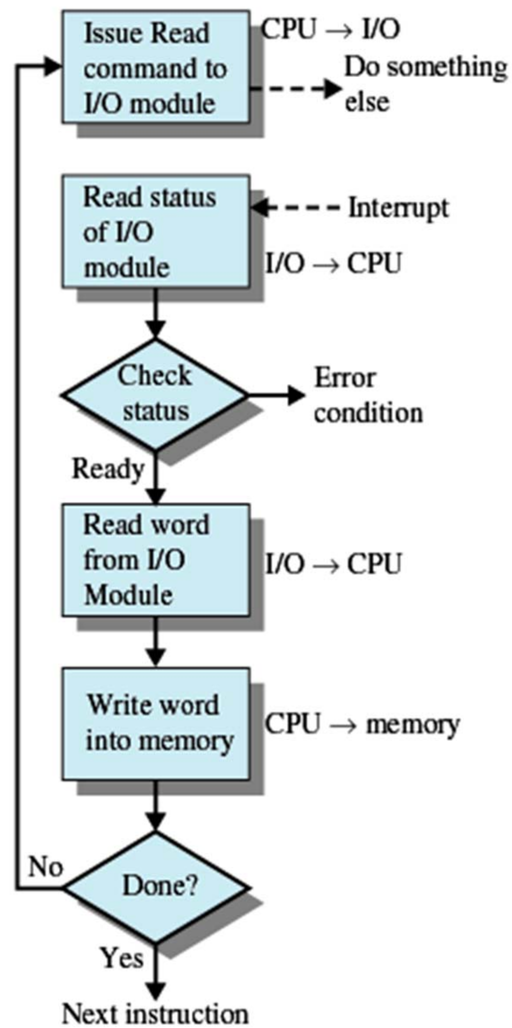
- I/O module performs the action, not the processor
- Sets appropriate bits in the I/O status register
- No interrupts occur
- Processor checks status until operation is complete
- Cons:
  - Performance as CPU must keep checking



(a) I

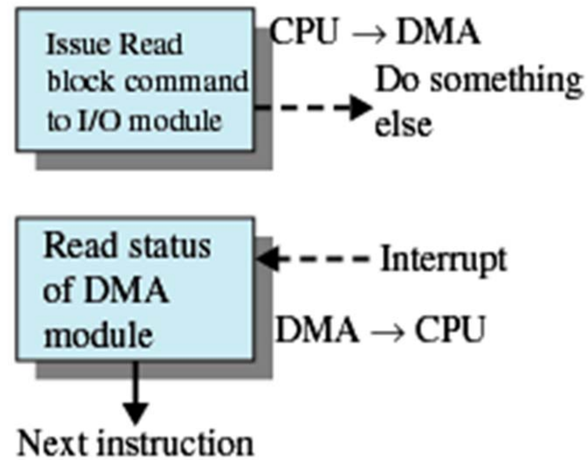
Interrupt-Driven I/O

- Processor is interrupted when I/O module ready to exchange data
- Processor saves context of program executing and begins executing interrupt-handler
- No needless waiting
- Consumes a lot of processor time because every word read or written passes through the processor



Direct Memory Access (DMA)

- I/O exchanges occur directly with memory
- Processor grants I/O module authority to read from or write to memory
- Relieves the processor responsibility for the exchange
- Transfers a block of data directly to or from memory
- An interrupt is sent when the transfer is complete
- Processor continues with other work



Architectural Support for OS

- Architectural support can vastly simplify (or complicate!) 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
    - Apollo workstation used two CPUs as a band-aid for non-restartable instructions!
  - Until very recently, Intel-based PCs still lacked support for 64-bit addressing (which has been available for a decade on other platforms: MIPS, Alpha, IBM, etc…)
    - changing rapidly due to 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)
  - privileged instructions
  - system calls (and software interrupts)
  - virtualization architectures
    - Intel: http://www.intel.com/technology/itj/2006/v10i3/1-hardware/7-architecture-usage.htm
    - AMD: http://sites.amd.com/us/business/it-solutions/usage-models/virtualization/Pages/amd-v.aspx

Privileged/Protected instructions

- some instructions are restricted to the OS
  - known as protected or privileged instructions
- e.g., only the OS can:
  - directly access I/O devices (disks, network cards)
    - why?
  - manipulate memory state management
    - page table pointers, TLB loads, etc.
    - why?
  - manipulate special ‘mode bits’
    - interrupt priority level
    - why?

OS protection

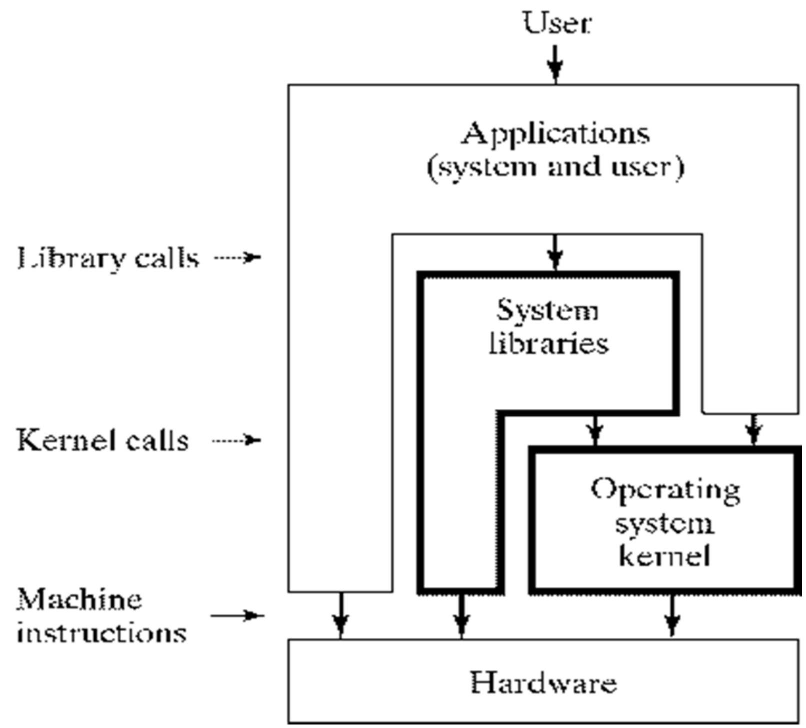
- So how does the processor know if a privileged instruction should be executed?
  - the architecture must support at least two modes of operation: kernel mode and user mode
    - VAX, x86 support 4 protection modes
  - mode is set by status bit in a protected processor register
    - user programs execute in user mode
    - OS executes in kernel (privileged) mode (OS == kernel)
- Privileged instructions can only be executed in kernel (privileged) mode
  - what happens if code running in user mode attempts to execute a privileged instruction?

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, get 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

