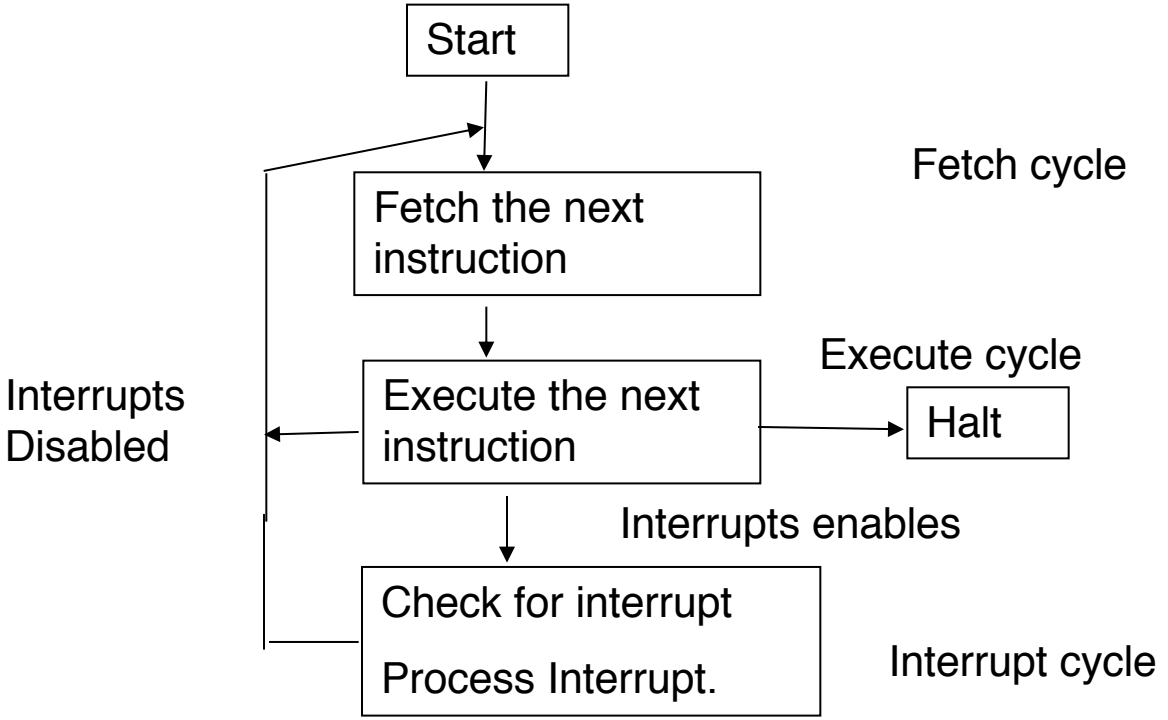
Operating Systems (CSE531)
Lecture # 04

Manish Shrivastava

LTRC, IIIT Hyderabad

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.

