CS3.304.M22
Advanced Operating Systems
Lecture # 03

Manish Shrivastava
LTRC, IIIT Hyderabad

Objectives

- The main objective is to understand the operational part of any computer.
- Understanding the general principles of OS design.
  - Focus on general-purpose, multi-user, uni-processor systems.
  - Emphasis on widely applicable concepts rather than any specific features of any specific OS.
- Understanding problems, solutions and design choices.
- Understanding the structure of specific OSs: UNIX, LINUX, WINDOWS

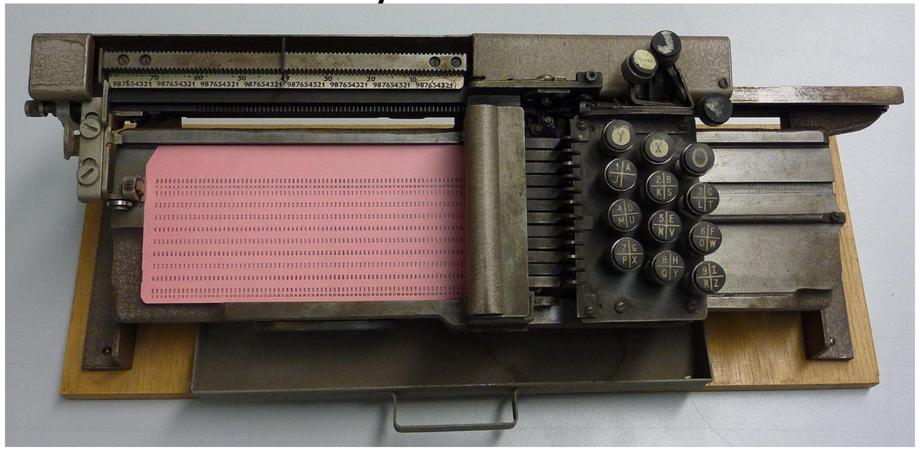
Early Systems (Serial Processing)

- **1940-50:**
  - The programmer interacted directly with the computer hardware.
  - Display light, switches, printer, card reader.
  - No OS. Error is displayed through lights.

- **Problems:**
  - Scheduling → Users spend lots of time at the computer.
    - Signup sheet was used.
  - Job Setup time
    - Loading and compiling
      - Mounting and Un-mounting of tapes
      - Setting up of card desks
  - Libraries of functions, linkers, loaders, debuggers, and I/O driver routines were available for all the users.

Early Systems

- Early computers were (physically) large machines run from a console.
- The programmer would operate the program directly from the console.
  - The program is loaded to the memory from panel of switches, paper tape, and from punched cards.
- As time went on, additional software and hardware were developed.
  - Card readers, line printers, and magnetic tape became common place.
  - Libraries, loaders, and common functions were created.
    - Software reusability.



IBM029

Early Systems

- The setup time was a real problem
- CPU is idle while tapes are being mounted or the programmer was operating the console.
- In the early days, few computers were available and they were expensive (millions of dollars).
  - operational costs: power, cooling, programmers.
- Main question:

How to increase the utilization of CPU ?

Early Systems

- The solution was two fold.
- First, a professional computer operator was hired.
  - Once the program was finished, the operator could start next job.
  - The operator sets up the job, produces the dump, and starts the next job.
  - The set up time was reduced due to operator’s experience.
- Second, jobs with similar needs were batched together and run through the computer as a group.
  - For example, if there is a FORTRAN job, COBOL job, and FORTRAN job, two FORTRAN jobs were batched together.
- However, during transition time CPU sat idle.
- Automatic job sequencing to avoid CPU idle time.
  - A first rudimentary OS was created
  - A small program called a **resident monitor** was developed.
  - The resident monitor always resided in memory.

Simple Batch Systems

- In serial systems
  - Machines were very expensive
  - Wasting time was not acceptable.
- To improve usage, the concept of batch OS was developed.
- The main idea is the use of software known as monitor.
  - The user no longer has access to machine.
- The user submits the job (tape) to the operator.
- The operator batches the jobs together sequentially, places entire batch as an input device for use by the computer.

Features of Batch System

- The batch OS is simply a program.
- It relies on the ability of the processor to fetch instructions from various portions of main memory to seize and relinquish control.
- Hardware features:
  - **Memory protection**: While the user program is running, it must not alter the memory area containing the monitor.
  - If such is the case the processor hardware should detect the error and transfer control to monitor.
  - **Timer:** A timer is used to prevent the single job from monopolizing the system

Features of Batch System

- Hardware Features
  - **Privileged instructions**
    - Contains instructions that are only executed by monitor.
    - I/O instructions
    - If a program encounters them the control shifts through monitor.
  - **Interrupts:** It gives OS more flexibility.
    - Relinquishing control and regain control
- With batch OS the machine time alters between execution of user programs and execution of monitor.
- Two overheads
  - Machine time is consumed by the monitor.
  - Memory is consumed by the monitor.
- Still, they improved the performance over serial systems.

Problem with Batch System

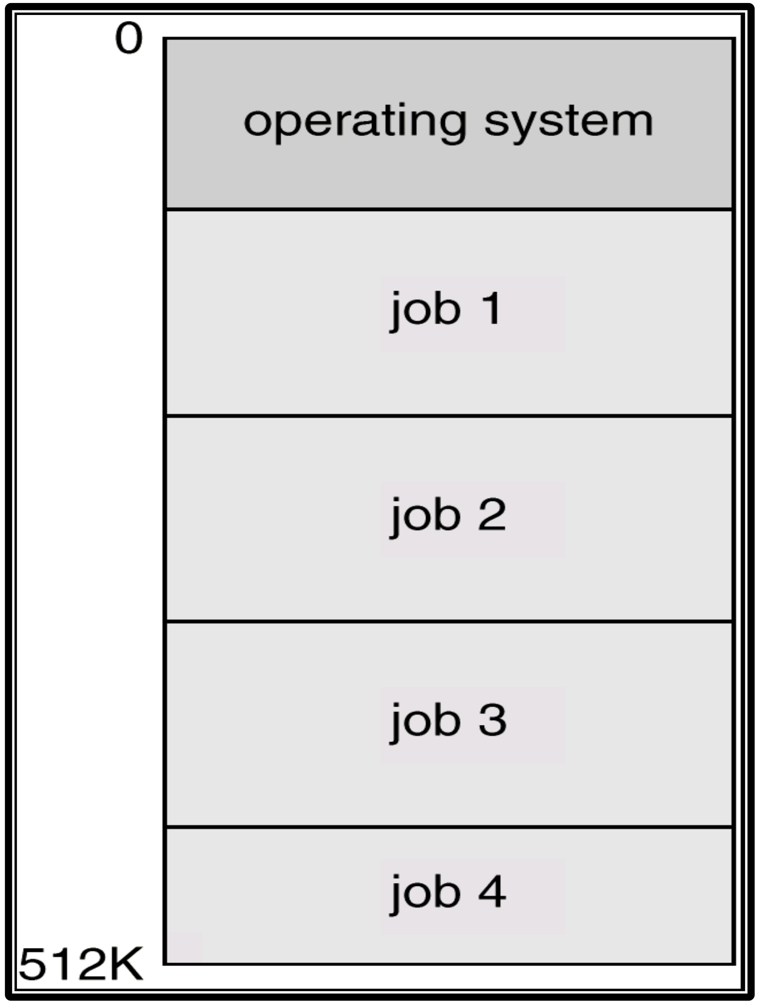
- CPU is idle
- Speed of mechanical devices is very slower than those of electronic devices.
- CPU works in a microsecond range
  - Thousands of instructions/second
- A card reader may read 1200 cards per minute (20 cards per second)
- CPU speed has increased at a faster rate.
- Tape technology improved the performance little-bit.
- Main perceived problem
  - Turn-around time: upto two days
  - CPU often underutilized
    - Most of the time was spent reading and writing from tape.

Multi-programmed Batched Systems (1960s) (or Multi tasking)

- If CPU is executing a job and requires a tape to be mounted
  - In a non multi-programmed system
    - CPU sits idle.
  - In a Multi-programmed system
    - CPU takes up another job.
- **Multiprogramming is the first instance when the OS started taking decisions.**
- Job scheduling is done by OS.
- Having several programs in the memory requires memory management.

Multi-programmed Batched Systems (1960s) (or Multi tasking)

- A single user can not keep either CPU or I/O busy.
- Multiprogramming increases CPU utilization by organizing jobs such that the CPU always has one to execute.
- The OS keeps several jobs in memory at a time and CPU is multiplexed among them



OS Requirements (60s)

- OS Research in 60s
  - MULTICS at MIT
  - Atlas (spooling, demand paging) at Manchester Univ.
  - Multiprogramming
    - Memory allocation and protection
    - I/O operations were responsibility of OS.
  - Interactive systems
    - Scheduling issues
    - Swapping and virtual memory.
  - Users wanted permanent files
    - Hierarchical directory systems.
  - Increased in size and complexity were not well understood
    - IBM: OS/360

UNIX (early 1970s)

- Originally developed at Bell labs for the PDP-7
  - Ken Thomson
  - Dennis Ritchie
- Smaller & Simpler
  - Process spawn and control
    - Each command creates a new process
  - Simple inter-process communication
  - Command interpreter not built in: runs as a user process
  - Files were streams of bytes.
  - Hierarchical file system
- Advantages
  - Written in a high-level language
  - Distributed in source form
  - Powerful OS primitives on an inexpensive platform

Personal Computers (1980s)

- Originally
  - Single user
  - Simplified OSs
    - No memory protection
    - MS-DOS
- Now run sophisticated OSs
  - Windows NT, Linux

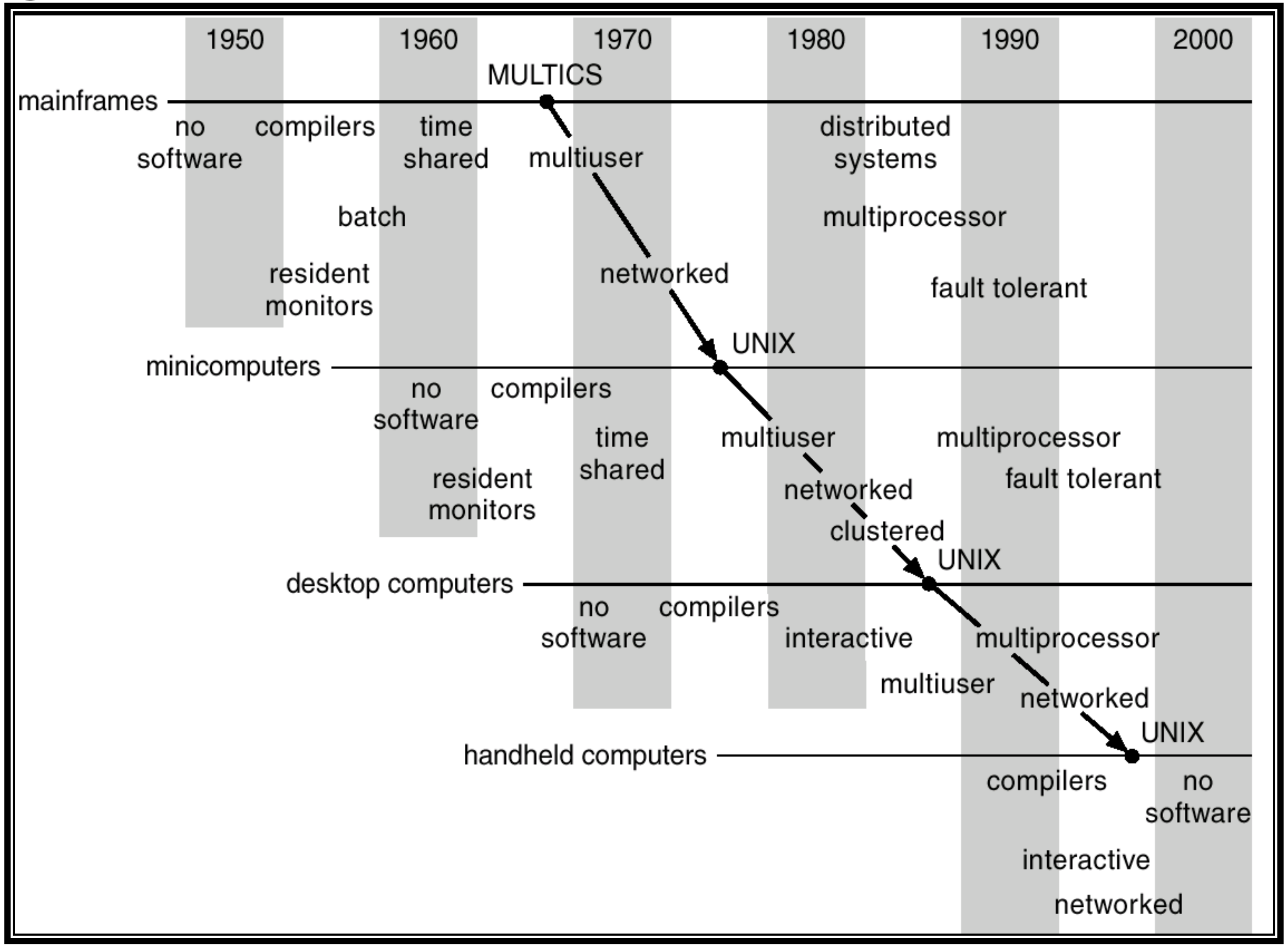
Networks of workstations (1990s)

- High-speed network connections
- Local & world-wide
- Client-server systems
  - File systems
  - Remote windowing systems
- Support a variety of node OSs
  - Unix, Windows XP, OS/2

Future

- Distributed systems
  - Network is invisible
- Micro-kernel and extensible OSs
  - Supports multiple OS flavors
    - E.g., Mach, Amoeba, WINDOWS XP
- Embedded services and network computers
  - Computer runs a very thin OS (Java Virtual machine).