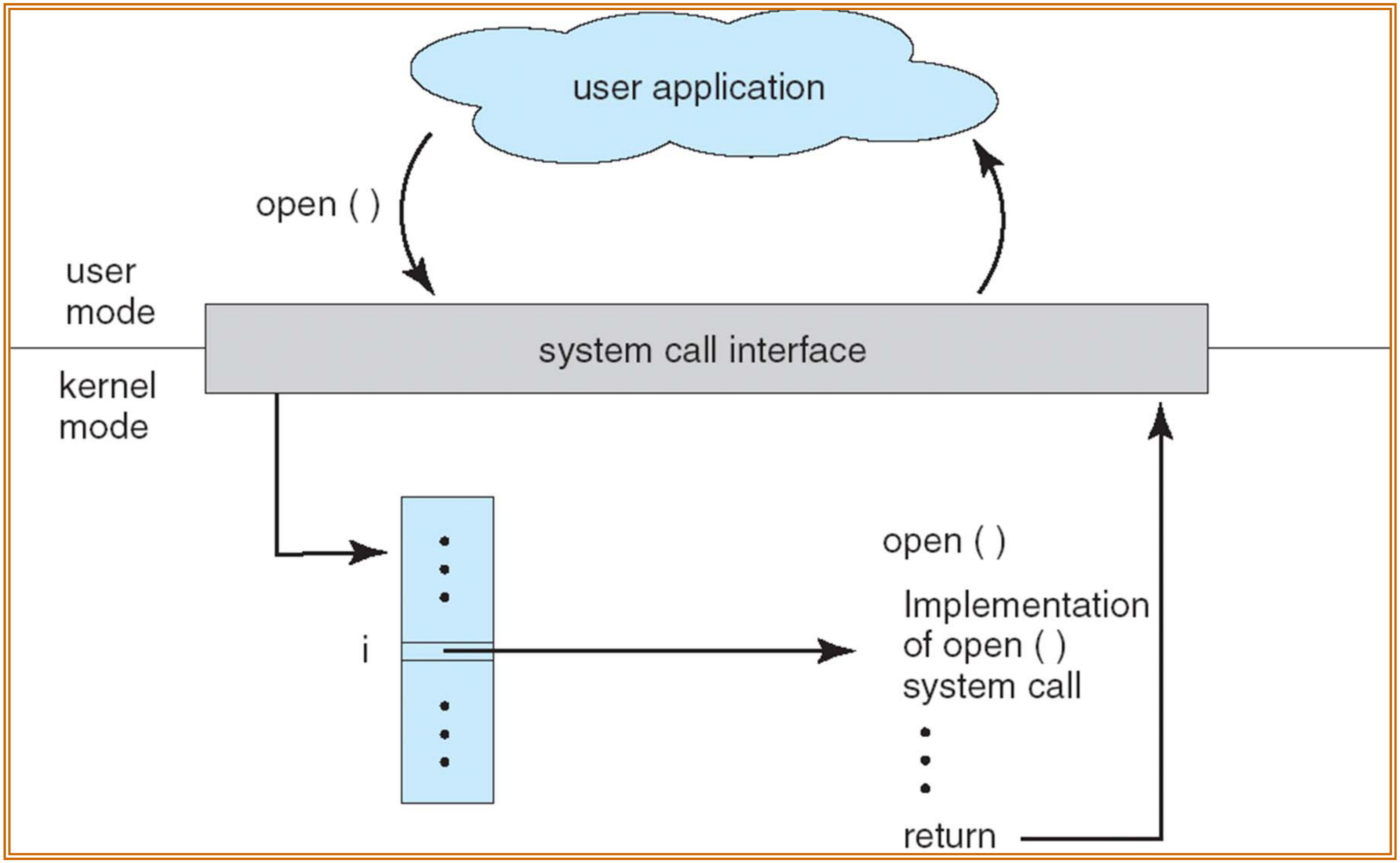
Organization of OSs

- Programming Interface



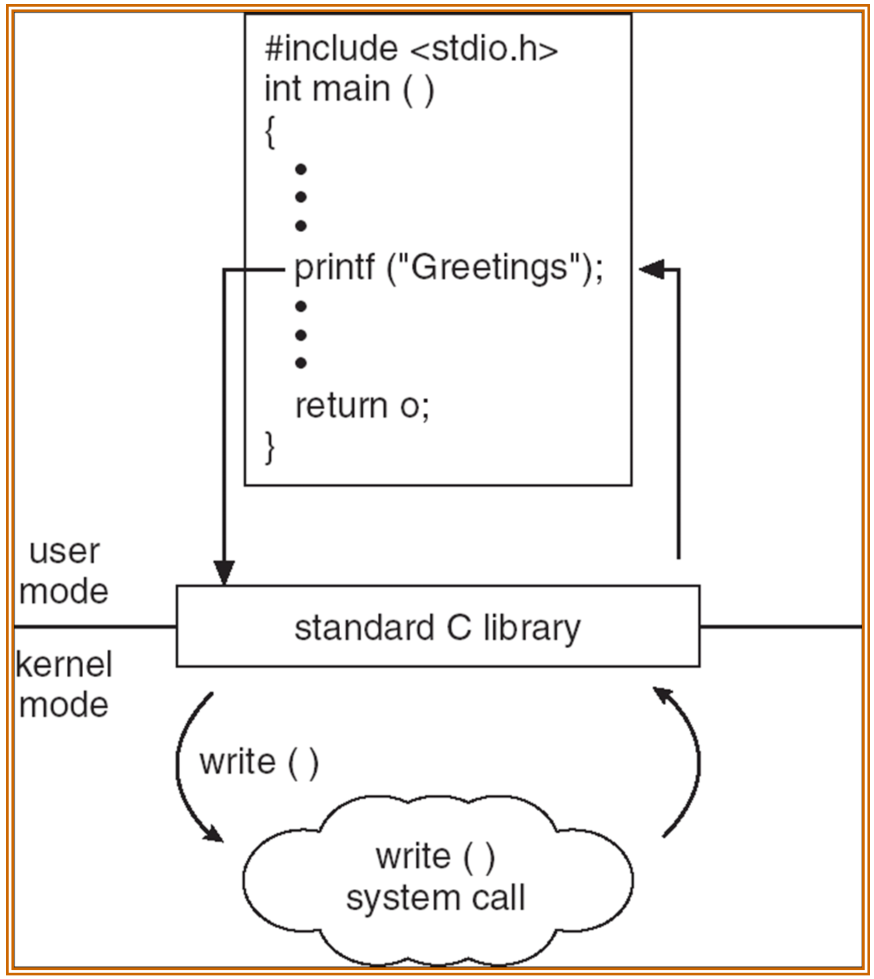
- Invoking system services
  - Library call (nonprivileged)
  - Kernel call (privileged)

API – System Call – OS Relationship



Standard C Library Example

- C program invoking printf() library call, which calls write() system call



System Calls

- Programming interface to the services provided by the OS
- Typically written in a high-level language (C or C++)
- Mostly accessed via a high-level **Application Program Interface (API)**
- Three most common APIs are Win32 API for Windows, POSIX API for POSIX-based systems (including virtually all versions of UNIX, Linux, and Mac OS X), and Java API for the Java virtual machine (JVM)
- Why use APIs rather than system calls?

Types of System Calls

- Process control
- Device management
- Communications
- File management
- Information maintenance

Process management

Call	Description
pid = fork( )	Create a child process identical to the parent
pid = waitpid(pid, &statloc, options)	Wait for a child to terminate
s = execve(name, argv, environp)	Replace a process' core image
exit(status)	Terminate process execution and return status

File management

Call	Description
fd = open(file, how, ...)	Open a file for reading, writing or both
s = close(fd)	Close an open file
n = read(fd, buffer, nbytes)	Read data from a file into a buffer
n = write(fd, buffer, nbytes)	Write data from a buffer into a file
position = lseek(fd, offset, whence)	Move the file pointer
s = stat(name, &buf)	Get a file's status information

Some System Calls

Directory and file system management

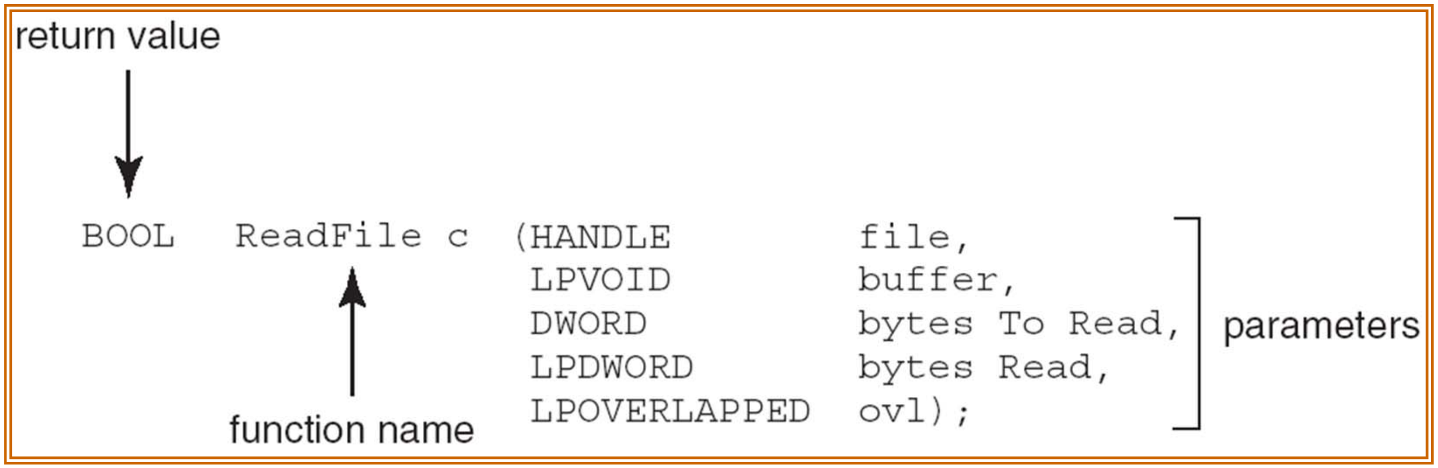
Call	Description
s = mkdir(name, mode)	Create a new directory
s = rmdir(name)	Remove an empty directory
s = link(name1, name2)	Create a new entry, name2, pointing to name1
s = unlink(name)	Remove a directory entry
s = mount(special, name, flag)	Mount a file system
s = umount(special)	Unmount a file system

Miscellaneous

Call	Description
s = chdir(dirname)	Change the working directory
s = chmod(name, mode)	Change a file's protection bits
s = kill(pid, signal)	Send a signal to a process
seconds = time(&seconds)	Get the elapsed time since Jan. 1, 1970

Example of Standard API

- Consider the ReadFile() function in the Win32 API—a function for reading from a file



- A description of the parameters passed to ReadFile()
  - HANDLE file—the file to be read
  - LPVOID buffer—a buffer where the data will be read into and written from
  - DWORD bytesToRead—the number of bytes to be read into the buffer
  - LPDWORD bytesRead—the number of bytes read during the last read
  - LPOVERLAPPED ovl—indicates if overlapped I/O is being used

System Call Implementation

- Typically, a number associated with each system call
  - System-call interface maintains a table indexed according to these numbers
- The system call interface invokes intended system call in OS kernel and returns status of the system call and any return values
- The caller need know nothing about how the system call is implemented
  - Just needs to obey API and understand what OS will do as a result call
  - Most details of OS interface hidden from programmer by API
    - Managed by run-time support library (set of functions built into libraries included with compiler)

Invoking System Services

application issues call to lib_func(params)		body of lib_func()
push params on stack branch to lib_func() body	→	
		perform service return from function
pop stack pointer	←	

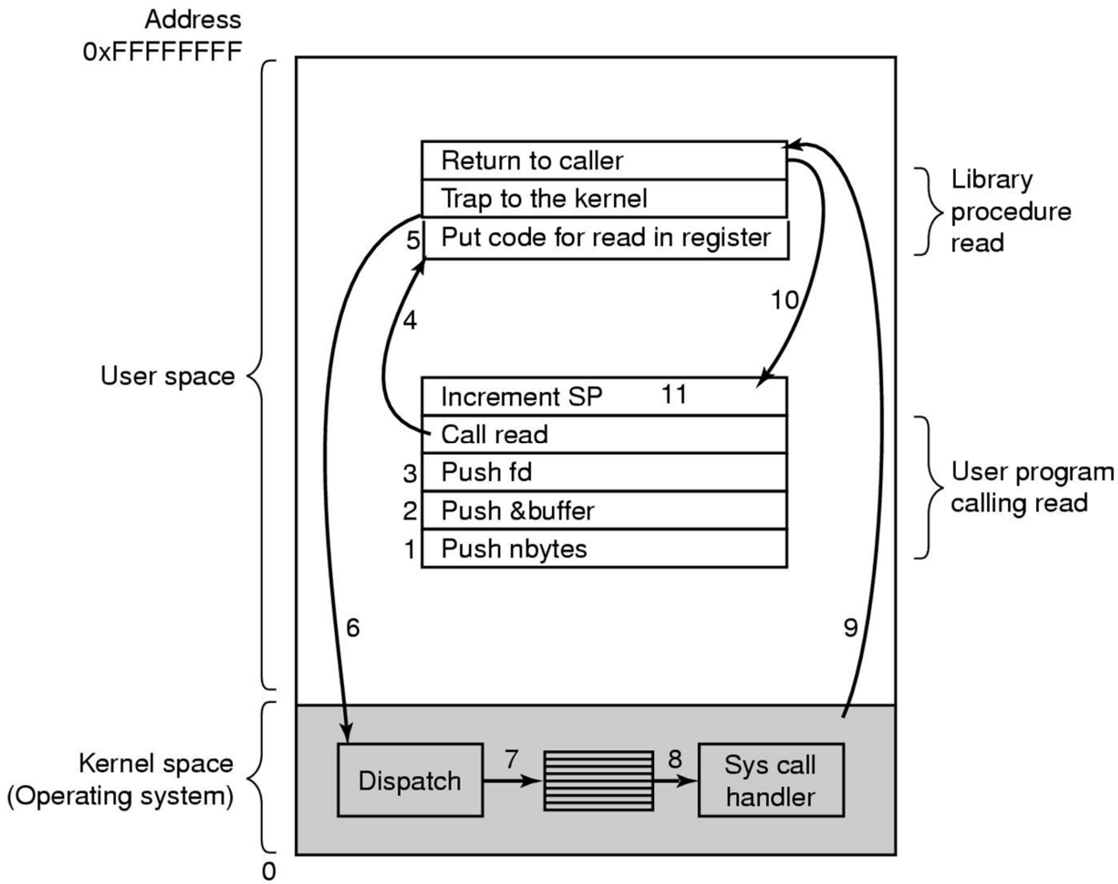
(a)

application issues call to kern_func(params)		body of kern_func()		kernel code
push params on stack branch to kern_func() body	→			
		set up regs for SVC SVC	→	
				perform service set nonprivileged mode return from SVC
		return from function	←	
pop stack pointer	←			

(b)