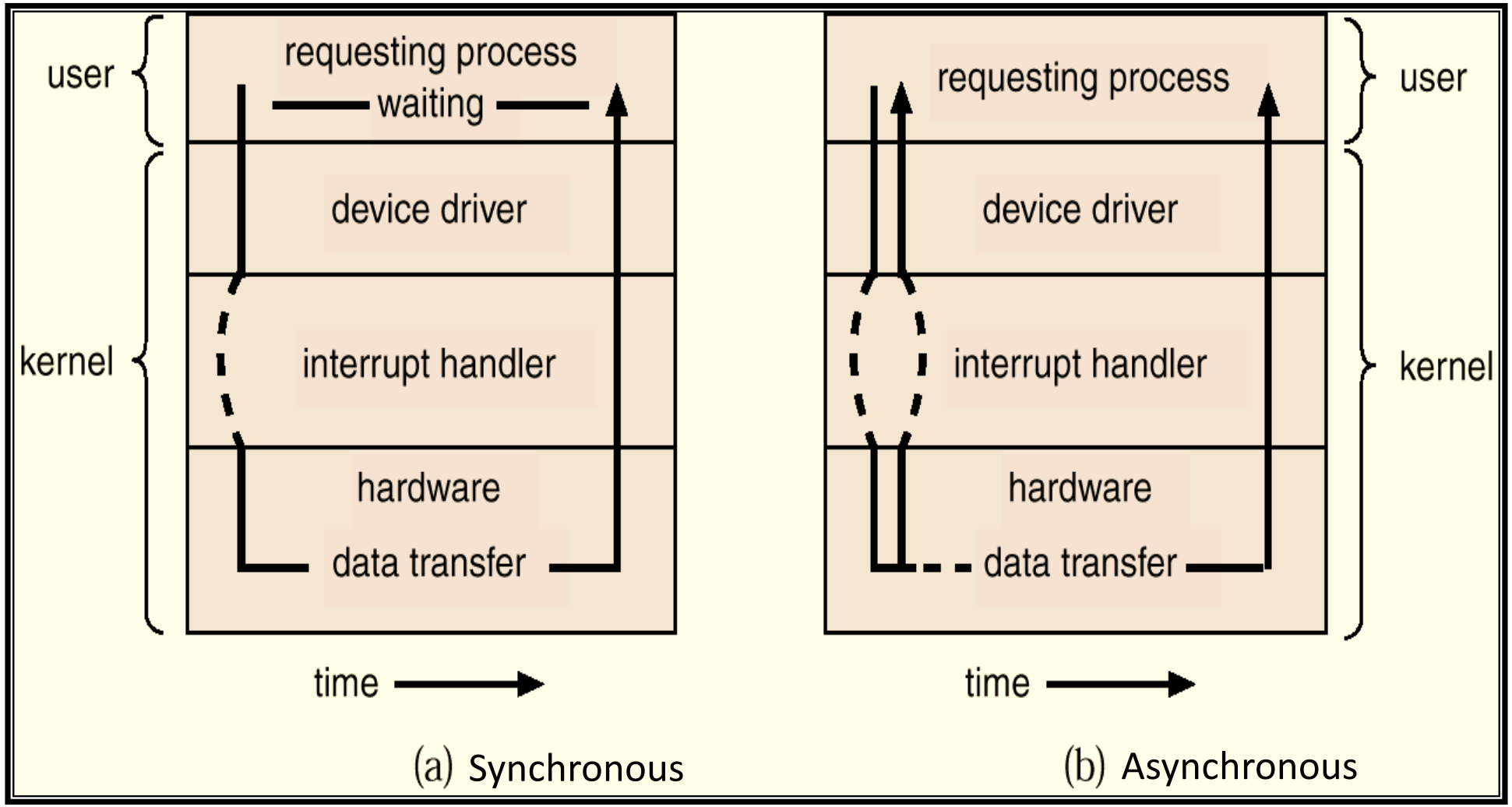
Interrupt Handling

- 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.)