Migration of Operating-System Concepts and Features

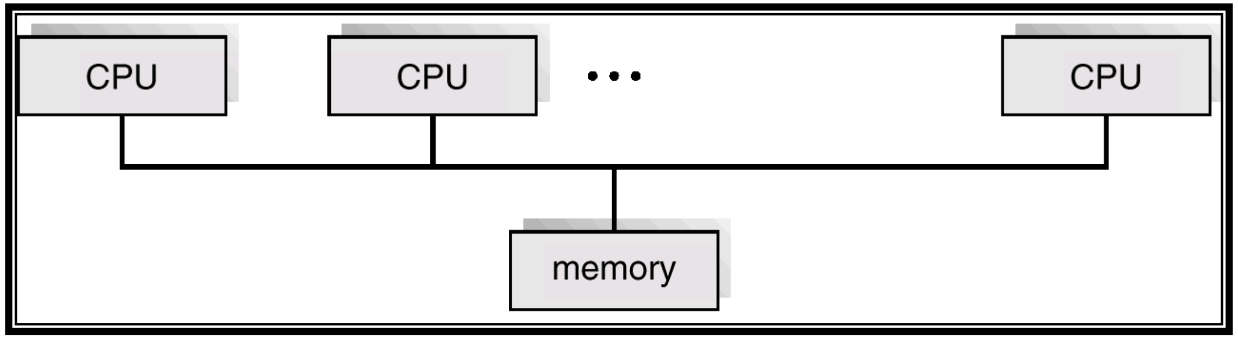


Multiprocessor Systems

- Multiprocessor systems with more than one CPU in close communication.
- *Tightly coupled system* – processors share memory and a clock; communication usually takes place through the shared memory.
- Advantages of multiprocessor system:
  - Increased *throughput:* more processors more work
  - Economical: more computing in same box
  - Increased reliability
    - Graceful degradation & Fault tolerance: failure of one processor will not halt the system. Service is proportional to the level of surviving hardware.

Parallel Systems

- *Symmetric multiprocessing (SMP)*
  - Each processor runs and identical copy of the operating system.
  - Many processes can run at once without performance deterioration.
  - Most modern operating systems support SMP
- *Asymmetric multiprocessing*
  - Each processor is assigned a specific task; master processor schedules and allocated work to slave processors.
  - More common in extremely large systems



SMP Architecture

Distributed Systems

- Distribute the computation among several physical processors.
- *Loosely coupled system*
  - each processor has its own local memory;
  - processors communicate with one another through various communications lines such as high-speed buses or telephone lines.
- Advantages of distributed systems
  - Resources Sharing
  - Computation speed up/load sharing
  - Reliability
  - Communications

Real-Time Systems

- Often used as a control device in a dedicated application such as controlling scientific experiments, medical imaging systems, industrial control systems, and some display systems.

- Well-defined fixed-time constraints.
  - A process must complete within the defined constraints or system will fail.

- Real-Time systems may be either *hard* or *soft* real-time.

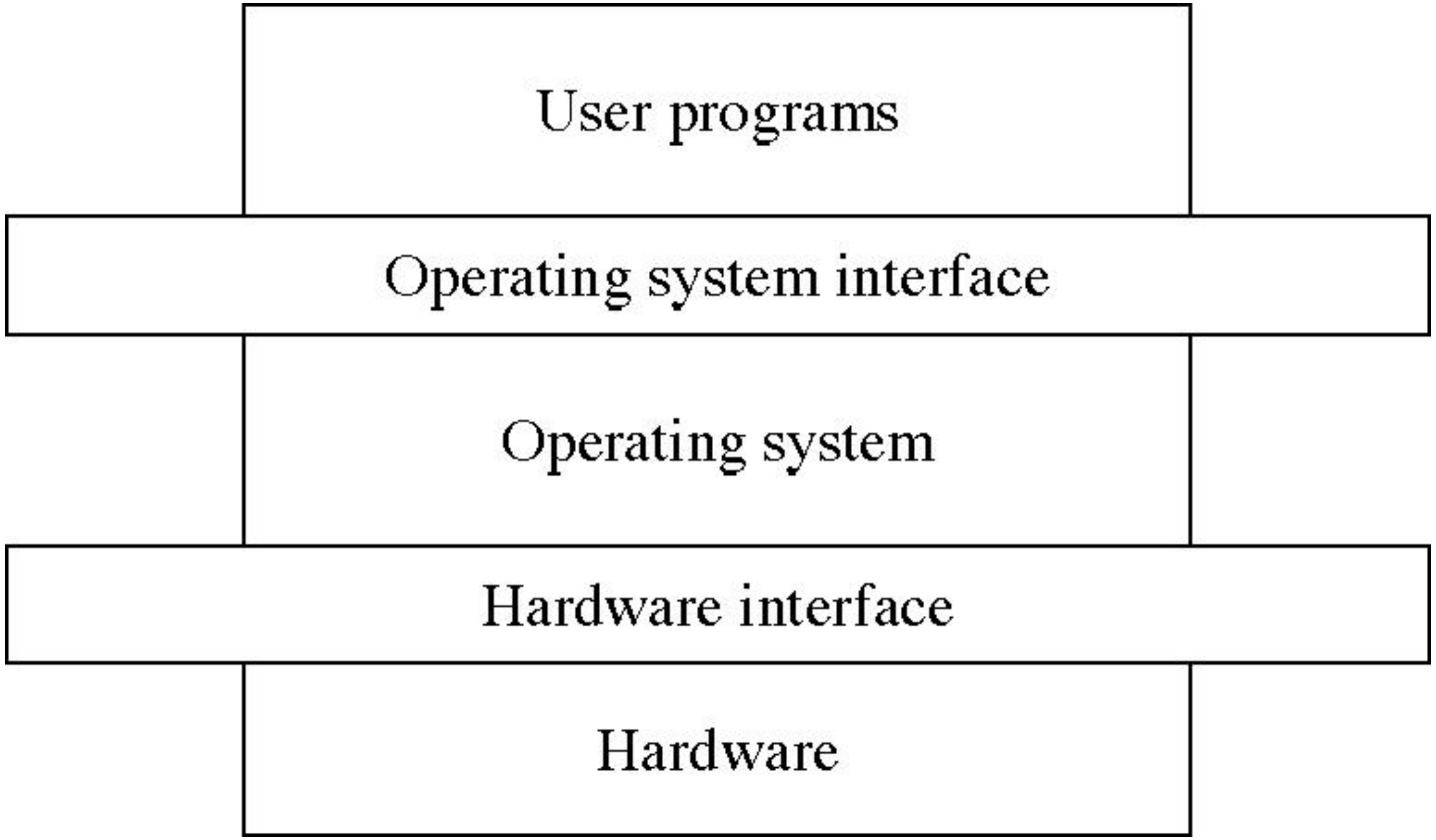
Real-Time Systems

- Hard real-time:
  - Guarantees that critical tasks complete within time.
  - All the delays in the system are bounded.
  - Secondary storage limited or absent, data stored in short term memory, or read-only memory (ROM)
  - Conflicts with time-sharing systems, not supported by general-purpose operating systems.
- Soft real-time
  - Critical time tasks gets priority over other tasks, and retails that priority until it completes.
  - Limited utility in industrial control of robotics
  - Useful in applications (multimedia, virtual reality) requiring advanced operating-system features.

Concept of Virtual Computer

- Multilevel implementation
  - also called layered
- Resources
  - Hardware: provided to the OS
  - Logical (virtual): created by the OS
- Resource management
  - transformation
  - multiplexing
    - time and space

Levels in a computer system



Design: Two-level Implementation

- Two-level implementation
  - Lower level is a problem-specific language
  - Upper level solves the problem at hand
  - Lower level is reusable

- In operating systems
  - *mechanism*: lower level of basic functions, does not change
  - *policy*: upper level policy decisions, easy to change and experiment

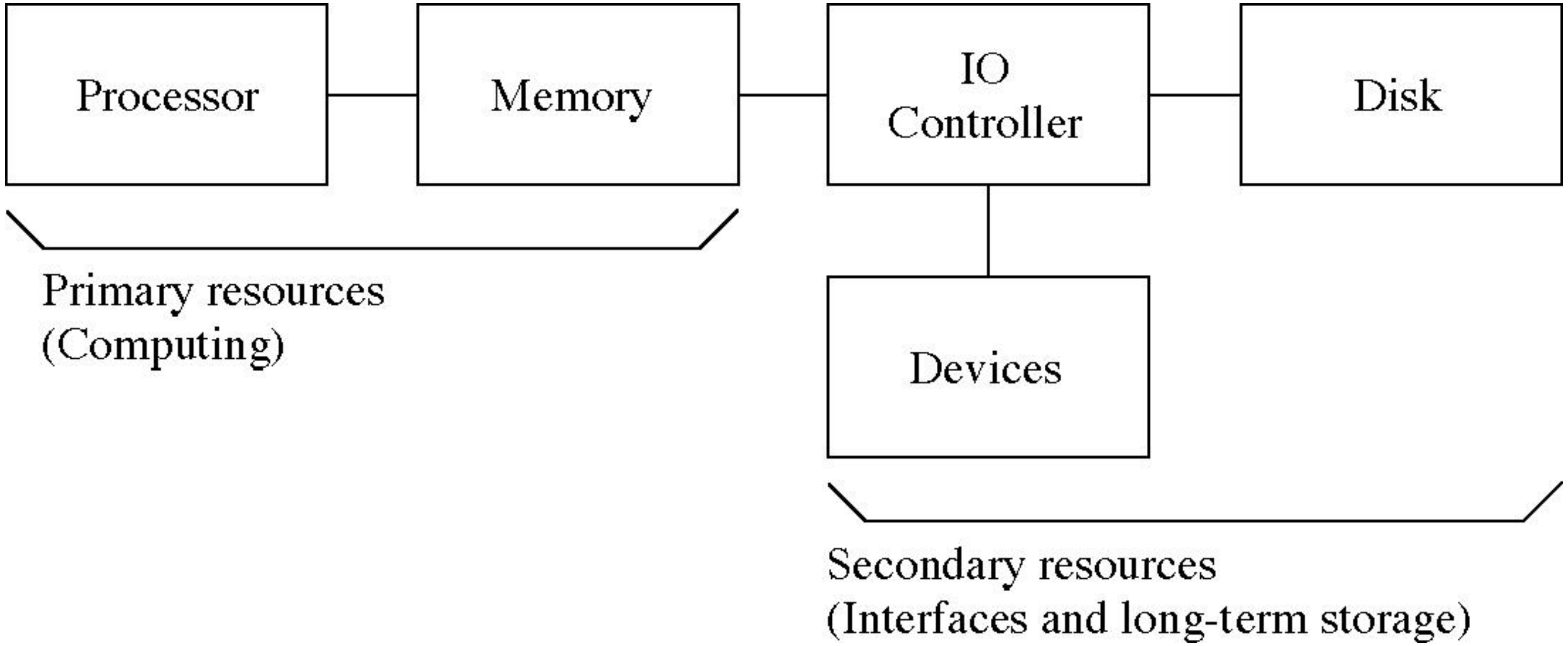
Operating System Functions

- Resource manager
  - manage hardware and software resources
- Virtual machine manager
  - implement a virtual machine for processes to run in
  - a nicer environment than the bare hardware

Hardware Resources

- *Processor*: execute instructions
- *Memory*: store programs and data
- *Input/output (I/O)controllers*: transfer to and from devices
- *Disk devices*: long-term storage
- *Other devices*: conversion between internal and external data representations

Hardware Resources



Resource Management Functions

- *Transforming* physical resources to logical resources
  - Making the resources easier to use
- *Multiplexing* one physical resource to several logical resources
  - Creating multiple, logical copies of resources
- *Scheduling* physical and logical resources
  - Deciding who gets to use the resources