Figure 1-10

Steps in Making a System Call



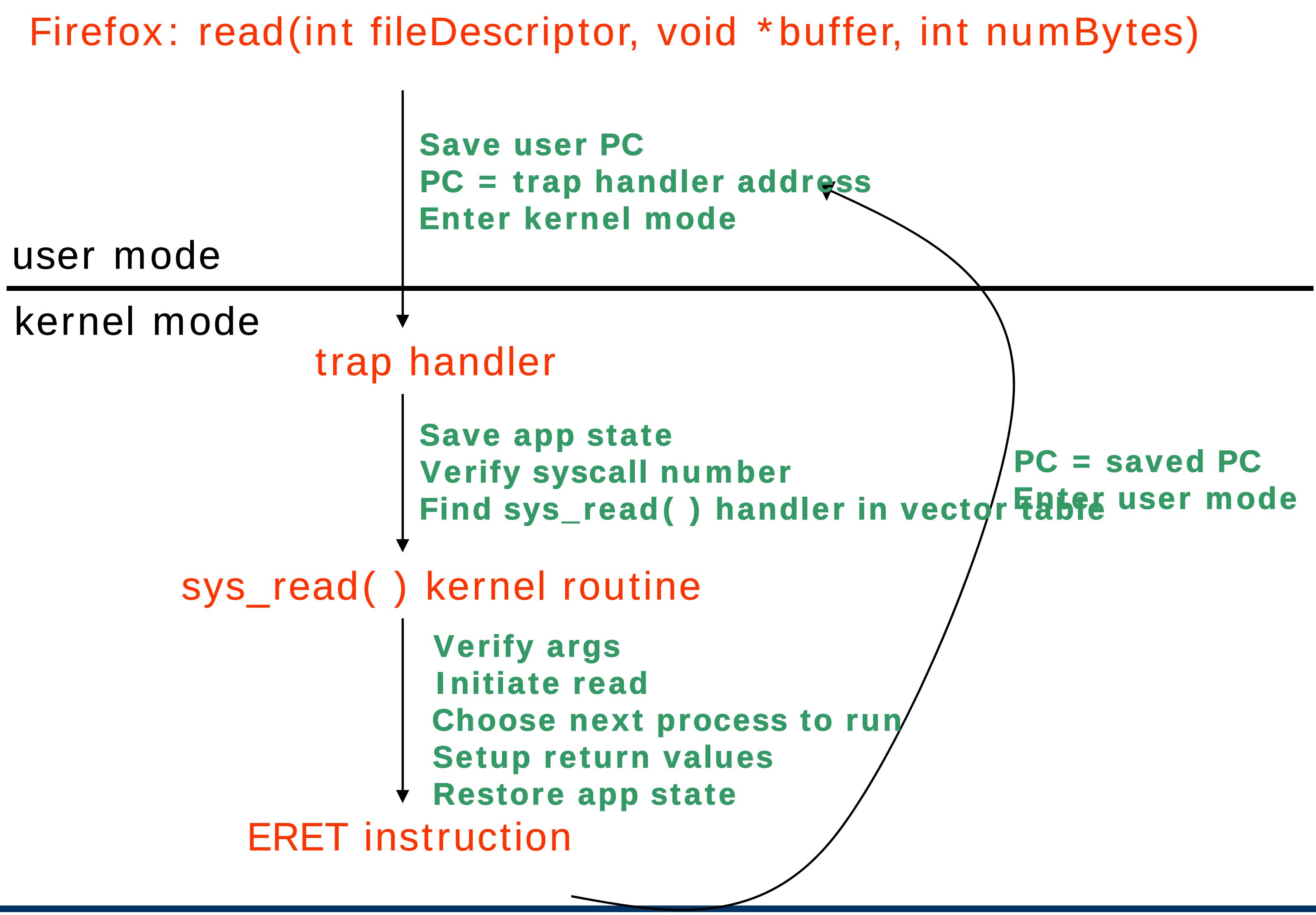
Steps in making the system call:

read (fd, buffer, nbytes)

System Call

- The syscall instruction atomically:
  - Saves the current PC
  - Sets the execution mode to privileged
  - Sets the PC to a handler address
- With that, it's a lot like a local 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

A kernel crossing illustrated

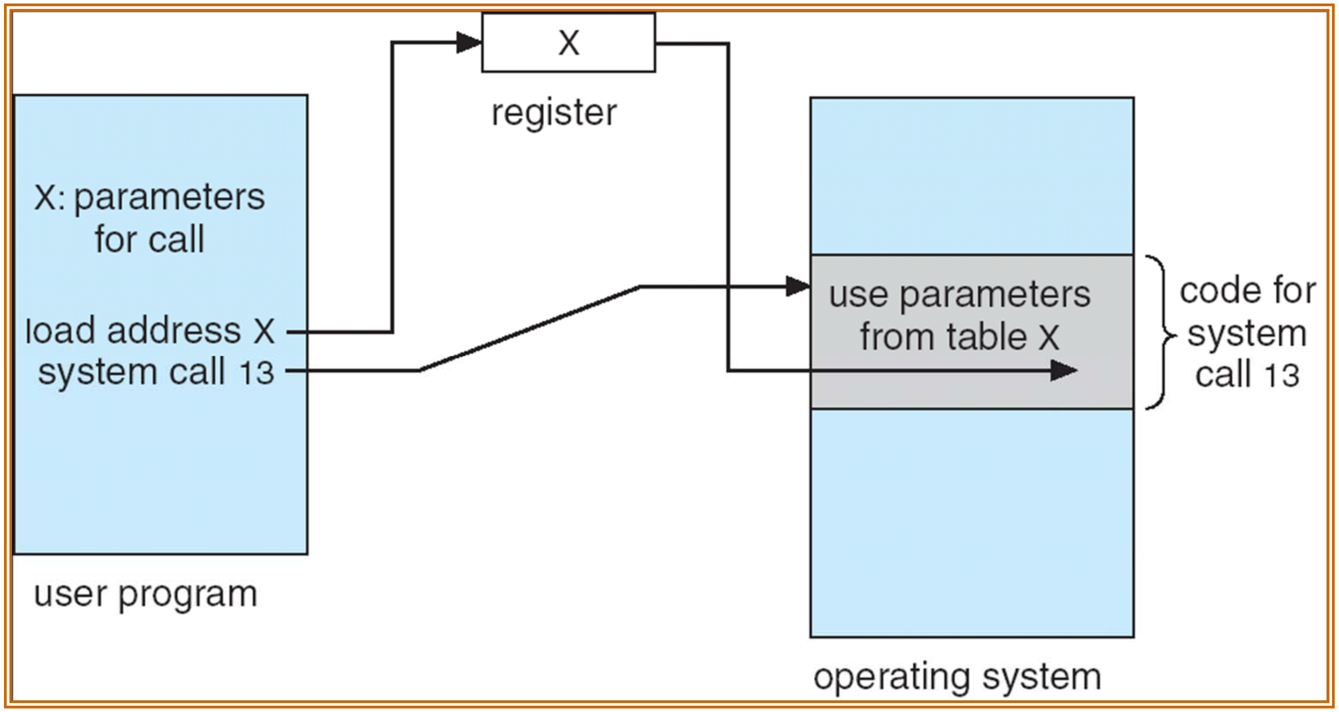


System Call Parameter Passing

- Three general methods used to pass parameters to the OS
- 1. Simplest: pass the parameters in *registers*
  - In some cases, may be more parameters than registers
- 2. Parameters placed, or *pushed,* onto the *stack* by the program and *popped* off the stack by the operating system

System Call Parameter Passing

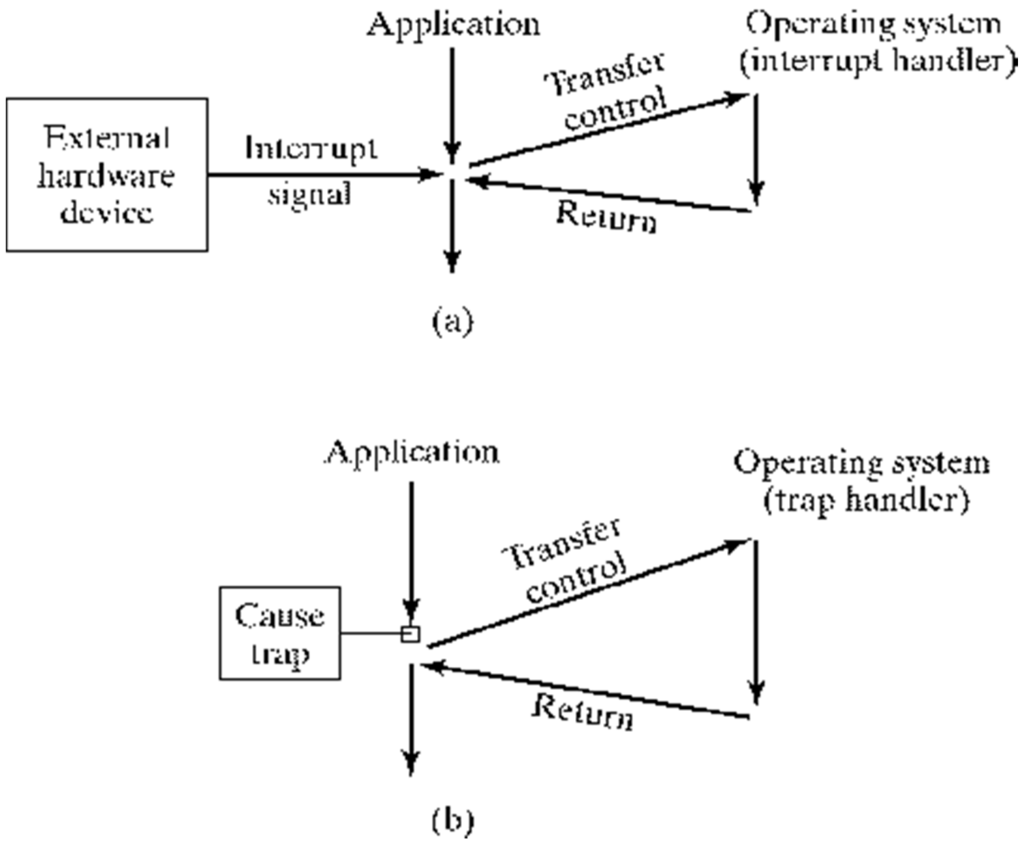
- 3. Parameters stored in a *block,* or table, in memory, and address of block passed as a parameter in a register
  - This approach taken by Linux and Solaris



System call issues

- What would be wrong if a syscall worked like a regular subroutine call, with the caller specifying the next PC?
- What would happen if kernel didn't save state?
- Why must the kernel verify arguments?
- How can you reference kernel objects as arguments to or results from system calls?

Principles of Interupts and Traps

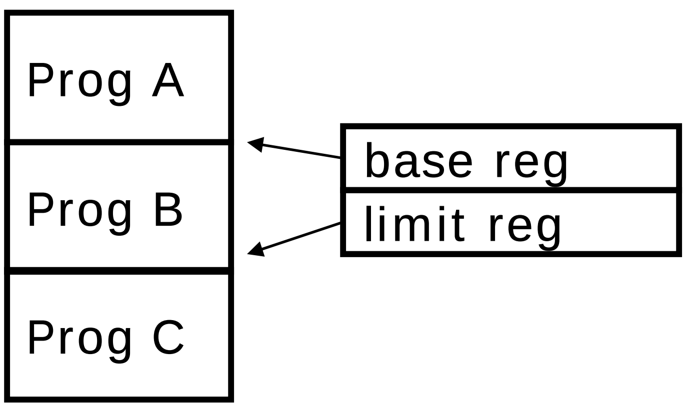


Exception Handling and Protection

- *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, …

Memory protection

- OS must protect user programs from each other
  - maliciousness, ineptitude
- OS must also protect itself from user programs
  - integrity and security
  - what about protecting user programs from OS?
- Simplest scheme: base and limit registers
  - are these protected?