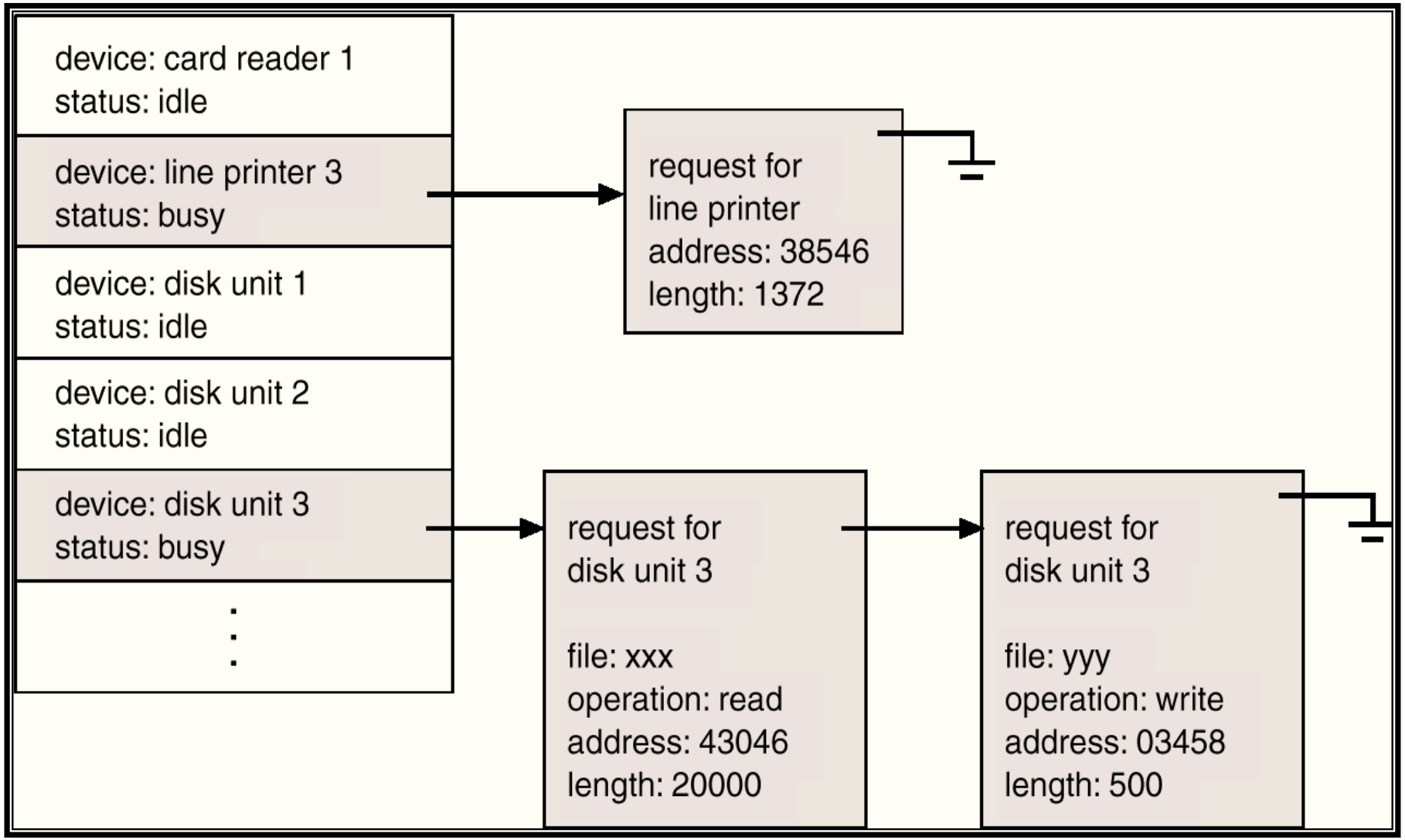
I/O Structure

- **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



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

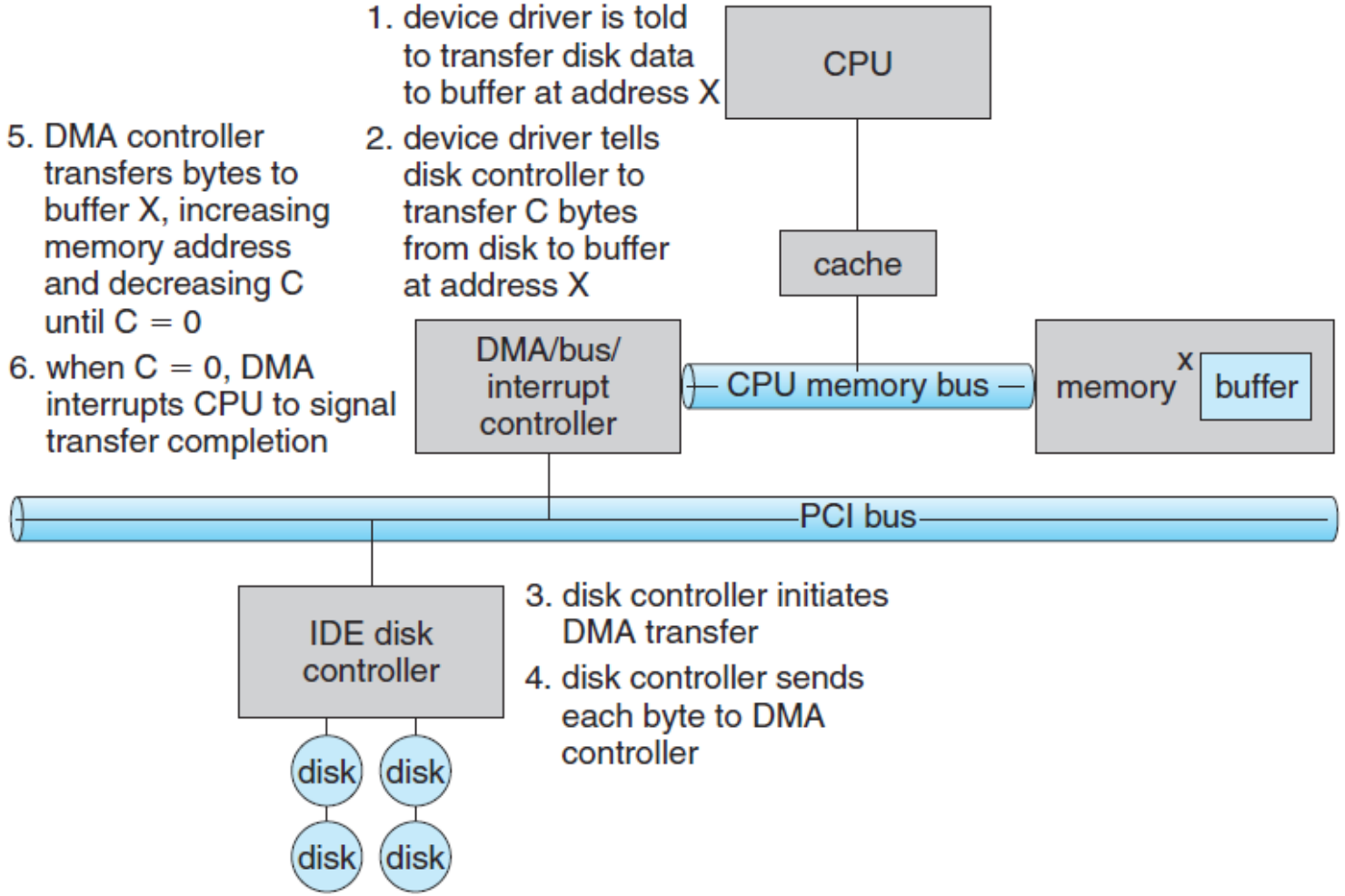
- System calls are implemented via ***software interrupts,*** a.k.a. ***traps.*** When a (library) program needs work performed in kernel mode, it sets command information and possibly data addresses in certain registers, and then raises a software interrupt.
- The completion of a disk read operation 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.