Types of Multiplexing

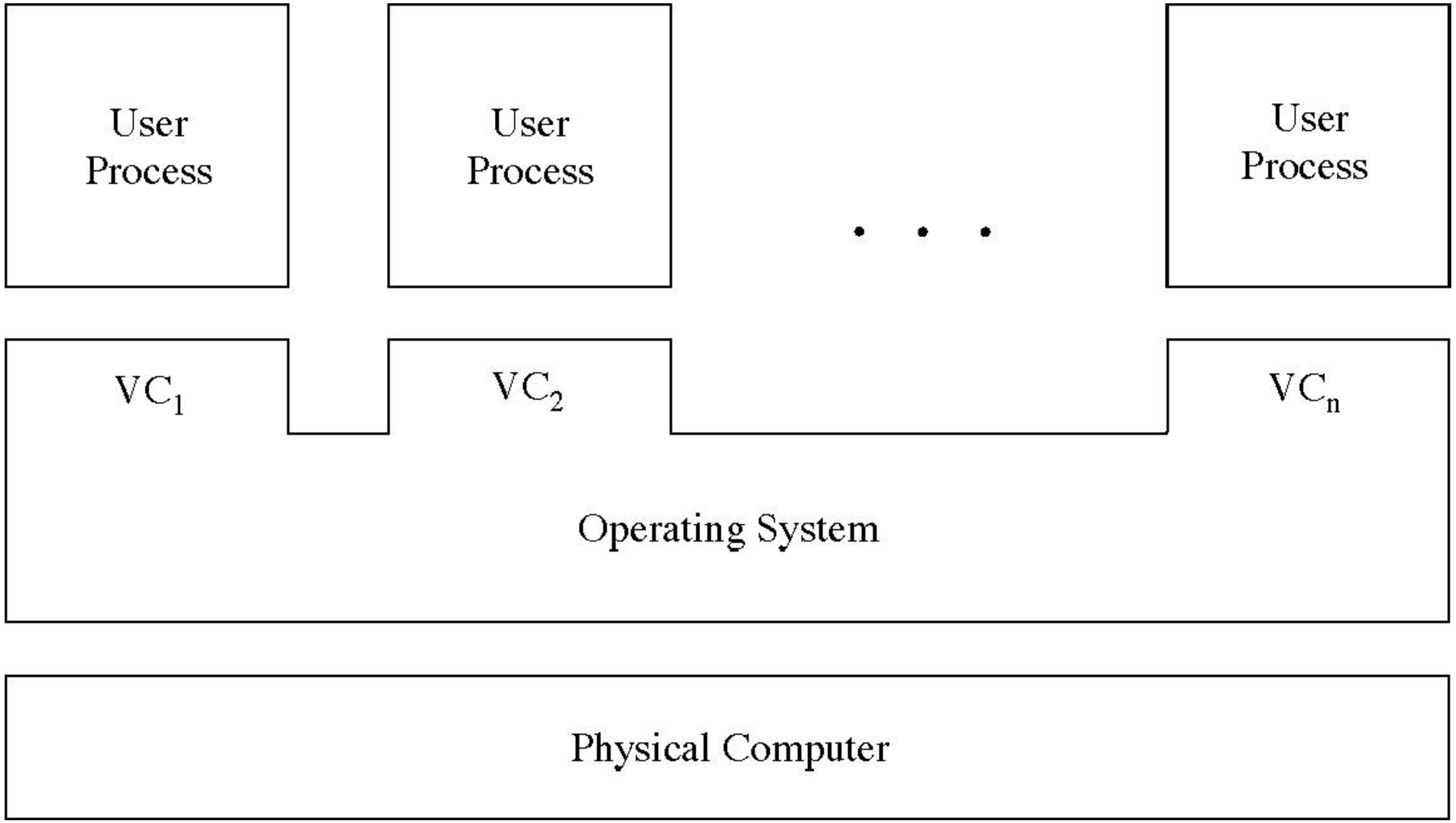
- Time multiplexing
  - time-sharing
  - scheduling a serially-reusable resource among several users

- Space multiplexing
  - space-sharing
  - dividing a multiple-use resource up among several users

Virtual Computers

- Processor virtualized to processes
  - mainly time-multiplexing
- Memory virtualized to address spaces
  - space and time multiplexing
- Disks virtualized to files
  - space-multiplexing
  - transforming

Multiple Virtual Computers



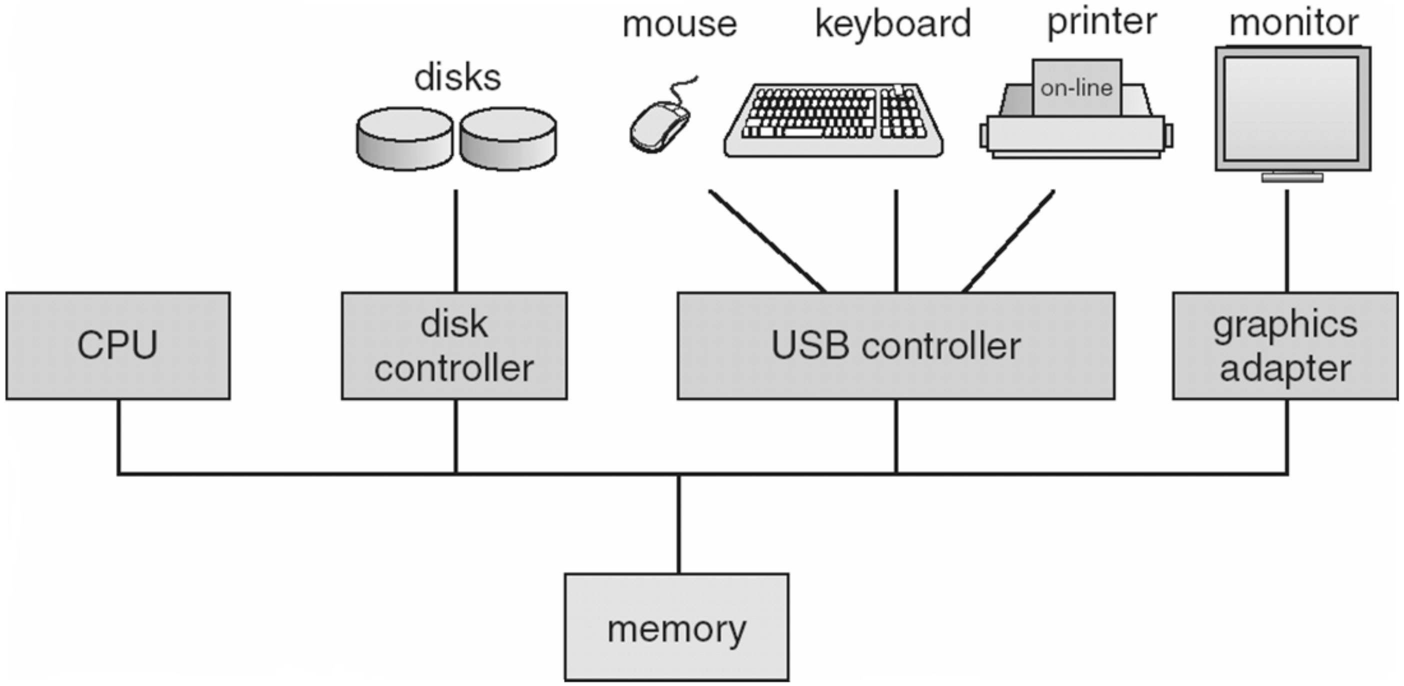
Do we need an OS?

- Not always
  - Some programs run “stand-alone”
- But they are very useful
  - Reusable functions
  - Easier to use than the bare hardware

System Structures I

Computer System Organization

- Computer-system operation
  - One or more CPUs, device controllers connect through common bus providing access to shared memory
  - Concurrent execution of CPUs and devices competing for memory cycles



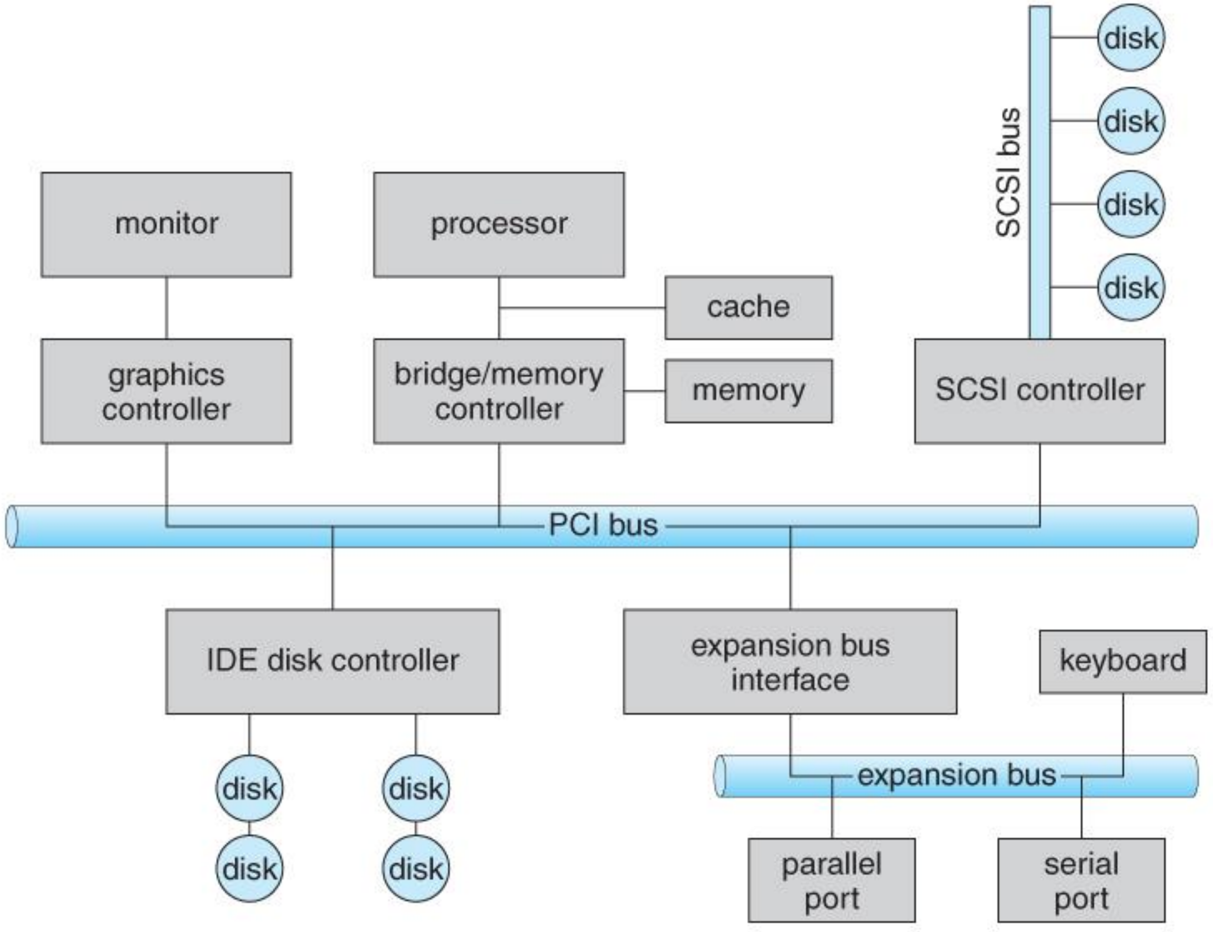
Computer-Components

- Computer consists of processor, memory, and I/O components, with one or more modules of each type. These modules are connected through interconnection network.
- I/O devices and the CPU can execute concurrently.
- Each device controller is in-charge of a particular device type.
- Each device controller has a local buffer.
- CPU moves data from/to main memory to/from local buffers
- I/O is from the device to local buffer of controller.
- **Device controller informs CPU that it has finished its operation by causing an *interrupt*.**

I/O Hardware

- I/O devices can be categorized as storage, communications, user-interface, and others
- Devices connect with the computer via ***ports***, e.g. a serial or parallel port.
- A common set of wires connecting multiple devices is termed a ***bus.***
  - Buses include rigid protocols for the types of messages that can be sent across the bus and the procedures for resolving contention issues.
  - Bus types commonly found in a modern PC:
    - The ***PCI bus*** connects high-speed high-bandwidth devices to the memory subsystem (and the CPU.)
    - The ***expansion bus*** connects slower low-bandwidth devices, which typically deliver data one character at a time (with buffering.)
    - The ***SCSI bus*** connects a number of SCSI devices to a common SCSI controller.
    - A ***daisy-chain bus,*** is when a string of devices is connected to each other like beads on a chain, and only one of the devices is directly connected to the host.

PC Bus Structure



Registers

- One way of communicating with devices is through ***registers*** associated with each port. Registers may be one to four bytes in size, and may typically include (a subset of) the following four:
  - The ***data-in register*** is read by the host to get input from the device.
  - The ***data-out register*** is written by the host to send output.
  - The ***status register*** has bits read by the host to ascertain the status of the device, such as idle, ready for input, busy, error, transaction complete, etc.
  - The ***control register*** has bits written by the host to issue commands or to change settings of the device such as parity checking, word length, or full- versus half-duplex operation.

Memory Mapped I/O

- Another technique for communicating with devices is ***memory-mapped I/O.***
  - In this case a certain portion of the processor's address space is mapped to the device, and communications occur by reading and writing directly to/from those memory areas.
  - Memory-mapped I/O is suitable for devices which must move large quantities of data quickly, such as graphics cards.
  - Memory-mapped I/O can be used either alone or more often in combination with traditional registers. For example, graphics cards still use registers for control information such as setting the video mode.
  - A potential problem exists with memory-mapped I/O, if a process is allowed to write directly to the address space used by a memory-mapped I/O device.

Polling

- One simple means of device ***handshaking*** involves polling:
  - The host repeatedly checks the ***busy bit*** on the device until it becomes clear.
  - The host writes a byte of data into the **data-out register**, and sets the ***write bit*** in the command register (in either order.)
  - The host sets the ***command ready bit*** in the command register to notify the device of the pending command.
  - When the device controller sees the **command-ready bit** set, it first sets the **busy bit**.
  - Then the device controller reads the **command register**, sees **the write bit** set, reads the byte of data from the data-out register, and outputs the byte of data.
  - The device controller then clears the ***error bit*** in the status register, the command-ready bit, and finally clears the busy bit, signaling the completion of the operation.
- Polling can be very fast and efficient, if both the device and the controller are fast and if there is significant data to transfer. It becomes inefficient, however, if the host must wait a long time in the busy loop waiting for the device, or if frequent checks need to be made for data that is infrequently there.

Architecture of a Simple Computer

- **Processor:** Controls the operation of the computer and performs data processing functions. It is called CPU.
- **Main memory:** Stores data and programs; it is volatile
- **I/O modules:** Moves data between the computer and external environment.
- **System bus:** mechanism of communication among processors, main memory, and I/O modules.

Simple Computer

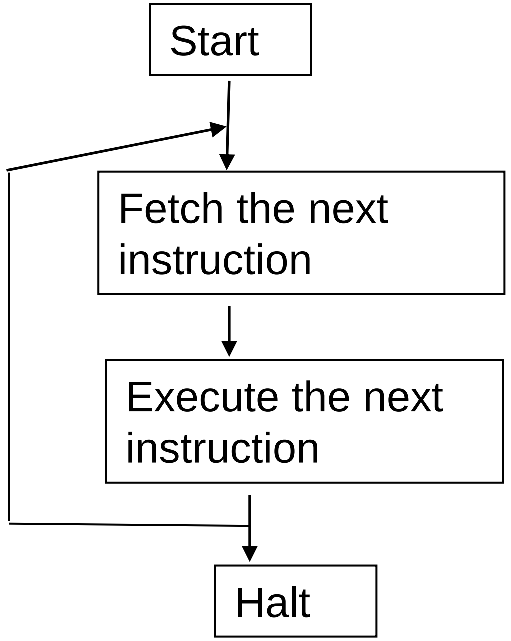
- Operation: Processor controls everything.
  - **MAR**: Memory address register: which specifies the address in memory for the next read or write.
  - **MBR**: Memory buffer register: which contains the data to be written into memory or which receives data read from memory.
  - **I/O AR**: Address of I/O device
  - **I/O BR**: Exchange of data between I/O and computer.