base and limit registers are loaded by OS before starting program

More sophisticated memory protection

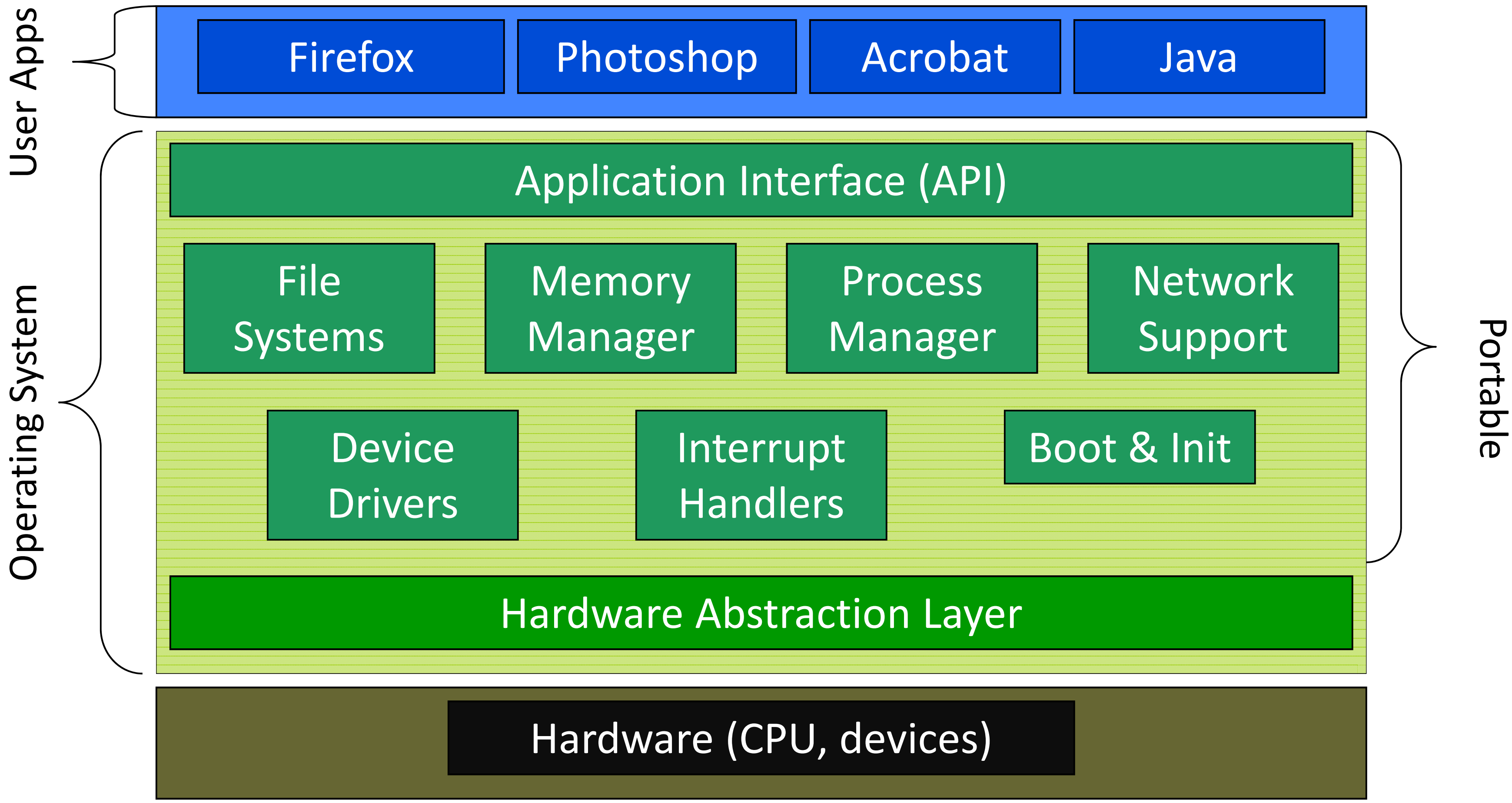
- coming later in the course
- paging, segmentation, virtual memory
  - page tables, page table pointers
  - translation lookaside buffers (TLBs)
  - page fault handling

Part II: OS Structure

Structure of Operating Systems

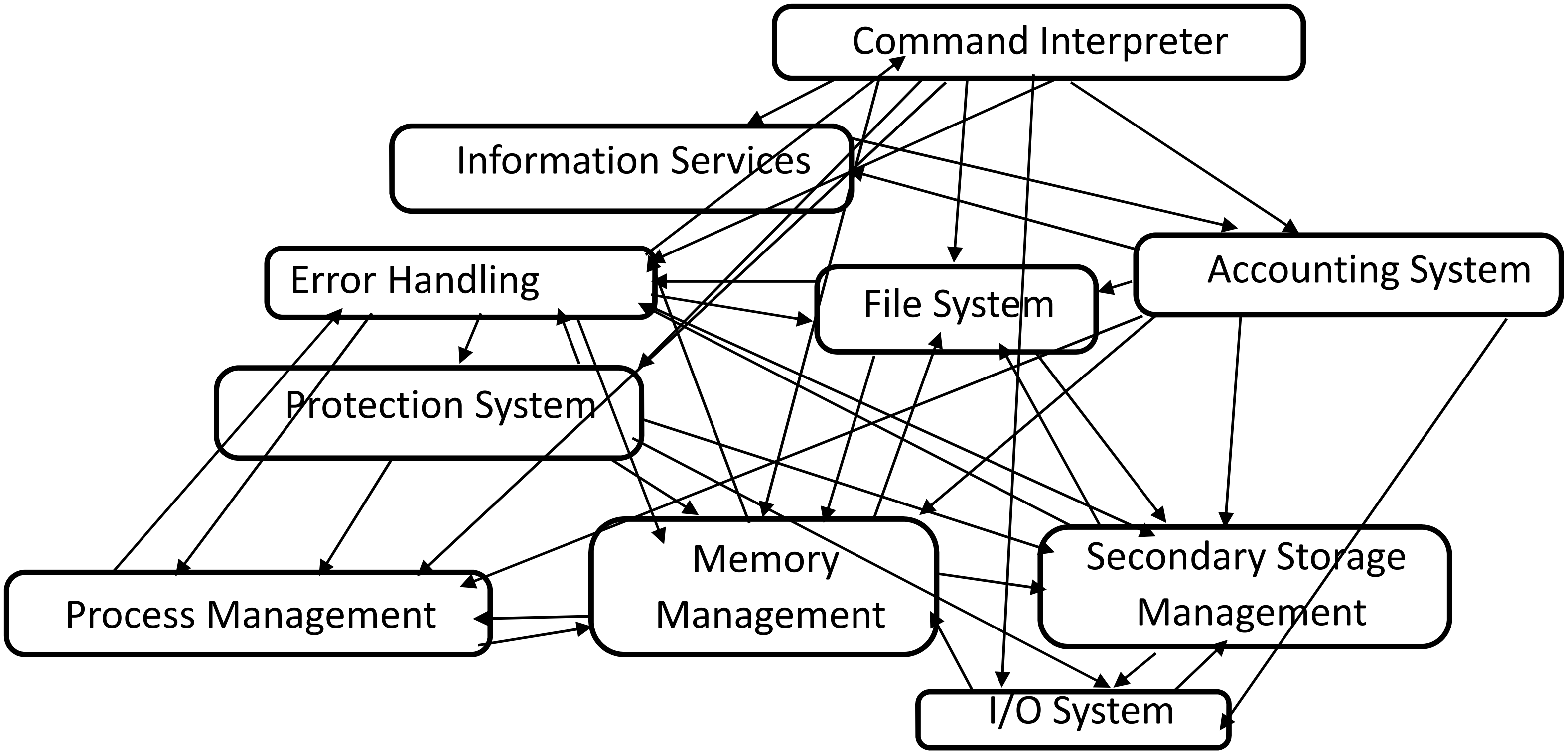
- What are components of traditional OSs?
- What are design issues that affect structures of OSs?
- How are components implemented?
- How are they stitched together?
- Major software engineering and design problem
  - Design a large complex system that
    - Is efficient
    - Is reliable
    - Is extensible
    - Is backwards compatible
    - Provides lots of services
- Typical structures
  1. Monolithic
  2. Layered
  3. Micro-Kernel

OS Structure



Source: Gribble, Lazowska, Levy, Zahorjan

Complex Runtime Interaction among OS components



Source: Gribble, Lazowska, Levy, Zahorjan

Components of Operating Systems

- Process Management
- Main Memory Management
- Secondary-Storage Management
- I/O System Management
- File Management
- Protection System
- Networking
- Command-Interpreter System
- Accounting

Process Management

- A *process* =
  - program in execution +
  - Resources: CPU time, memory, files, and I/O devices
  - Privileges
- Supported operations:
  - Process creation and deletion.
  - process suspension and resumption.
  - Provision of mechanisms for:
    - process synchronization
    - process communication

Main-Memory Management

- Memory =
  - Array of bytes
  - Sharing between CPU and I/O devices
  - Volatile
- Supported operations:
  - Keep track of which parts of memory are currently being used and by whom.
  - Decide which processes to load when memory space becomes available.
  - Allocate and deallocate memory space as needed

Secondary-Storage Management

- *Secondary storage* to back up main memory + long term storage
  - Disks
  - Store program and data
- Disk management operations:
  - Free space management
  - Storage allocation
  - Disk scheduling
    - Read
    - Write

I/O System Management

- I/O
  - Input/Output
  - Networking Interface
  - Display
  - Others
- The I/O system consists of:
  - A buffer-caching system
  - A general device-driver interface
  - Drivers for specific hardware devices

File Management

- Information representation:
  - Files
    - Program
    - Data
  - Directory:
    - Organize information Operations for file management:
- Operations supported;
  - File creation and deletion
  - Directory creation and deletion
  - Support of primitives for manipulating files and directories
  - Mapping files onto secondary storage
  - File backup on stable (nonvolatile) storage media