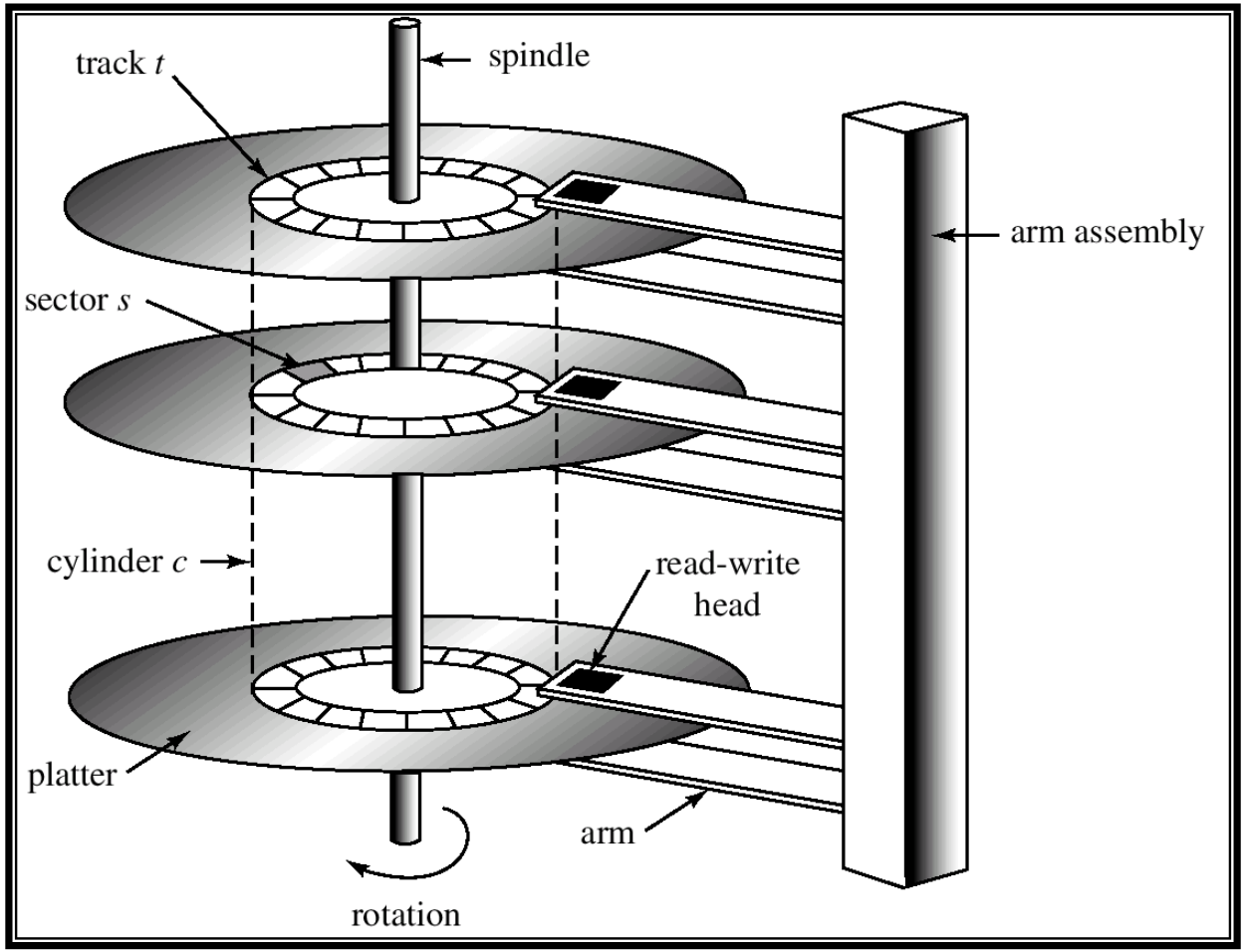
- Direct DMA access by user processes can speed up operations, but is generally forbidden by modern systems for security and protection reasons.