Simple Computer

- **Processor Registers**: Within the processor there is a set of registers that provide a level of memory that is faster and smaller than main memory.
  - User visible registers: Available to programmer
    - **Data registers**: Can be used by the programmer.
    - **Address Registers**: Contains Main memory address of data and instructions.
      - **Index register**: Index to base value
      - **Segment pointer**: It contains a reference to a particular segment.
      - **Stack pointer**: Points top of the stack.
  - **Control and status registers**: These are employed to control the operation of the processor. Differ from machine to machine.
    - MAR, MBR,I/O AR, and I/O BR
    - **Program Counter (PC):** Contains the address of the instruction to be fetched.
    - **Instruction Register (IR)**: Contains the instruction most recently fetched.
    - **Program Status Word (PSW)**: It is a register or a set of registers.

Simple Computer

- **PSW** (Program Status word): It is a register or a set of registers.
    - **Sign:** contains sign bit of last arithmetic operation
    - **Zero**: it is set if the result of arithmetic operation is zero
    - **Carry:** It is set if there is a carry or borrow.
    - **Equal:** If the compare result is equality
    - **Overflow:** It is set if the result is overflow.
    - **Interrupt enable/disable:** Used to disable or enable interrupts.
    - **Supervisor:** Indicates whether the processor is executing in supervisor or user mode.
- **Instruction execution:**
    - Program execution is the main function of the computer.
    - Instruction fetch and execute

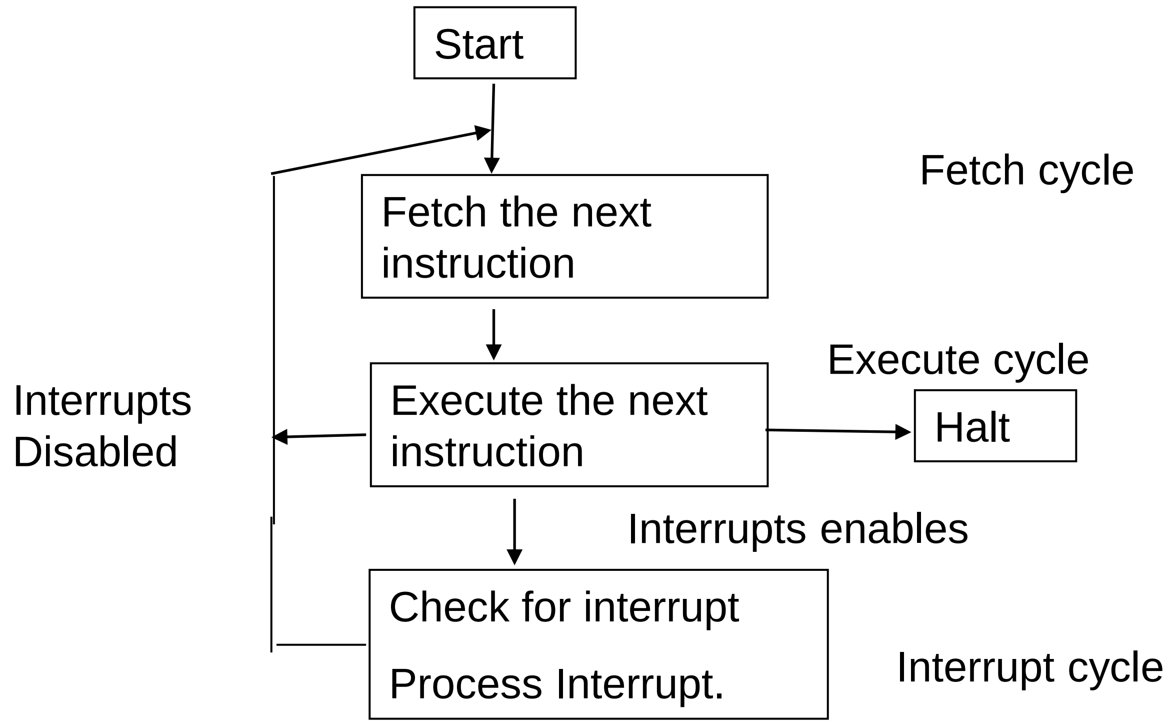


Simple Computer

- **Fetching:** Bringing the instructions from the main memory.
  - **PC**: Contains the address of the next instruction to be fetched.
    - Incremented unless told otherwise
- **Execute:**
  - The processor interprets the instructions and performs the required action.
    - **Processor-memory:** Transfer data from the memory
    - **Processor-I/O-** transfer data from peripheral device from the memory.
    - **Data processing:** Arithmetic and logic operations
    - **Control:** The instructions may specify the sequence of the next instruction to be fetched.

Interrupt Processing

- To improve the performance, interrupts are provided.
- With interrupts, the processor can be engaged in executing other processes while an I/O operation is in progress.
- Interrupt cycle is added to the instruction cycle.



Common Functions of Interrupts

- Interrupt transfers control to the interrupt service routine generally, through the ***interrupt vector***, which contains the addresses of all the service routines.
- Interrupt architecture must save the address of the interrupted instruction.
- Incoming interrupts are *disabled* while another interrupt is being processed to prevent a *lost interrupt*.
- A ***trap*** is a software-generated interrupt caused either by an error or a user request.
- **An operating system is *interrupt* driven.**

I/O Controller Interrupting

- To start I/O operation, CPU loads the appropriate registers within the device controller
- The device controller examines the contents of these registers to determine what action to take.
- If it s a read operation the controller transfers the data into local buffer.
- Then it informs the CPU through interrupt.
- The operating system preserves the state of the CPU by storing registers and the program counter.

Interrupt Handling

- **Interrupt handler:** The CPU hardware has a wire called interrupt request line; that CPU senses after executing every instruction.
  - When a CPU senses a signal, it saves the PC, and PSW on a stack and jump to the interrupt handler routine at a fixed address in memory.

- **Interrupt vector:** It contains the memory addresses of specialized interrupt handlers.

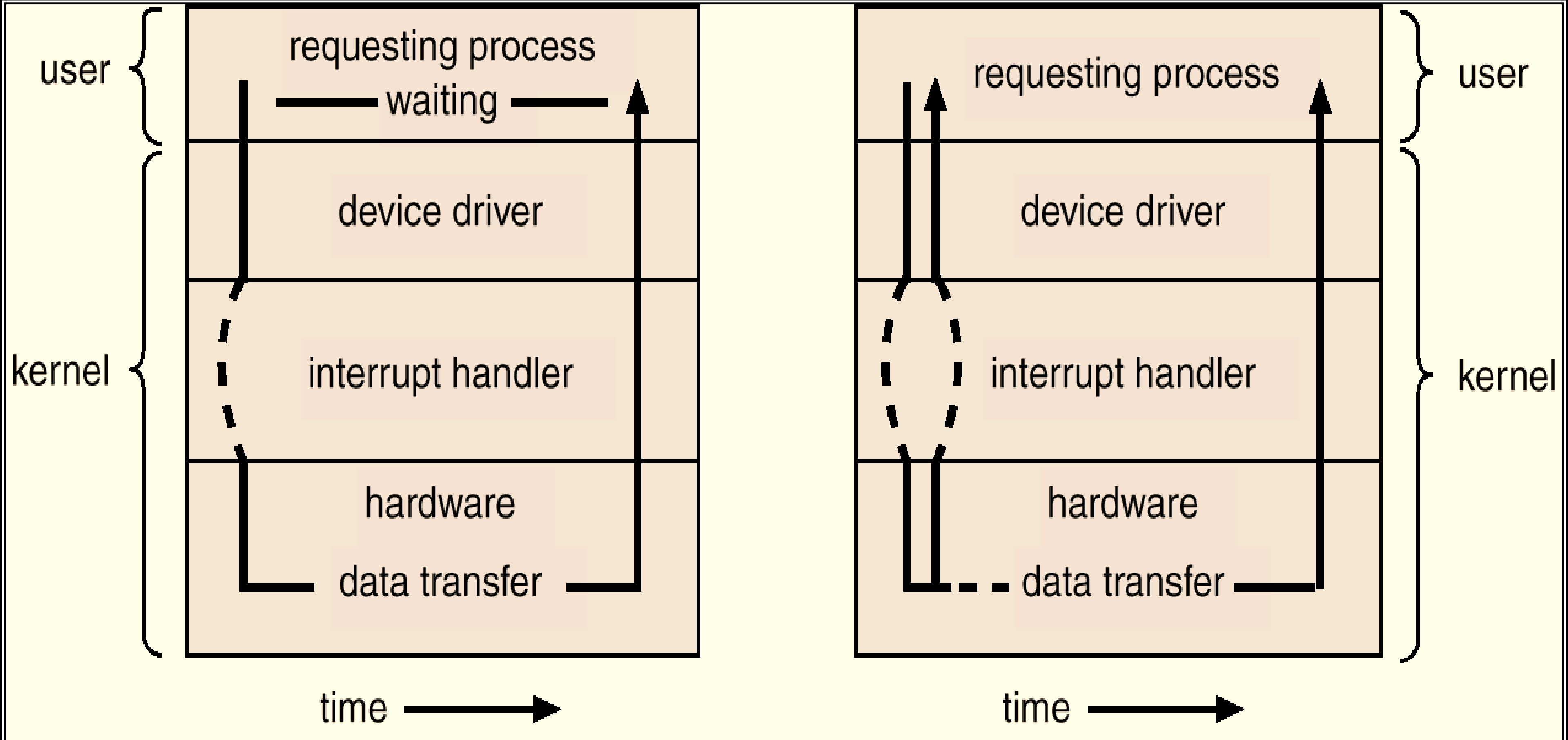
Interrupt Handling

- Interrupts allow devices to notify the CPU when they have data to transfer or when an operation is complete, allowing the CPU to perform other duties when no I/O transfers need its immediate attention.
- The CPU has an ***interrupt-request line*** that is sensed after every instruction.
  - A device's controller ***raises*** an interrupt by asserting a signal on the interrupt request line.
  - The CPU then performs a state save, and transfers control to the ***interrupt handler*** routine at a fixed address in memory. (The CPU ***catches*** the interrupt and ***dispatches*** the interrupt handler.)
  - The interrupt handler determines the cause of the interrupt, performs the necessary processing, performs a state restore, and executes a ***return from interrupt*** instruction to return control to the CPU. (The interrupt handler ***clears*** the interrupt by servicing the device.)
    - (Note that the state restored does not need to be the same state as the one that was saved when the interrupt went off.)

I/O Structure

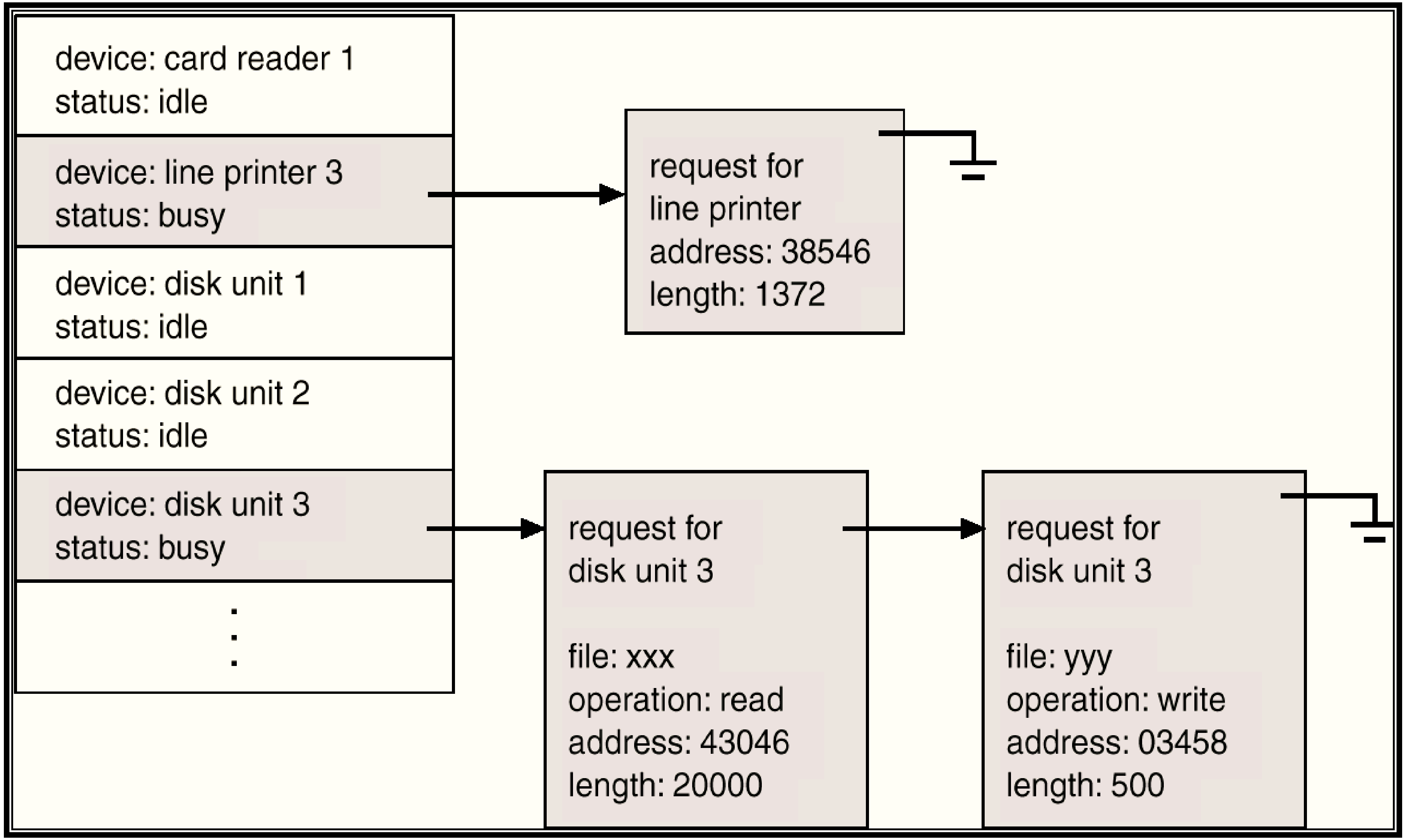
- **Synchronous I/O**: After I/O starts, control returns to user program only upon I/O completion.
  - Wait instruction idles the CPU until the next interrupt
  - At most one I/O request is outstanding at a time, no simultaneous I/O processing.
- **Asynchronous I/O**: After I/O starts, control returns to user program without waiting for I/O completion.
- ***non-blocking I/O:*** the I/O request returns immediately, whether the requested I/O operation has (completely) occurred or not
- *Device-status table* contains entry for each I/O device indicating its type, address, and state.
- Operating system indexes into I/O device-status table to determine device status and to modify table entry.