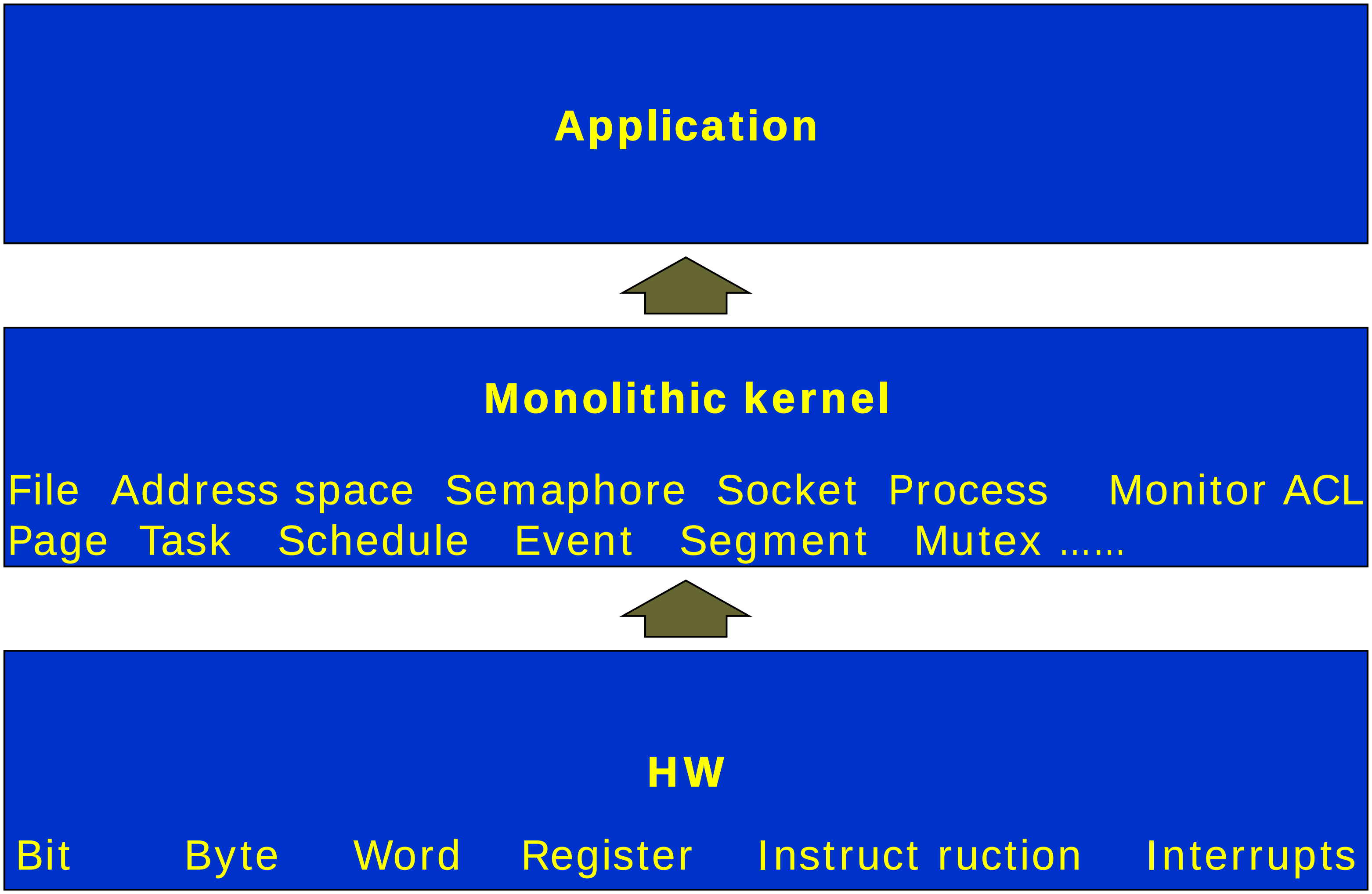
Protection System

- *Protection* refers to a mechanism for controlling access by programs, processes, or users to both system and user resources.
- The protection mechanism must:
  - distinguish between authorized and unauthorized usage.
  - specify the controls to be imposed.
  - provide a means of enforcement.

Organization of Operating Systems

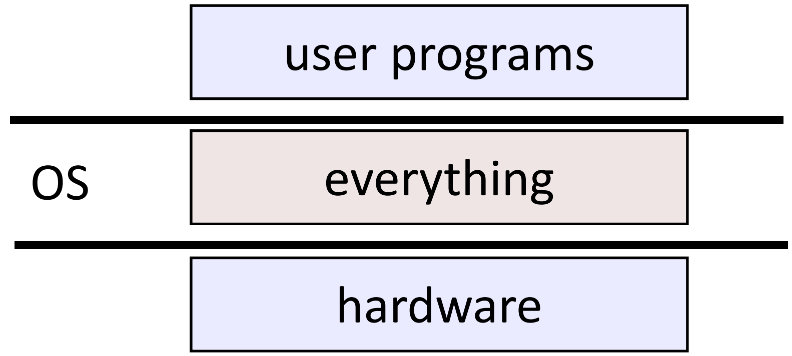
1. Monolithic
2. Layered
3. Micro-Kernel
4. Extensible operating systems

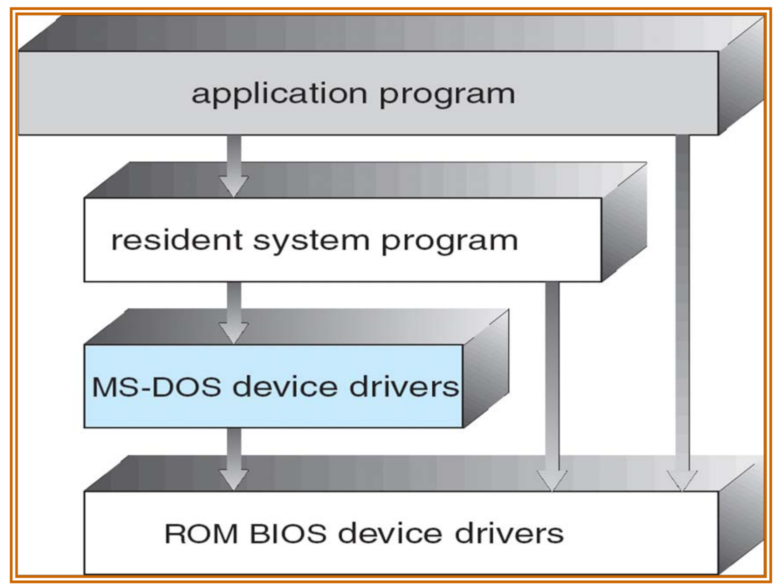
OS structuring: Monolithic kernels



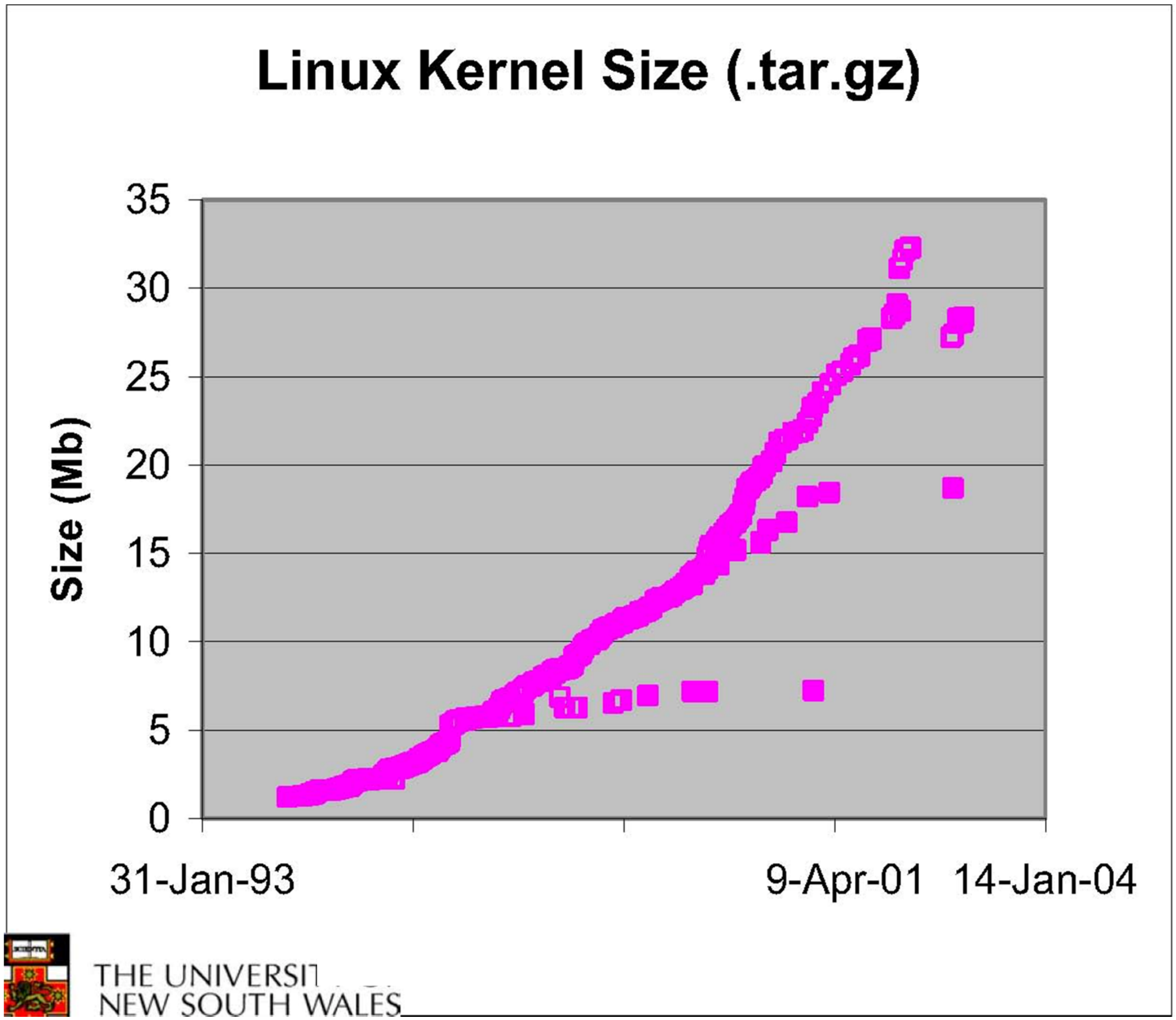
Simple Structure: Monolithic OS

- Traditional OS's: Built as monolithic entity
- Advantage:
  - Efficient: I/O routines can directly write to display and disk drives => more efficient
- Disadvantages:
  - Hard to understand
  - Hard to modify
  - Unreliable: OS vulnerable to malicious or buggy programs
  - Hard to maintain
- MS-DOS –provide the most functionality in the least space: Not divided into modules
  - Although MS-DOS has some structure, its interfaces and levels of functionality are not well separated