Two I/O Methods



(a) Synchronous (b) Asynchronous

Device-Status Table



I/O communication techniques

- Two kinds of data transfer
  - **Program data transfer**: CPU checks the I/O status
    - The I/O module does not interrupt the processor.
    - Processor is responsible for extracting the data from main memory and shifting data out of main memory.
    - It is a time-consuming process that keeps the processor busy unnecessarily.
  - **Interrupt driven data transfer:** when I/O is ready it interrupts the CPU
    - In Interrupt driven data transfer, the I/O module will interrupt the processor to request service when it is ready to exchange data with the processor.

Modern Computer

- The above description is adequate for simple interrupt-driven I/O, but there are three needs in modern computing which complicate the picture:
  - The need to defer interrupt handling during critical processing,
  - The need to determine ***which*** interrupt handler to invoke, without having to poll all devices to see which one needs attention, and
  - The need for multi-level interrupts, so the system can differentiate between high- and low-priority interrupts for proper response.

Interrupt Controller

- These issues are handled in modern computer architectures with ***interrupt-controller*** hardware.
  - Most CPUs now have two interrupt-request lines: One that is ***non-maskable*** for critical error conditions and one that is ***maskable,*** that the CPU can temporarily ignore during critical processing.
  - The interrupt mechanism accepts an ***address,*** which is usually one of a small set of numbers for an offset into a table called the ***interrupt vector.*** This table holds the addresses of routines prepared to process specific interrupts.
  - The number of possible interrupt handlers still exceeds the range of defined interrupt numbers, so multiple handlers can be ***interrupt chained***. Effectively the addresses held in the interrupt vectors are the head pointers for linked-lists of interrupt handlers.
  - Modern interrupt hardware also supports ***interrupt priority levels***, allowing systems to mask off only lower-priority interrupts while servicing a high-priority interrupt, or conversely to allow a high-priority signal to interrupt the processing of a low-priority one.

Interrupt Vector Table

vector number	description
0	divide error
1	debug exception
2	null interrupt
3	breakpoint
4	INTO-detected overflow
5	bound range exception
6	invalid opcode
7	device not available
8	double fault
9	coprocessor segment overrun (reserved)
10	invalid task state segment
11	segment not present
12	stack fault
13	general protection
14	page fault
15	(Intel reserved, do not use)
16	floating-point error
17	alignment check
18	machine check
19–31	(Intel reserved, do not use)
32–255	maskable interrupts

Interrupt Controller

- At boot time the system determines which devices are present, and loads the appropriate handler addresses into the interrupt table.
- During operation, devices signal errors or the completion of commands via interrupts.
- Exceptions, such as dividing by zero, invalid memory accesses, or attempts to access kernel mode instructions can be signaled via interrupts.
- Time slicing and context switches can also be implemented using the interrupt mechanism.
    - The scheduler sets a hardware timer before transferring control over to a user process.
    - When the timer raises the interrupt request line, the CPU performs a state-save, and transfers control over to the proper interrupt handler, which in turn runs the scheduler.
    - The scheduler does a state-restore of a ***different*** process before resetting the timer and issuing the return-from-interrupt instruction.

Some Interrupts

- A similar example involves the paging system for virtual memory –
  - A page fault causes an interrupt, which in turn issues an I/O request and a context switch, moving the interrupted process into the wait queue and selecting a different process to run. When the I/O request has completed then the device interrupts, and the interrupt handler moves the process from the wait queue into the ready queue.

- Some OSs (Solaris OS) use a multi-threaded kernel and priority threads to assign different threads to different interrupt handlers. This allows for the "simultaneous" handling of multiple interrupts, and the assurance that high-priority interrupts will take precedence over low-priority ones and over user processes.

Some Interrupts

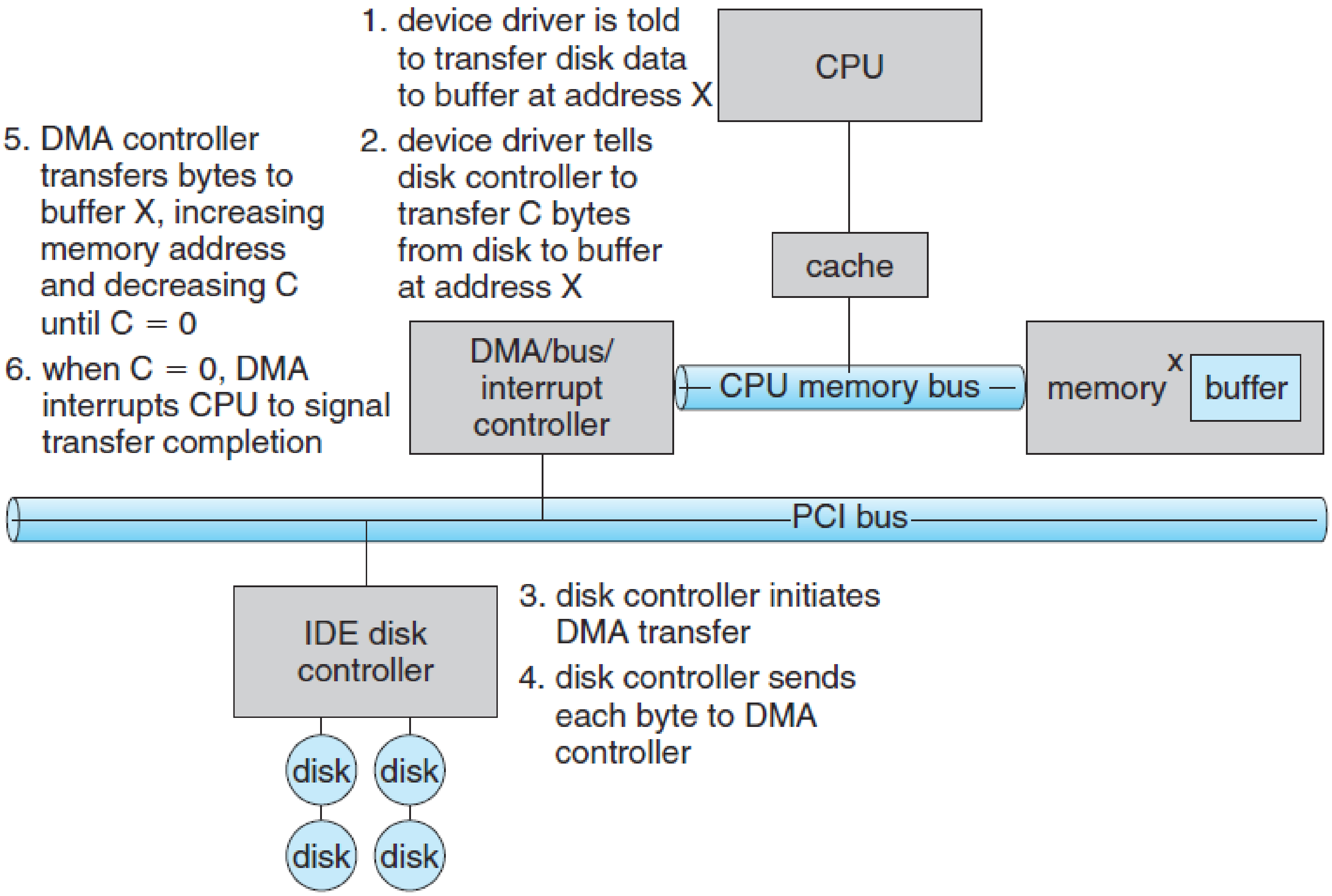
- System calls are implemented via ***software interrupts,*** a.k.a. ***traps.*** When a (library) program needs work performed in kernel mode, it sets command information and possibly data addresses in certain registers, and then raises a software interrupt.
- The completion of a disk read operation involves **two** interrupts:
  - A high-priority interrupt acknowledges the device completion, and issues the next disk request so that the hardware does not sit idle.
  - A lower-priority interrupt transfers the data from the kernel memory space to the user space, and then transfers the process from the waiting queue to the ready queue.

Direct Memory Access

- For devices that transfer large quantities of data (such as disk controllers), it is wasteful to tie up the CPU transferring data in and out of registers one byte at a time

- Instead this work can be off-loaded to a special processor, known as the ***Direct Memory Access, DMA, Controller.***
    - The host issues a command to the DMA controller, indicating the details of transfer operation. The DMA interrupts the CPU when the transfer is complete.
    - A simple DMA controller is a standard component in modern PCs, and many ***bus-mastering*** I/O cards contain their own DMA hardware.
    - Handshaking between DMA controllers and their devices is accomplished through two wires called the **DMA-request** and **DMA-acknowledge** wires.
    - While the DMA transfer is going on the CPU does not have access to the PCI bus (including main memory), but it does have access to its internal registers and primary and secondary caches.

Direct Memory Access



Direct Memory Access

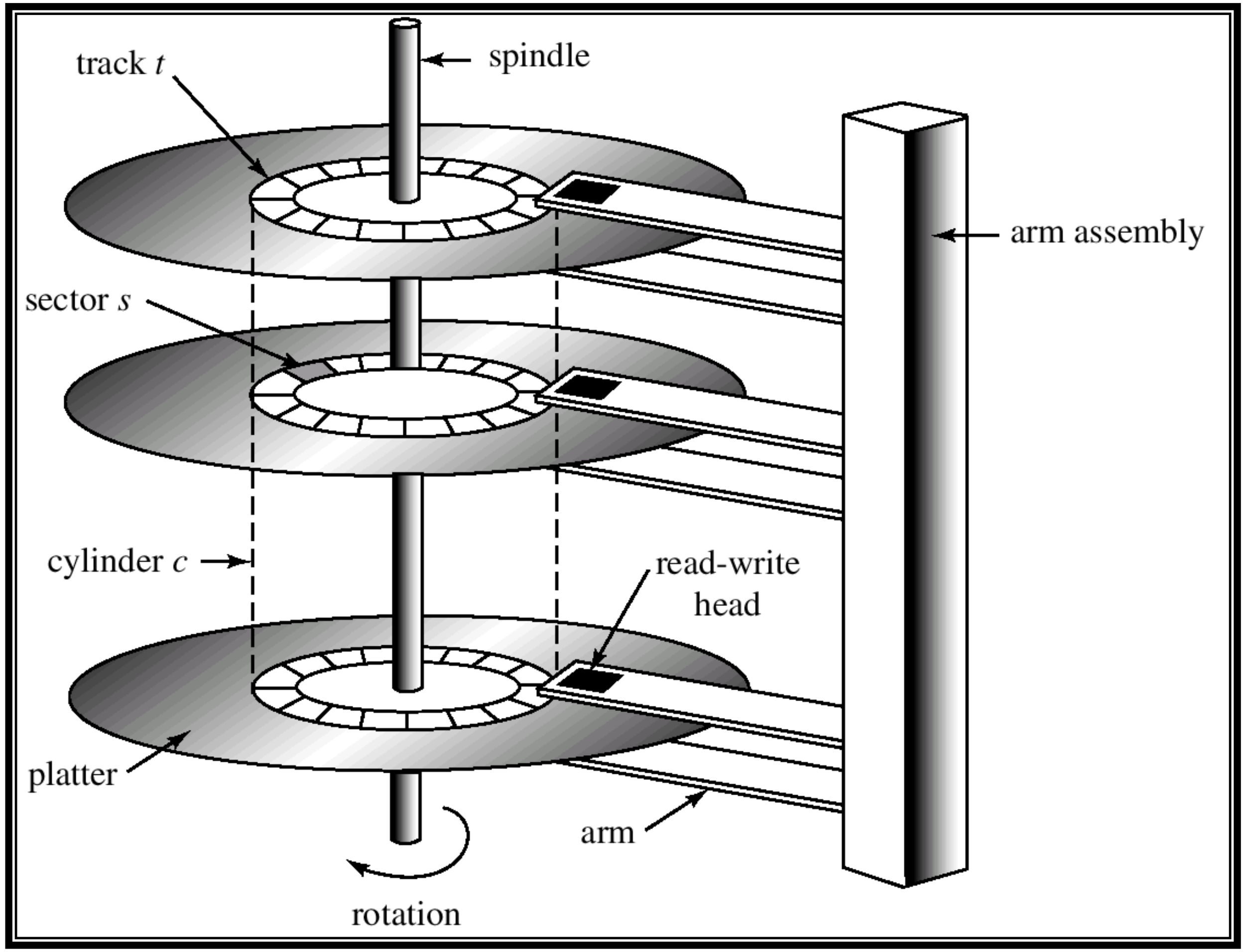
- DMA can be done in terms of either physical addresses or virtual addresses that are mapped to physical addresses. The latter approach is known as ***Direct Virtual Memory Access, DVMA,*** and allows direct data transfer from one memory-mapped device to another without using the main memory chips.

- Direct DMA access by user processes can speed up operations, but is generally forbidden by modern systems for security and protection reasons.

Storage Structure

- Main memory – only large storage media that the CPU can access directly
  - **Random access**
  - Typically **volatile**
- Secondary storage – extension of main memory that provides large **nonvolatile** storage capacity
- Magnetic disks – rigid metal or glass platters covered with magnetic recording material
  - Disk surface is logically divided into **tracks**, which are subdivided into **sectors**
  - The **disk controller** determines the logical interaction between the device and the computer

Moving-Head Disk Mechanism

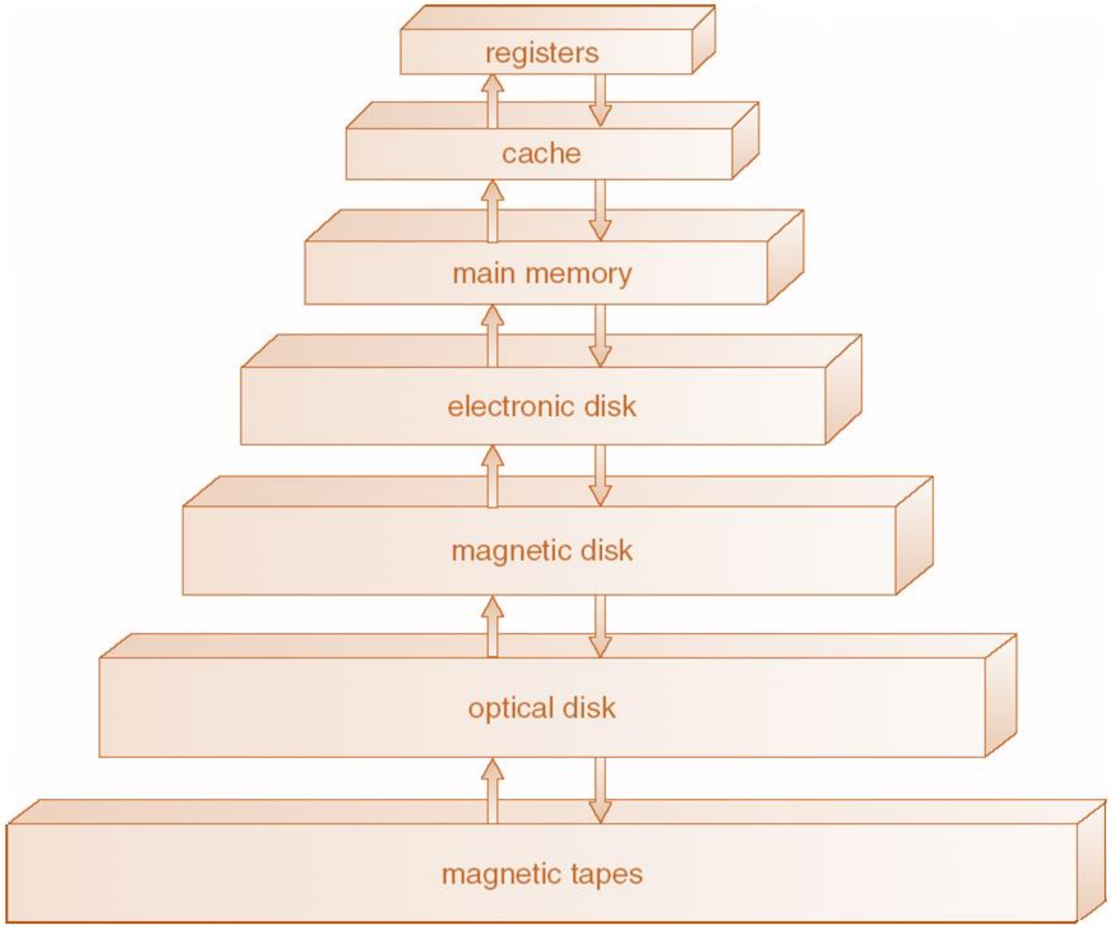


Storage Hierarchy

- Storage systems organized in hierarchy
  - Speed
  - Cost
  - Volatility

- **Caching** – copying information into faster storage system; main memory can be viewed as a *cache* for secondary storage

Storage-Device Hierarchy



Caching

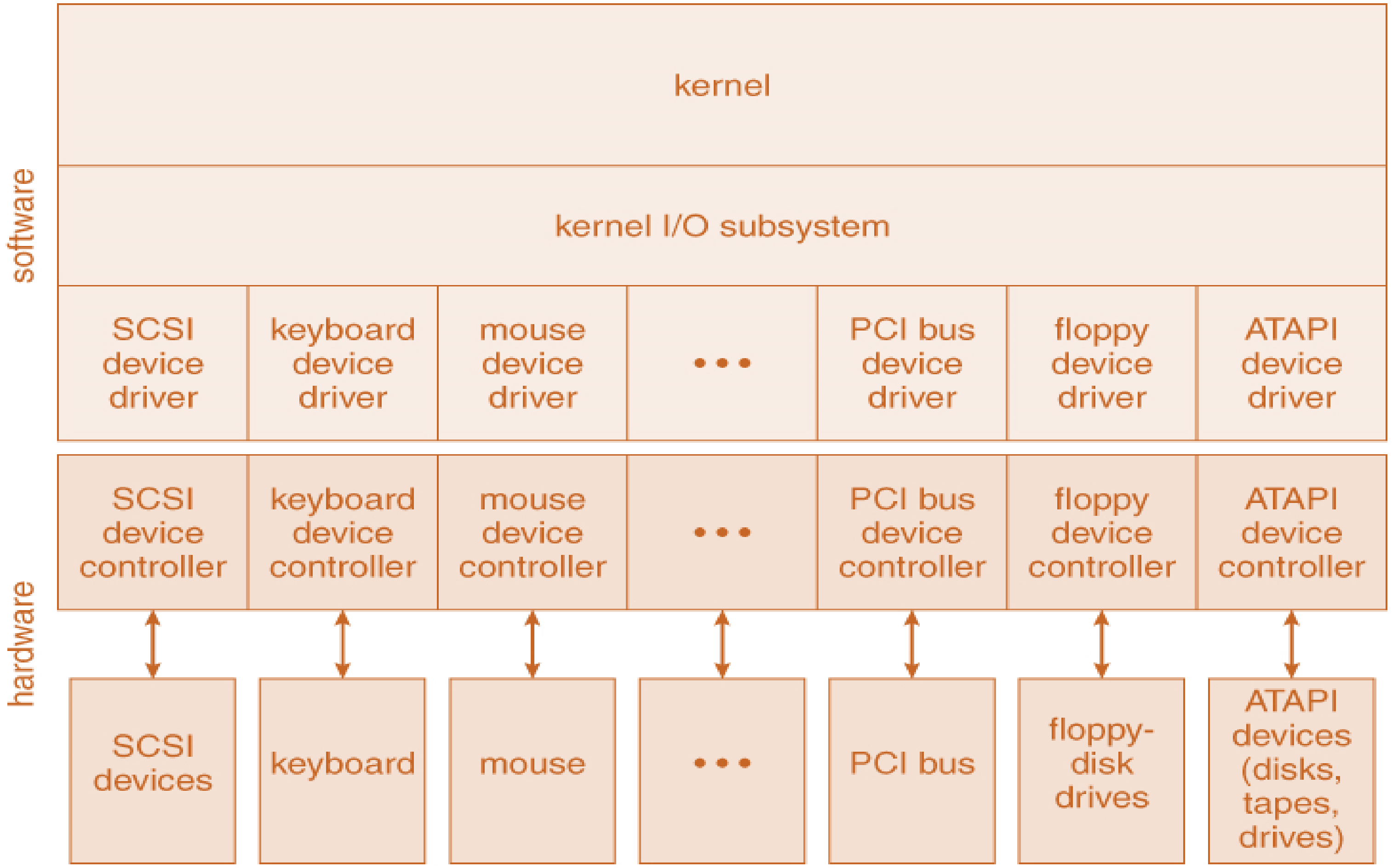
- Important principle, performed at many levels in a computer (in hardware, operating system, software)

- Information in use copied from slower to faster storage temporarily

- Faster storage (cache) checked first to determine if information is there
  - If it is, information used directly from the cache (fast)
  - If not, data copied to cache and used there

- Cache smaller than storage being cached
  - Cache management important design problem
  - Cache size and replacement policy

Application I/O Interface



Application I/O Interface

- **Block and Character Devices**
  - Block : HDD Filesystems etc
  - Character : Any guesses ??

- **Network Devices**
  - Socket Interfaces

- **Clocks and Timers**
  - PIT: Programmable interrupt Timer

Kernel I/O handling

- **I/O Scheduling**
- **Buffering**
- **Caching**
- **Spooling and Device Reservation**
  - SPOOL : ***Simultaneous Peripheral Operations On-Line***
- **Error Handling**
- **I/O Protection**

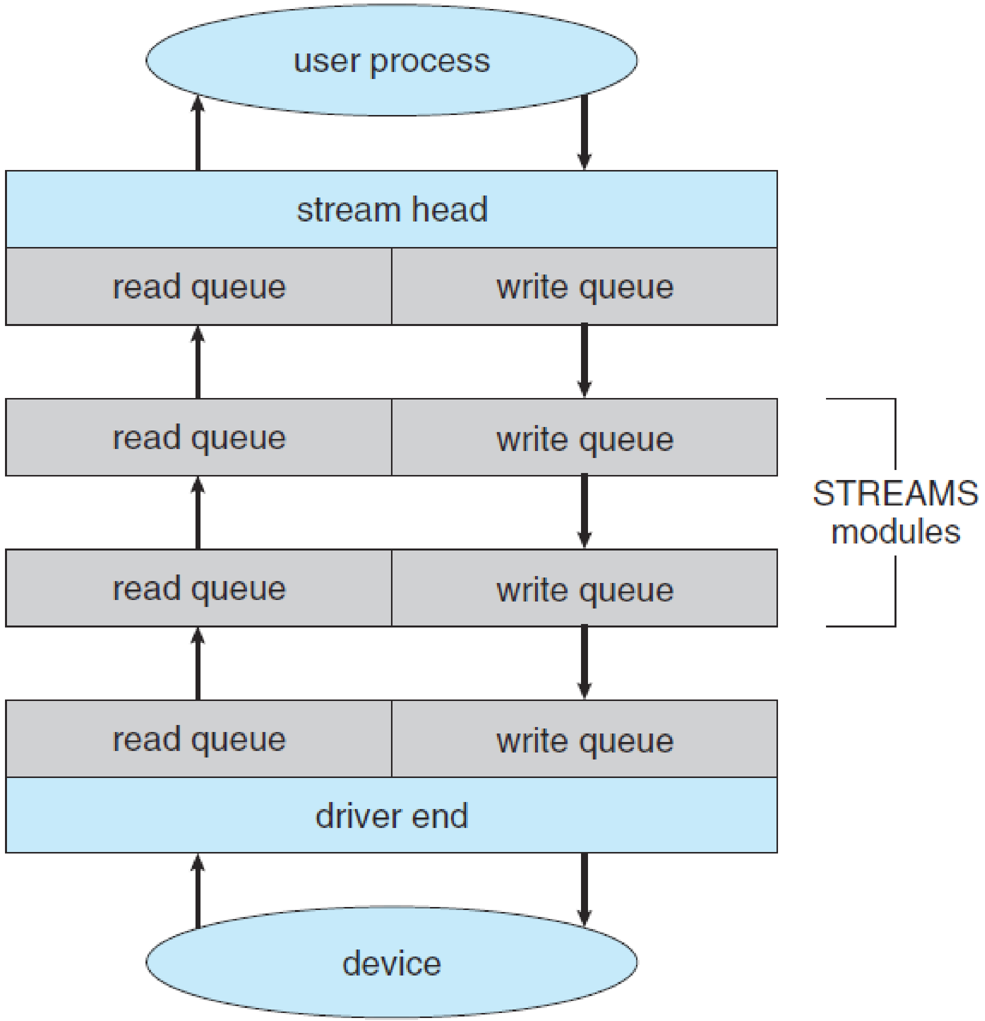
I/O request and hardware

- Users request data using file names, which must ultimately be mapped to specific blocks of data from a specific device managed by a specific device driver.
  - DOS uses the colon separator to specify a particular device (e.g. C:, LPT:, etc.)
  - UNIX uses a ***mount table*** to map filename prefixes (e.g. /usr) to specific mounted devices.
- UNIX uses special ***device files,*** usually located in /dev, to represent and access physical devices directly.
  - Each device file has a major and minor number associated with it, stored and displayed where the file size would normally go.
  - The major number is an index into a table of device drivers, and indicates which device driver handles this device. (E.g. the disk drive handler.)
  - The minor number is a parameter passed to the device driver, and indicates which specific device is to be accessed, out of the many which may be handled by a particular device driver. (e.g. a particular disk drive or partition.)
- A series of lookup tables and mappings makes the access of different devices flexible, and somewhat transparent to users.