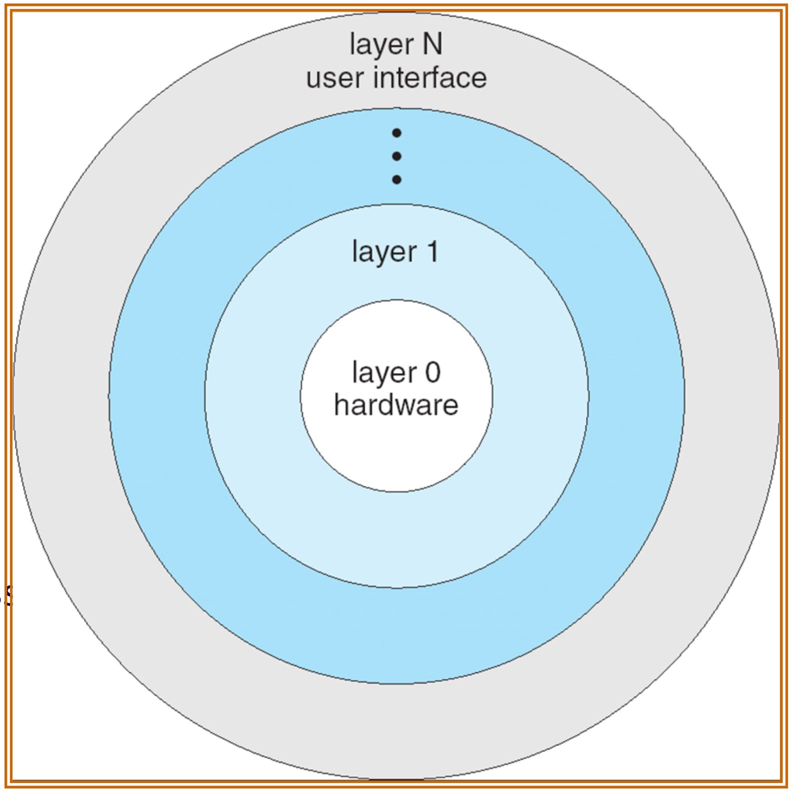


Monolithic OS



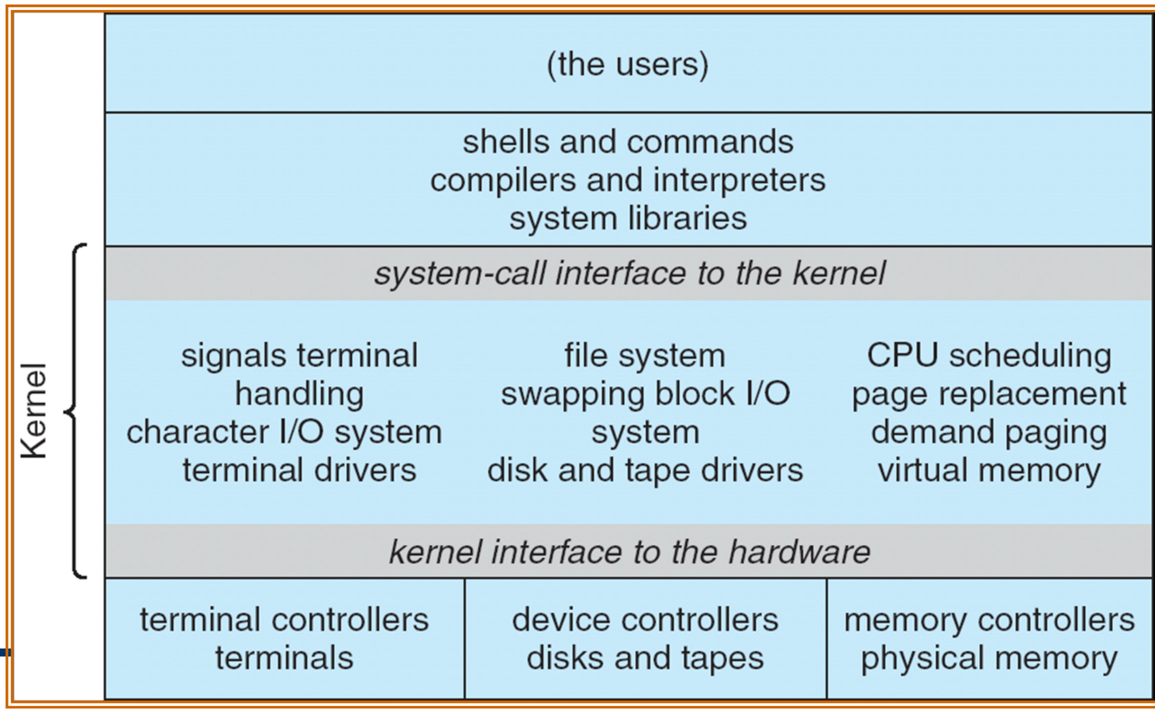
Layered Operating Systems

- OS divided into layers (levels),
- 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



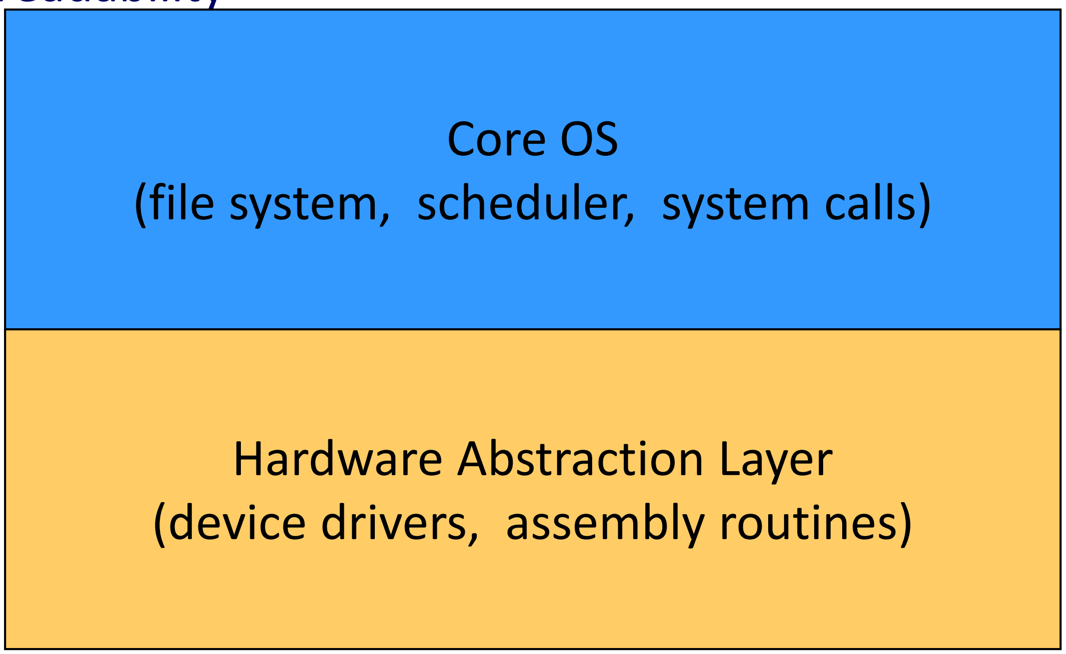
Example: UNIX System Structure

- Limited structuring.
- The UNIX OS two parts:
  - **Systems programs**
  - **The kernel:** Consists of everything below the system-call interface and above the physical hardware
    - Provides the file system, CPU scheduling, memory management, and other operating-system functions; a large number of functions for one level



Example of layering: Hardware Abstraction Layer

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



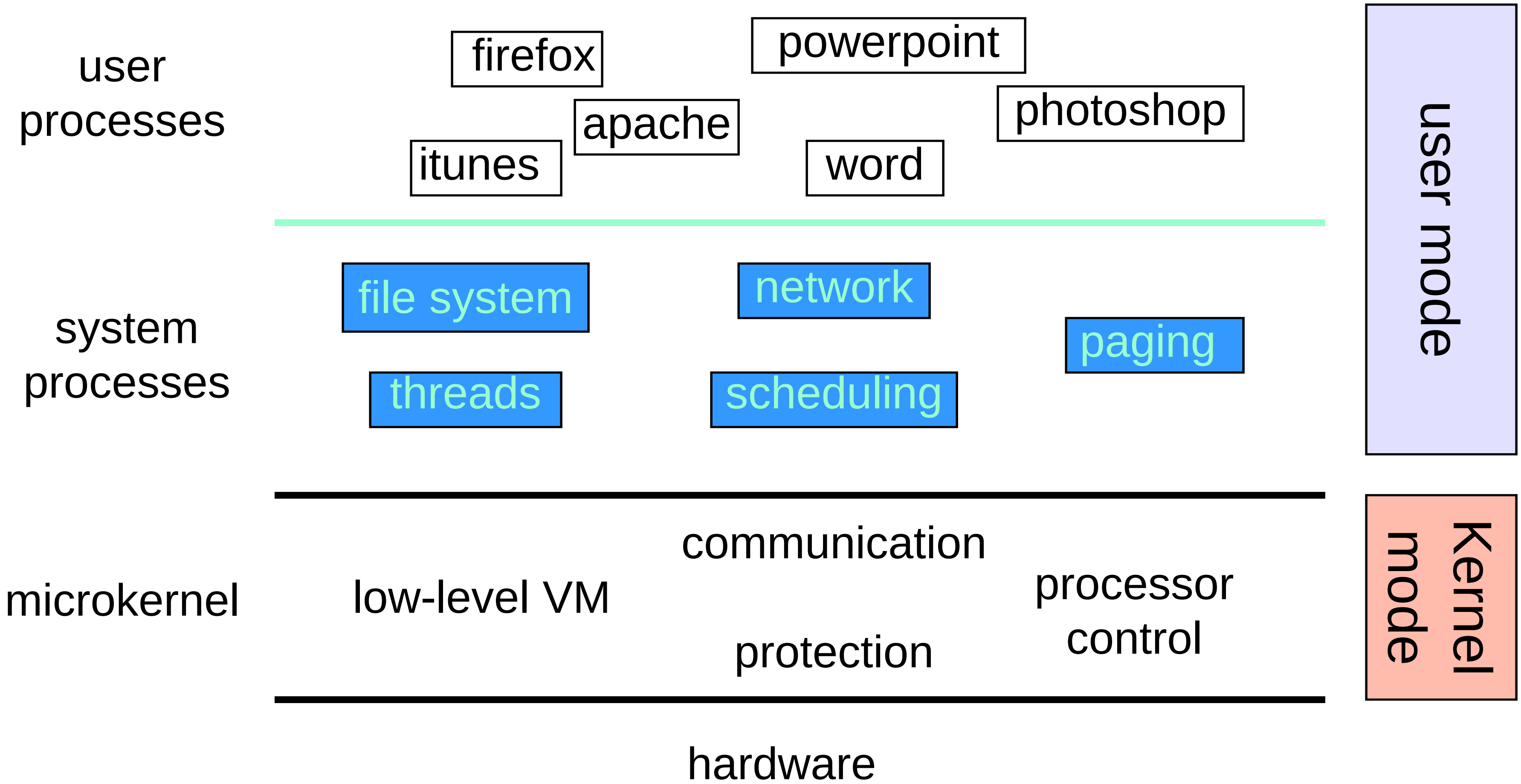
Problems with layered approach

- Imposes hierarchical structure: one-one, unidirectional relationship…
  - 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
  - Overhead of crossing each layer
- Widening range of services and application
  => OS bigger, more complex, slower and more error prone
- Portability problems
  - Does one layering structure translate to similar one on a different architecture?
- Harder to support different OS environments
- Distribution
  => impossible to provide all services from same kernel