Unix Streams

- The ***streams*** mechanism in UNIX provides a bi-directional pipeline between a user process and a device driver, onto which additional modules can be added.
    - The user process interacts with the ***stream head.***
    - The device driver interacts with the ***device end.***
    - Zero or more ***stream modules*** can be pushed onto the stream, using ioctl( ). These modules may filter and/or modify the data as it passes through the stream.
    - Each module has a ***read queue*** and a ***write queue.***
- ***Flow control*** can be optionally supported, in which case each module will buffer data until the adjacent module is ready to receive it
- Streams I/O is **asynchronous**, except for the interface between the user process and the stream head.
- The device driver **must** respond to interrupts from its device - If the adjacent module is not prepared to accept data and the device driver's buffers are all full, then data is typically dropped.
- Streams are widely used in UNIX, and are the preferred approach for device drivers.

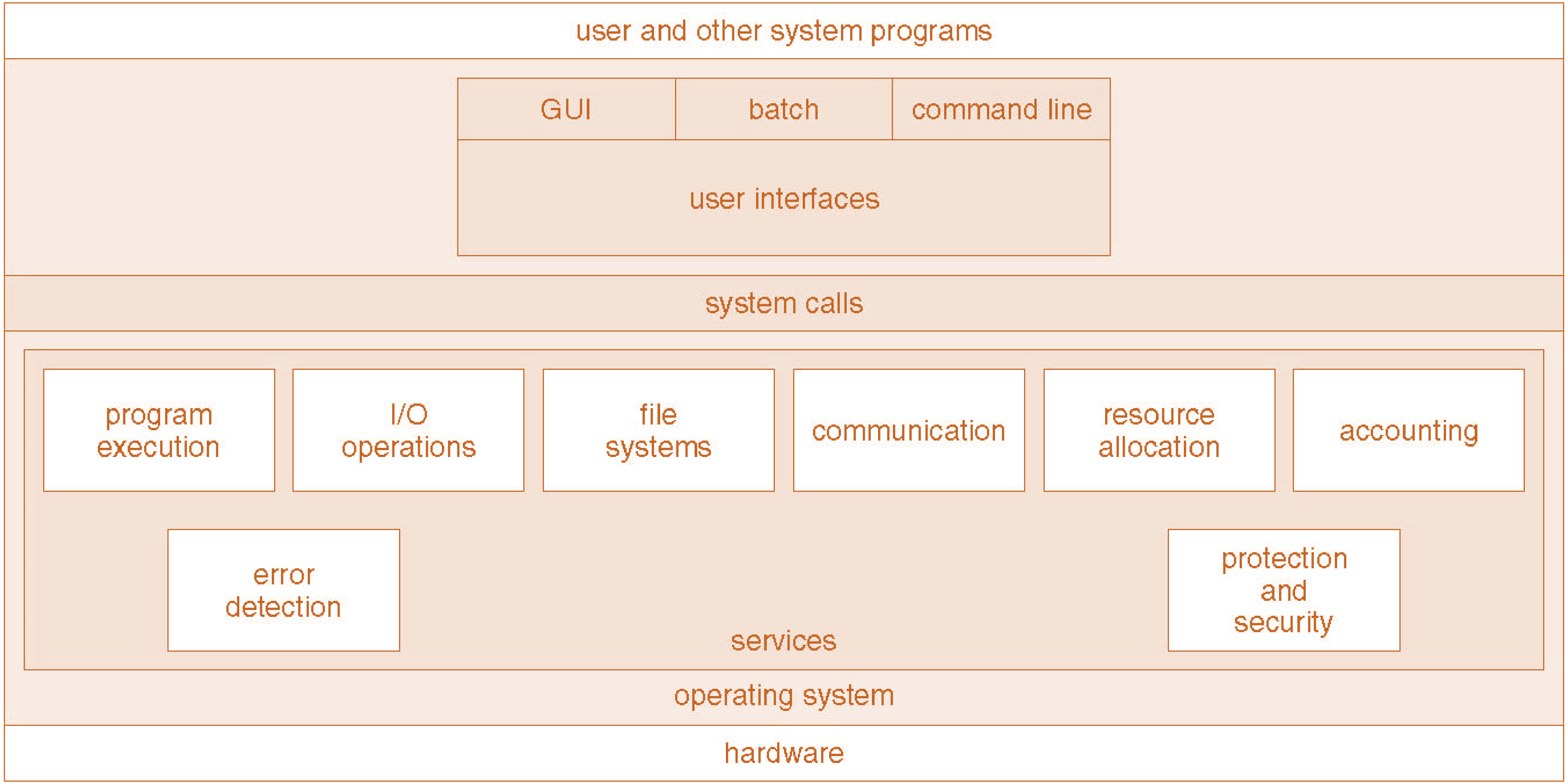
Unix Streams



OS Services

- OS Services
  - User Functions: (G/C)UI, Program execution, I/O, FS manipulation
  - Process Communication, Error Detection, Resource allocation accounting, protection and security
- User level Services
  - GUI/CLI

A View of Operating System Services



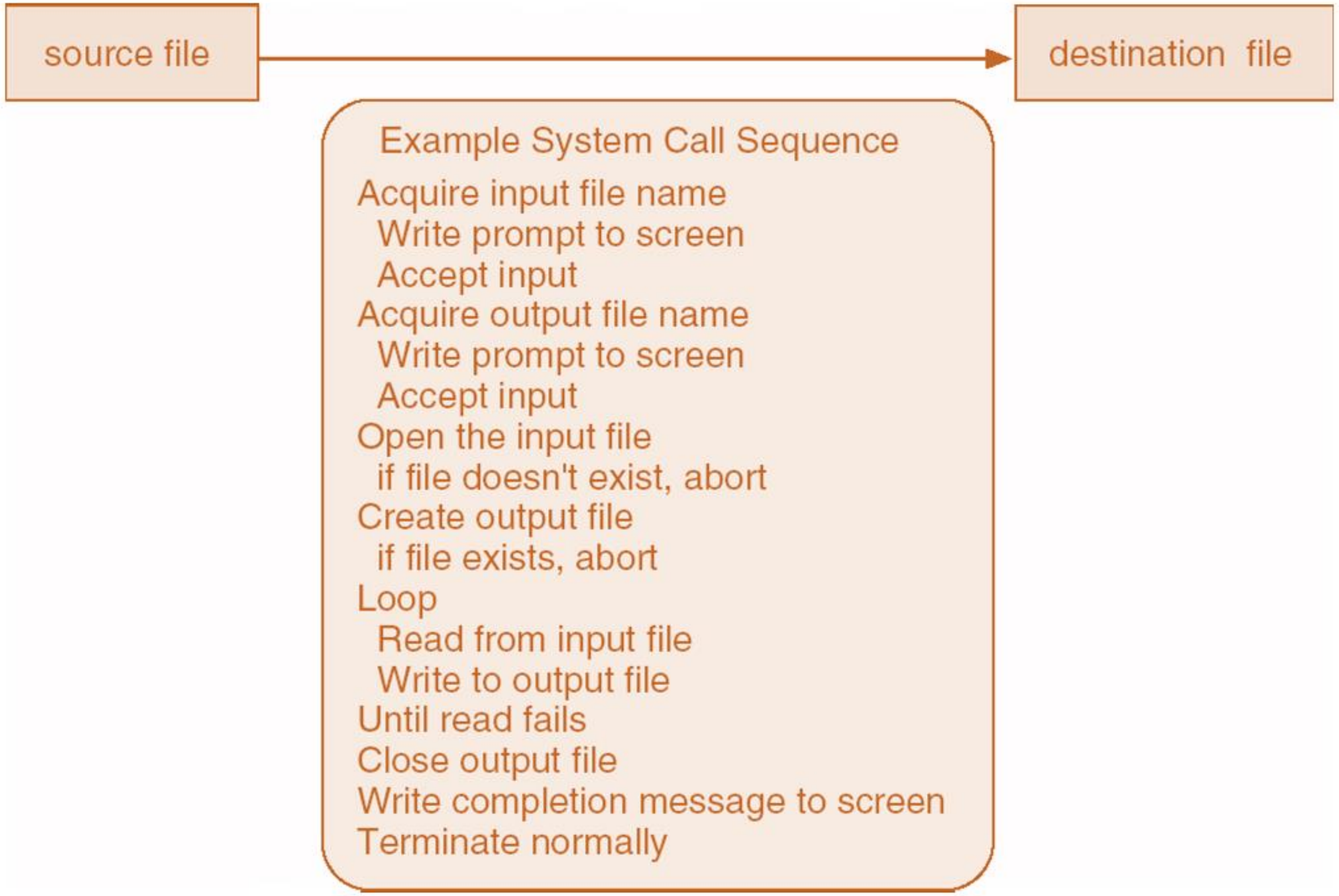
System Calls

- The system calls are the instruction set of the OS virtual processor.
- Programming interface to the services provided by the OS.
- Typically written in a high-level language (C or C++).
- Mostly accessed by programs via a high-level **Application Program Interface (API)** rather than direct system call use.
- 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).

*Portable Operating System Interface (POSIX)

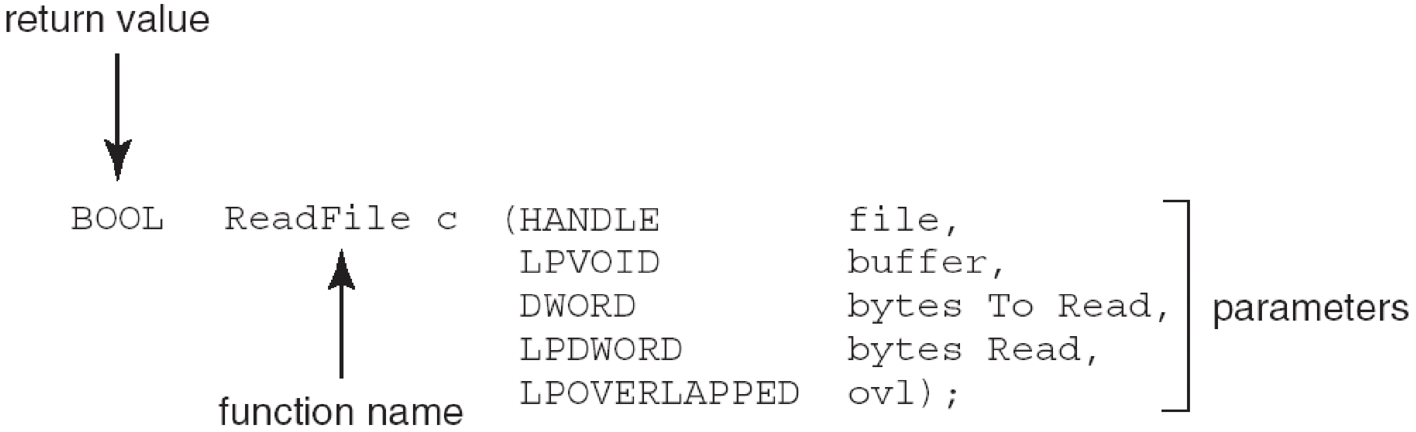
Example of System Calls

- System call sequence to copy the contents of one file to another file



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

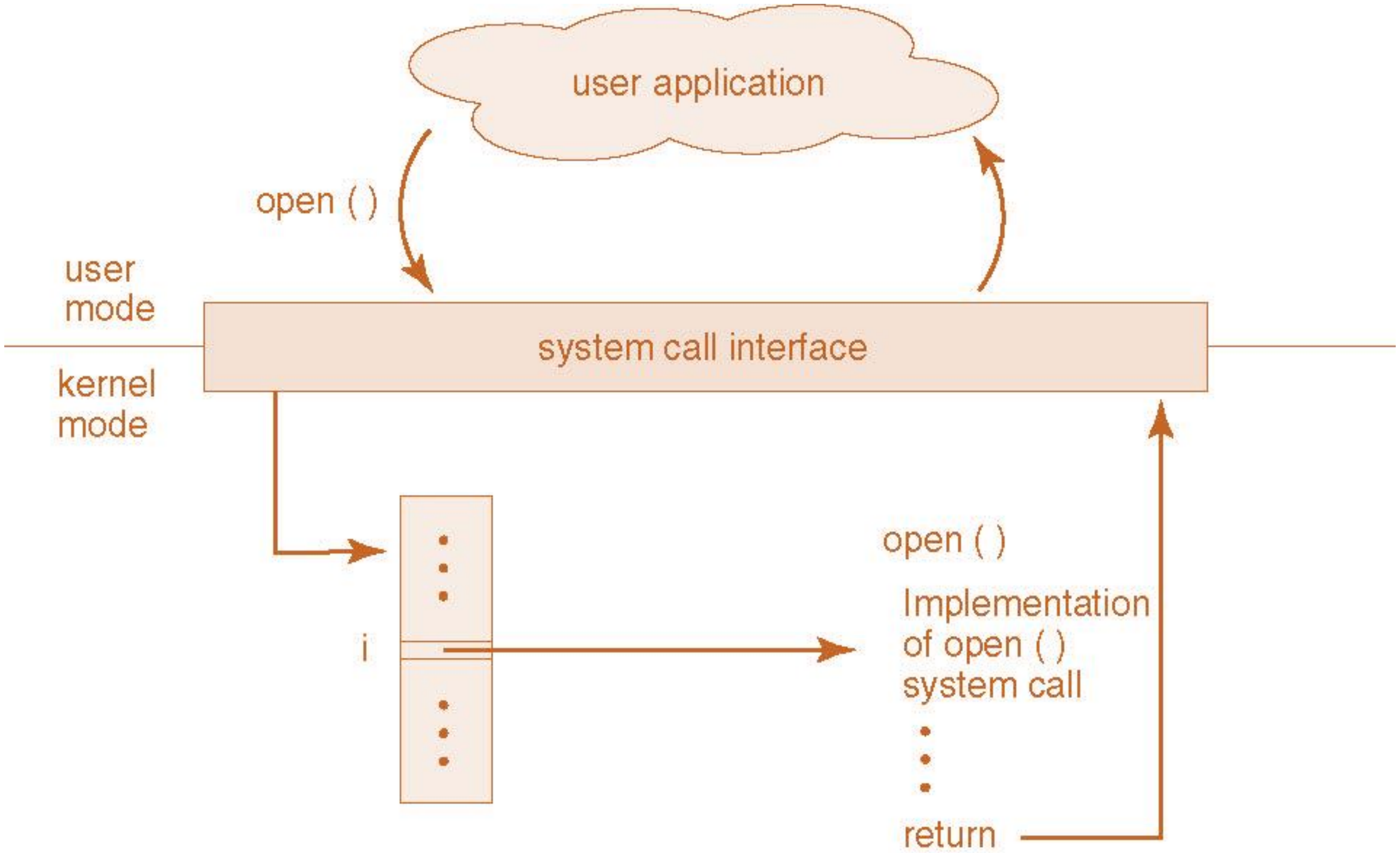
System Call Implementation

- 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

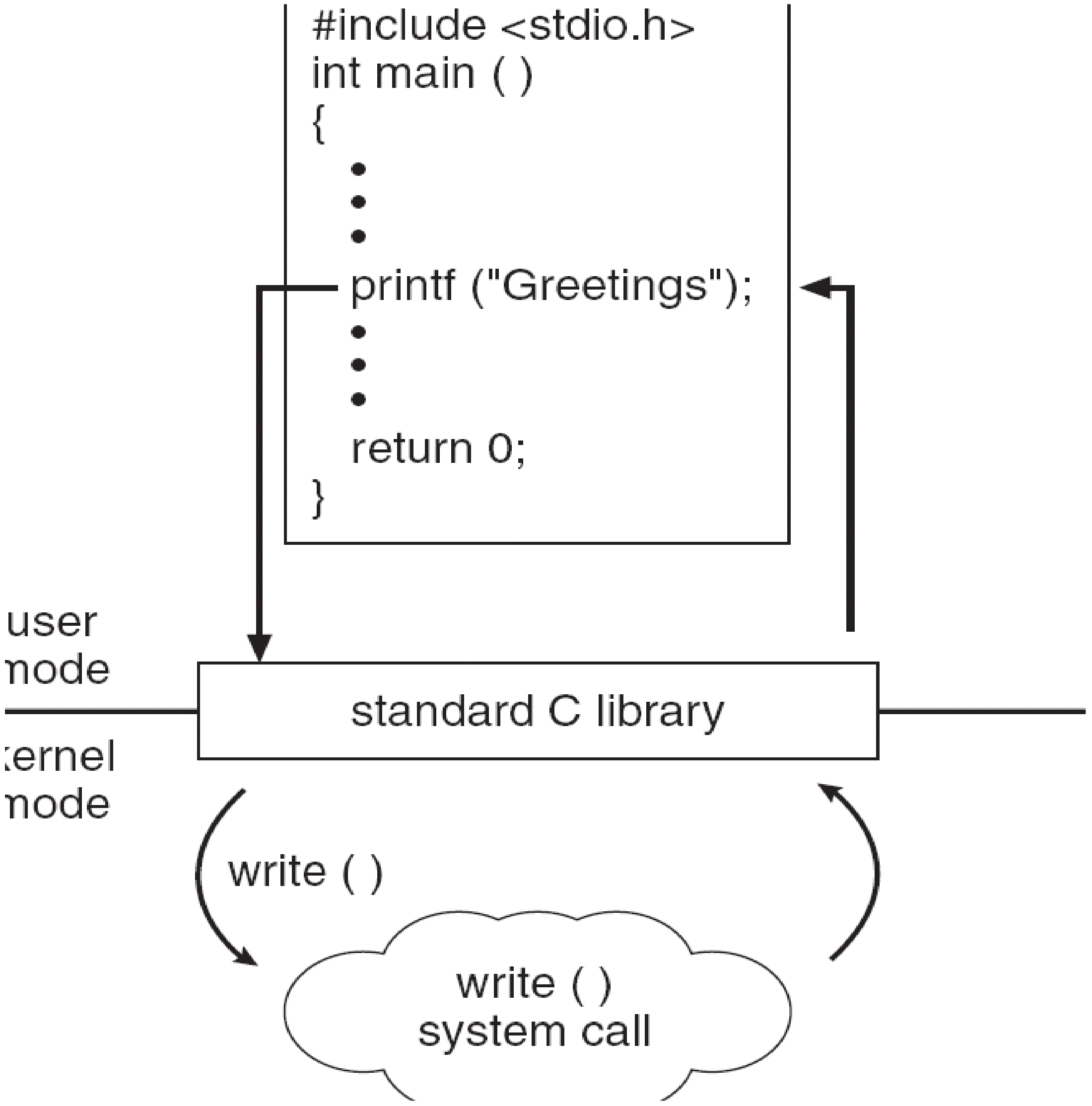
- The caller need know nothing about how the system call is implemented
  - Just needs to obey API and understand what OS will do as a result call
  - Most details of OS interface hidden from programmer by API
    - Managed by run-time support library (set of functions built into libraries included with compiler)

API – System Call – OS Relationship



Standard C Library Example

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

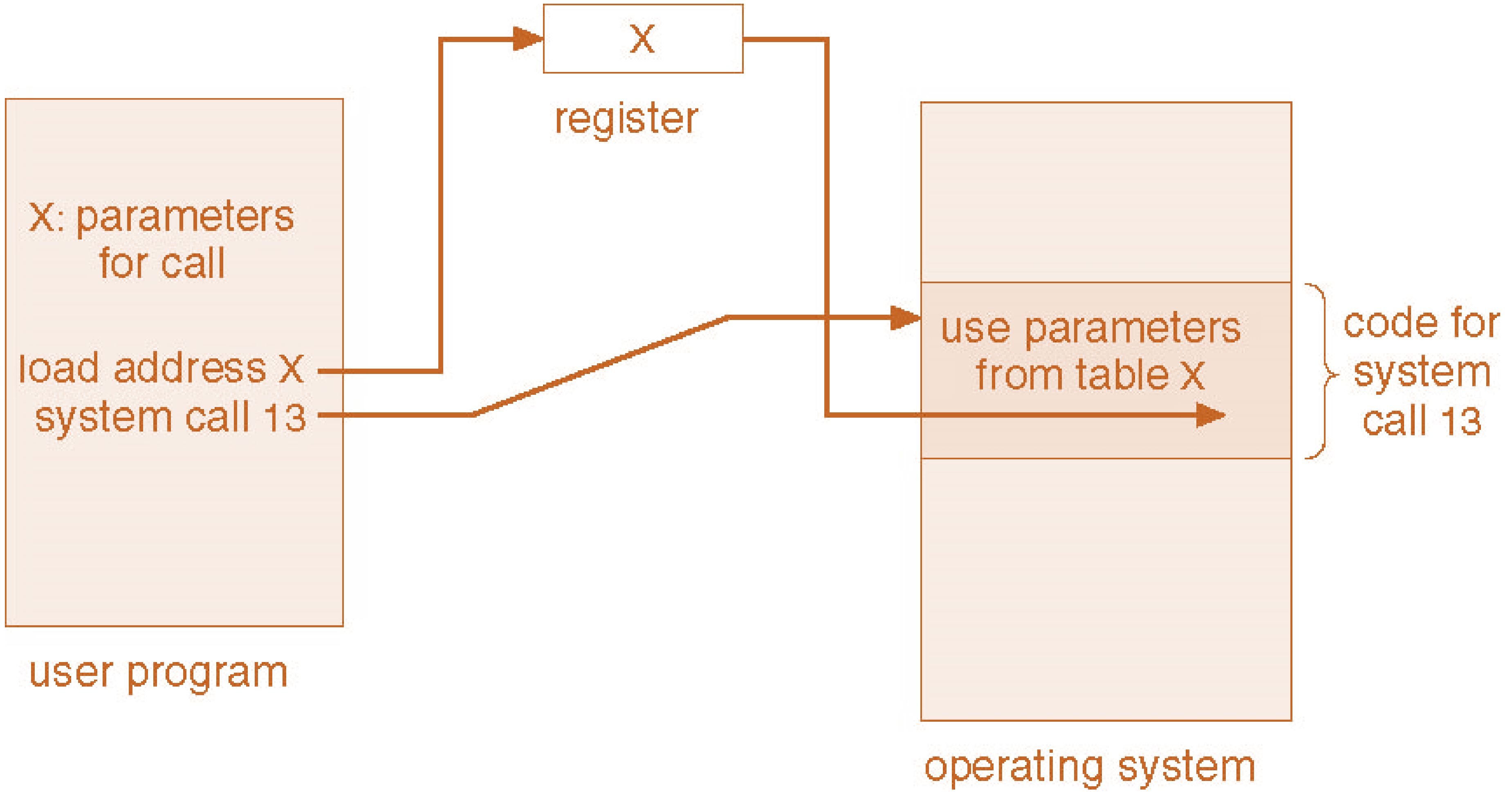


System Call Parameter Passing

- Often, more information is required than simply identity of desired system call
  - Exact type and amount of information vary according to OS and call

- Three general methods used to pass parameters to the OS
  - Simplest: pass the parameters in *registers*
    - In some cases, may be more parameters than registers
  - 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
  - Parameters placed, or *pushed,* onto the *stack* by the program and *popped* off the stack by the operating system
  - Block and stack methods do not limit the number or length of parameters being passed

Parameter Passing via Table



Types of System Calls

- Process control
  - end, abort
  - load, execute
  - create process, terminate process
  - get process attributes, set process attributes
  - wait for time
  - wait event, signal event
  - allocate and free memory

- File management
  - create file, delete file
  - open, close file
  - read, write, reposition
  - get and set file attributes

Types of System Calls

- Device management
  - request device, release device
  - read, write, reposition
  - get device attributes, set device attributes
  - logically attach or detach devices

- Information maintenance
  - get time or date, set time or date
  - get system data, set system data
  - get and set process, file, or device attributes

- Communications
  - create, delete communication connection
  - send, receive messages
  - transfer status information
  - attach and detach remote devices

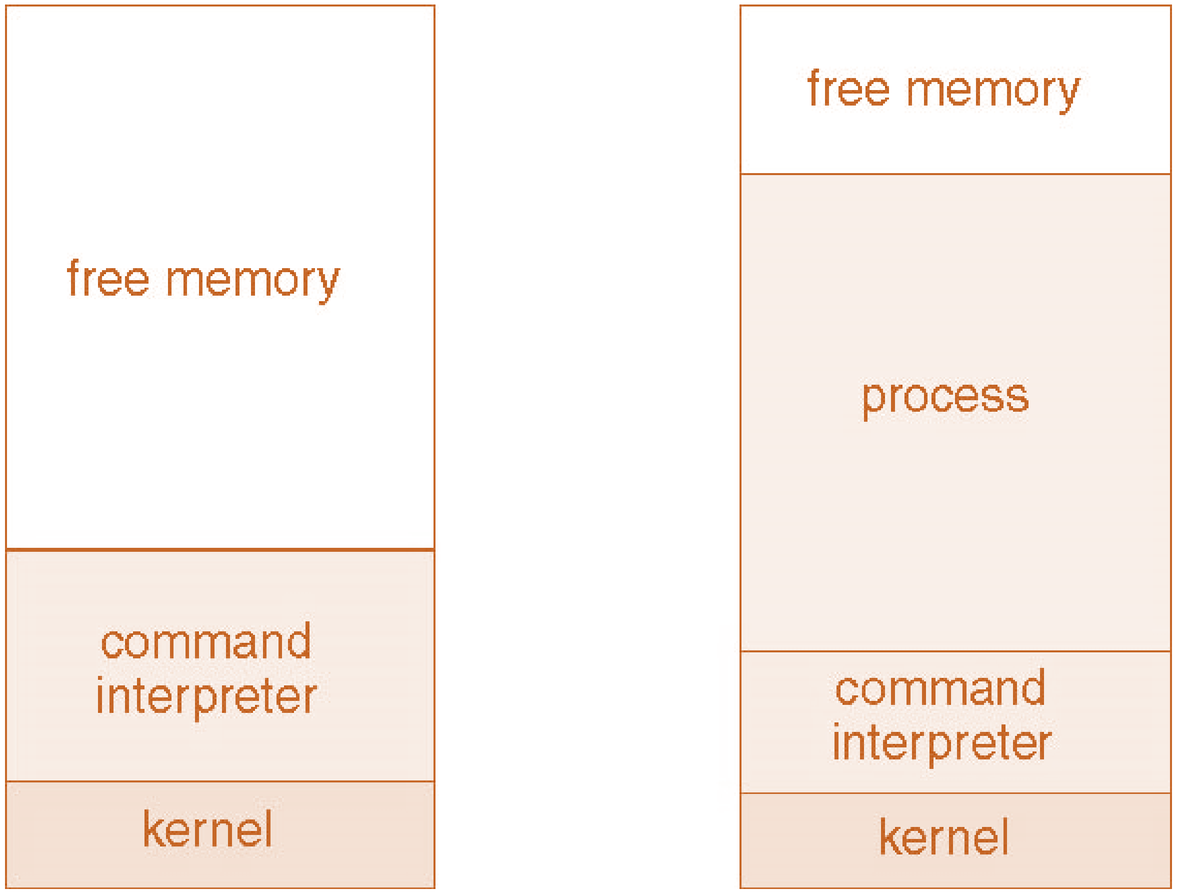
Examples of Windows and Unix System Calls

	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()

Types of System Calls: Process Control Example: MS-DOS

- Single-tasking
- Shell invoked when system booted
- Simple method to run program
  - No process created
- Single memory space
- Loads program into memory, overwriting all but the kernel
- Program exit -> shell reloaded

MS-DOS execution



(a)

(b)

THANK YOU

Outline

- System Structures I
  - Interrupt Handling
  - I/O Structure
- System Structures II
  - System Calls