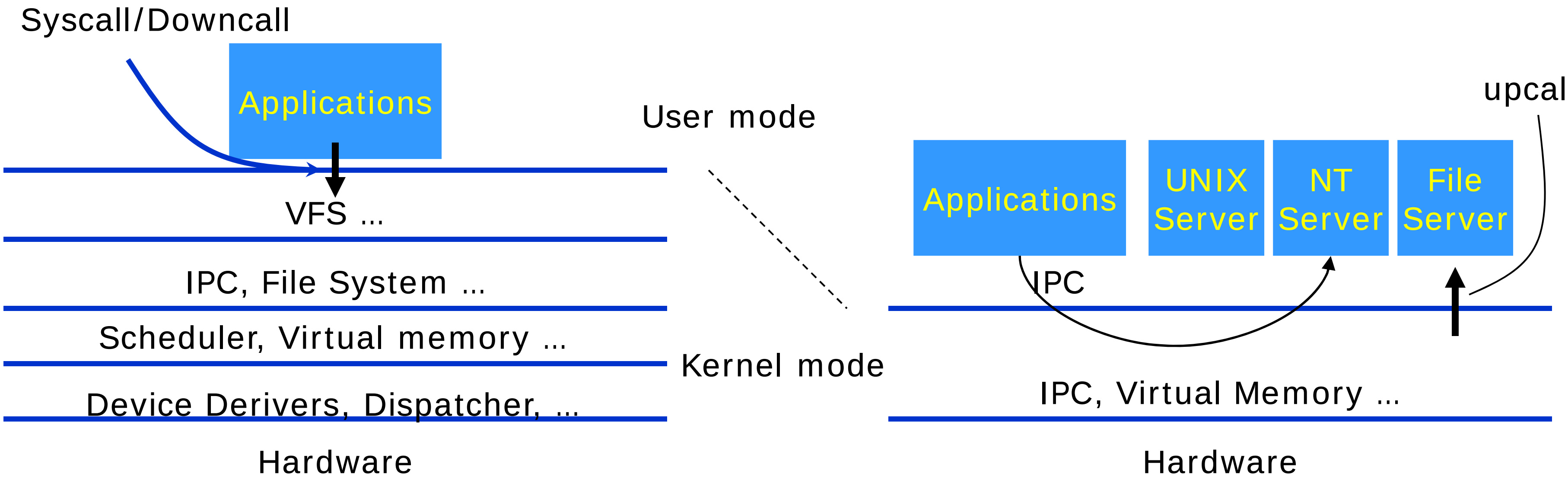
Microkernels

- 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



Microkernels: break up OS

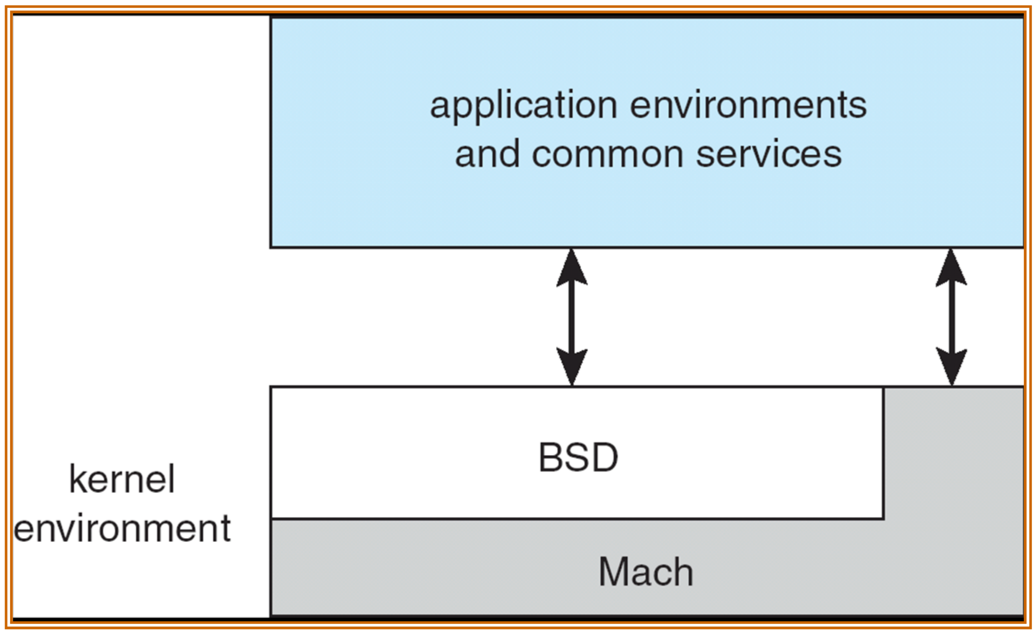


- Kernel: Implement mechanisms
  - Code that must run in supervisory mode
  - Isolate hardware dependencies from higher levels
  - Small and fast
- User-level servers:
  - Hardware independent/portable
  - Provide "OS environment/OS personality"
  - May be invoked from:
    - Application (IPC)
    - Kernel (upcalls)

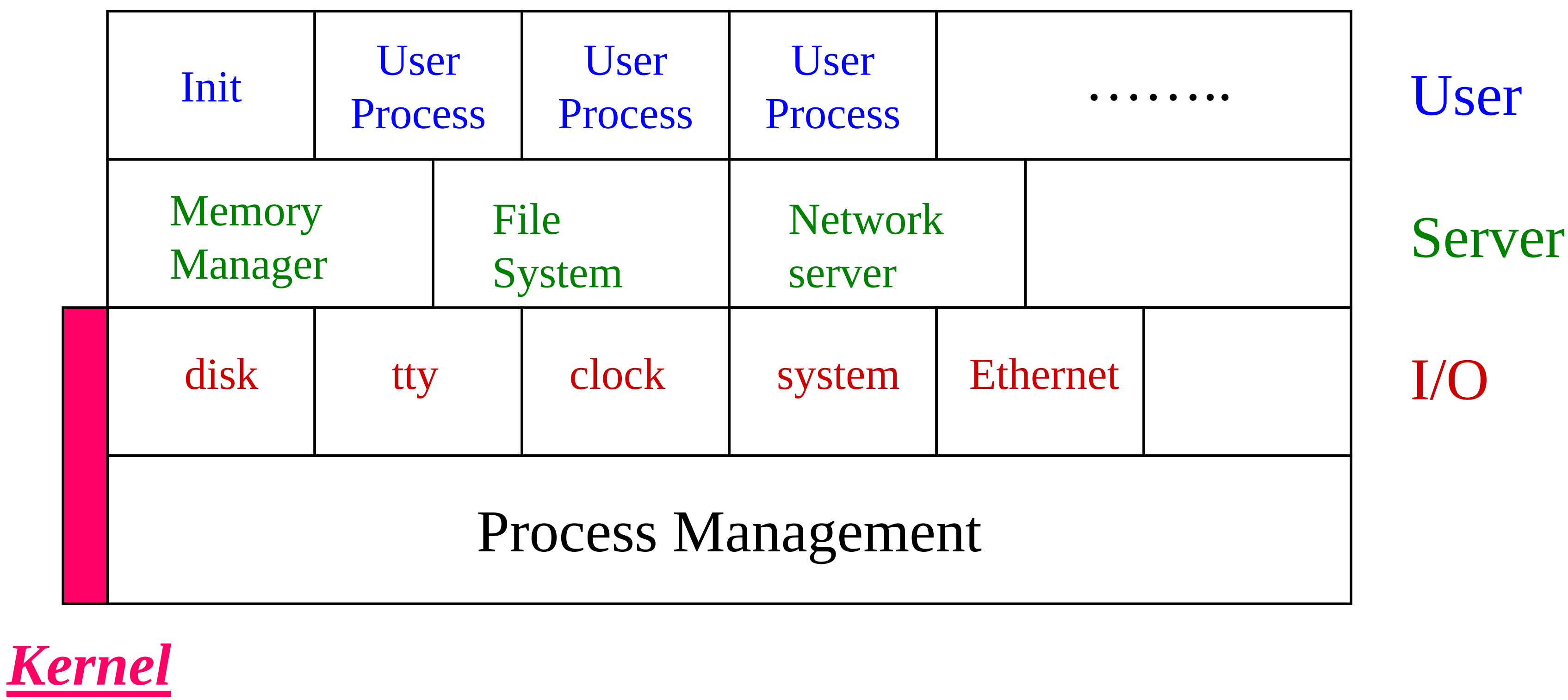
Promise of Microkernels

- Co-existence of different
  - APIs
  - File systems
  - OS Personalities
- Flexibility
- Extensibility
- Simplicity
- Maintainability
- Security
- Safety

Example: Mac OS X Structure



Example: Minix OS Structure



Windows

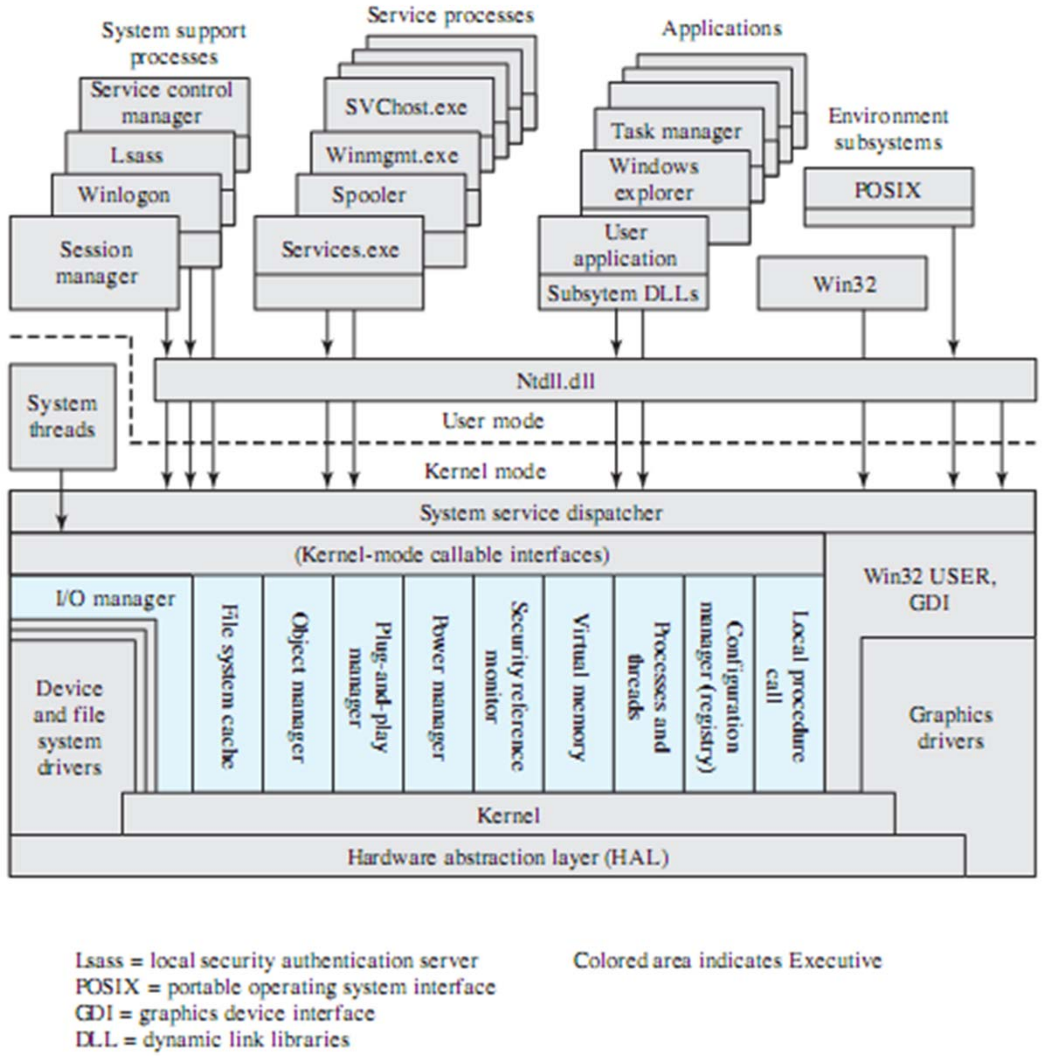


Figure 2.13 Windows and Windows Vista Architecture [RUSS05]

Source: Stallings

Linux Kernel Structure

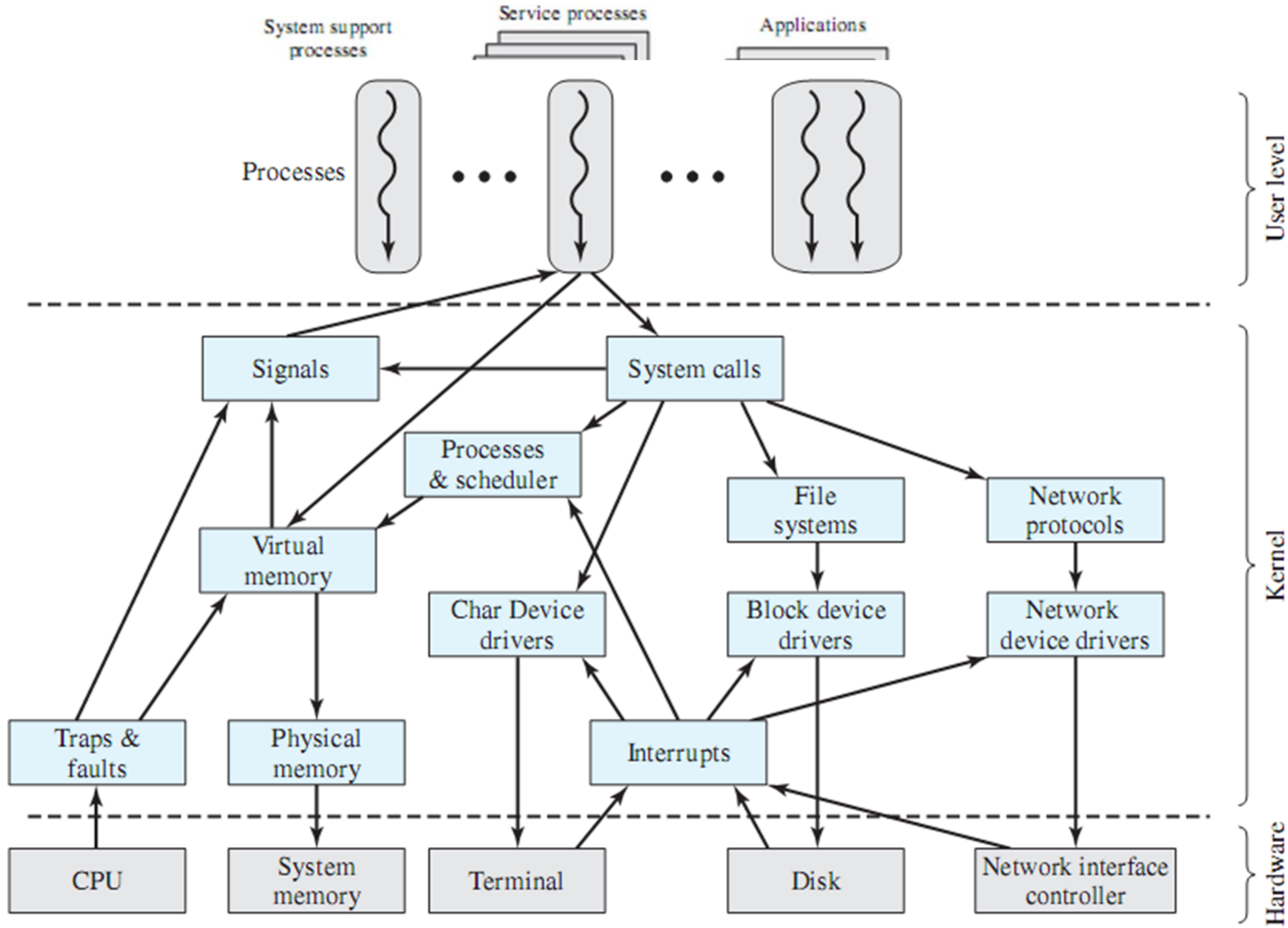


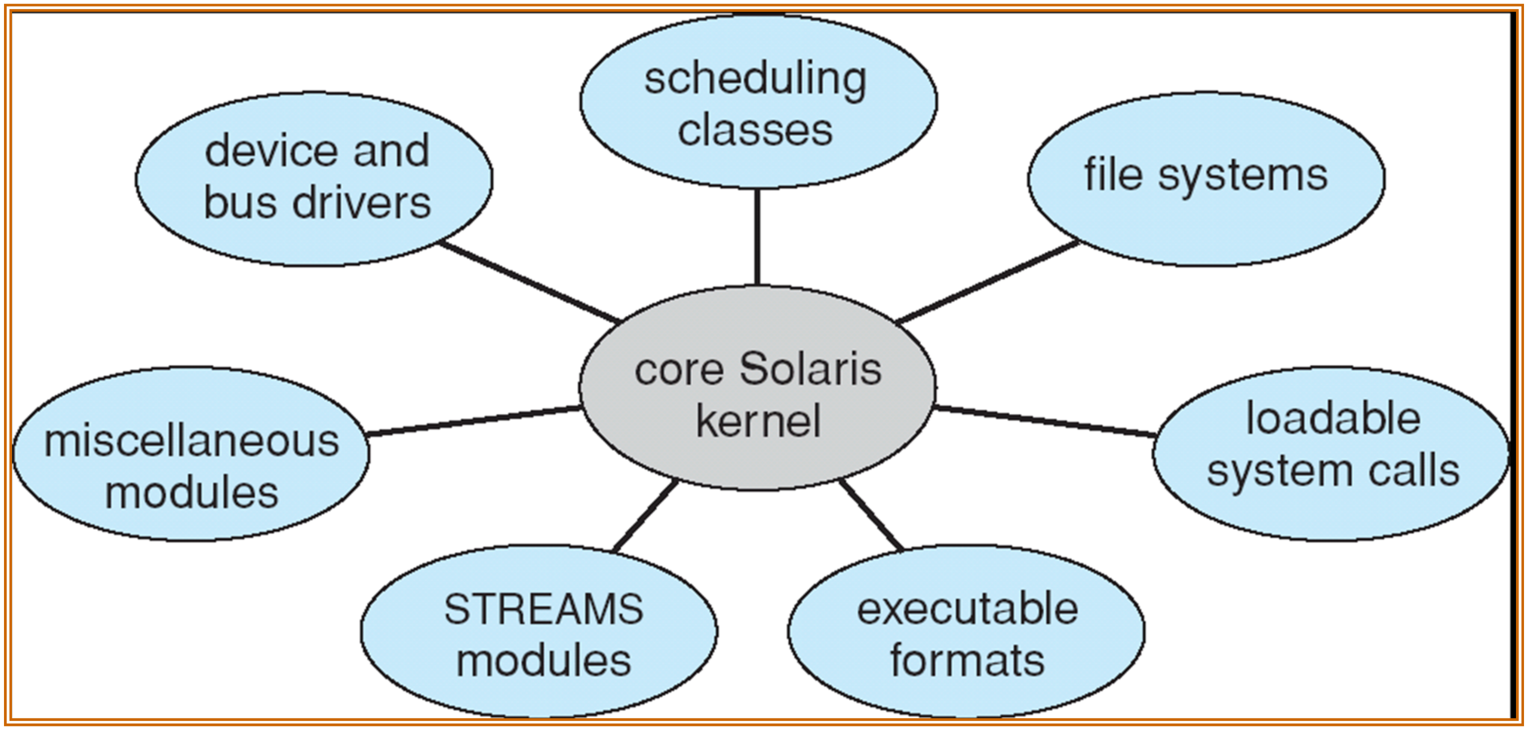
Figure 2.18 **Linux Kernel Components**

:: Stallings

Modules

- Most modern operating systems implement kernel *modules*
  - Uses object-oriented approach
  - Each core component is separate
  - Each talks to the others over known interfaces
  - Each is loadable as needed within the kernel
- Overall, similar to layers but with more flexible

Solaris Modular Approach



Operating System Services

- One set of operating-system services provides functions that are helpful to the user:
  - User interface - Almost all operating systems have a user interface (UI)
    - Varies between Command-Line (CLI), Graphics User Interface (GUI), Batch
  - Program execution - The system must be able to load a program into memory and to run that program, end execution, either normally or abnormally (indicating error)
  - I/O operations - A running program may require I/O, which may involve a file or an I/O device.
  - File-system manipulation - The file system is of particular interest. Obviously, programs need to read and write files and directories, create and delete them, search them, list file Information, permission management
  - Communications – Processes may exchange information, on the same computer or between computers over a network
    - Communications may be via shared memory or through message passing (packets moved by the OS)
  - Error detection – OS needs to be constantly aware of possible errors
    - May occur in the CPU and memory hardware, in I/O devices, in user program
    - For each type of error, OS should take the appropriate action to ensure correct and consistent computing
    - Debugging facilities can greatly enhance the user’s and programmer’s abilities to efficiently use the system

Operating System Services (Cont.)

- Another set of OS functions exists for ensuring the efficient operation of the system itself via resource sharing
  - **Resource allocation -** When multiple users or multiple jobs running concurrently, resources must be allocated to each of them
    - Many types of resources - Some (such as CPU cycles,mainmemory, and file storage) may have special allocation code, others (such as I/O devices) may have general request and release code.
  - **Accounting -** To keep track of which users use how much and what kinds of computer resources
  - **Protection and security -** The owners of information stored in a multiuser or networked computer system may want to control use of that information, concurrent processes should not interfere with each other
    - **Protection** involves ensuring that all access to system resources is controlled
    - **Security** of the system from outsiders requires user authentication, extends to defending external I/O devices from invalid access attempts
    - If a system is to be protected and secure, precautions must be instituted throughout it. A chain is only as strong as its weakest link.

Interaction between Application and OS

- CLI allows direct command entry
  - Sometimes implemented in kernel, sometimes by systems program
  - Sometimes multiple flavors implemented – **shells**
  - Primarily fetches a command from user and executes: Sometimes commands built-in, sometimes just names of program (adding new features doesn't require shell modification)
- User-friendly **desktop** metaphor interface
  - Usually mouse, keyboard, and monitor
  - **Icons** represent files, programs, actions, etc
  - Various mouse buttons over objects in the interface cause various actions (provide information, options, execute function, open directory (known as a **folder**)
  - Invented at Xerox PARC
- Many systems now include both CLI and GUI interfaces
  - Microsoft Windows is GUI with CLI "command" shell
  - Apple Mac OS X as "Aqua" GUI interface with UNIX kernel underneath and shells available
  - Solaris is CLI with optional GUI interfaces (Java Desktop, KDE)

Accessing an OS Service

- Runtime organization
  - Service is a Subroutine
  - Service is an Autonomous Process ("client-server")

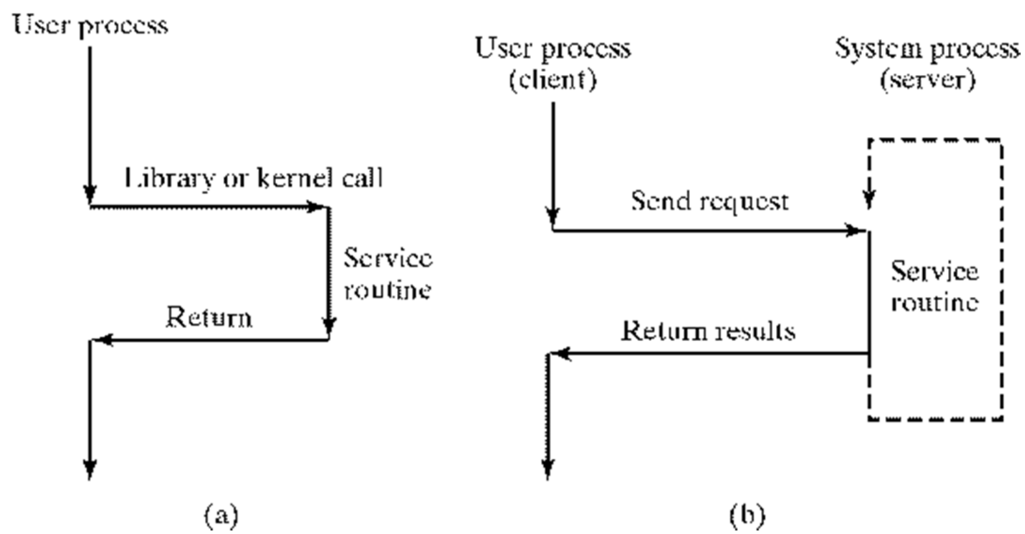


Figure 1-12

Summary

- Organization of computing systems
- Components of OS
  - Process, Memory, I/O, File, Security, etc.
  - Safety
- Organization of components
  - Monolithic, Layered, Microkernel
- Interaction between programs and operating systems
  - System calls