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

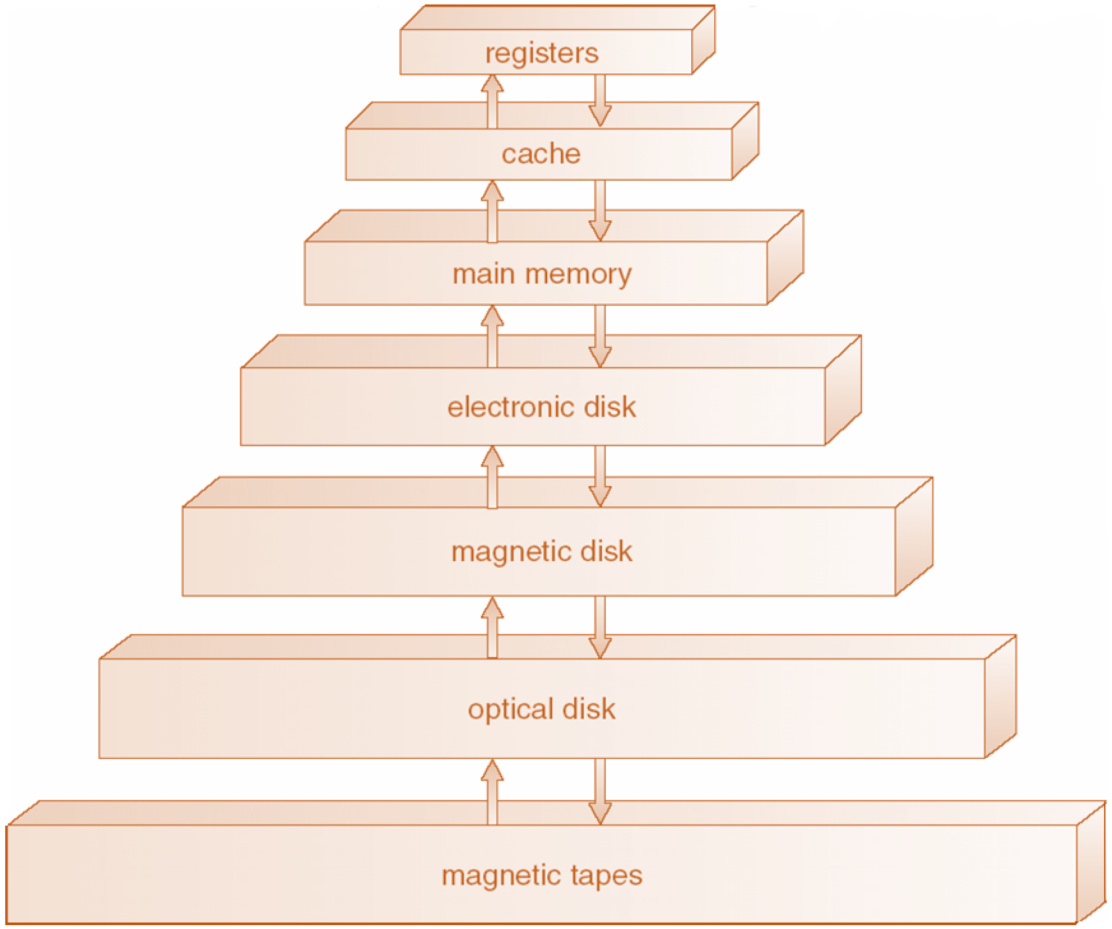


Storage Hierarchy

- 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

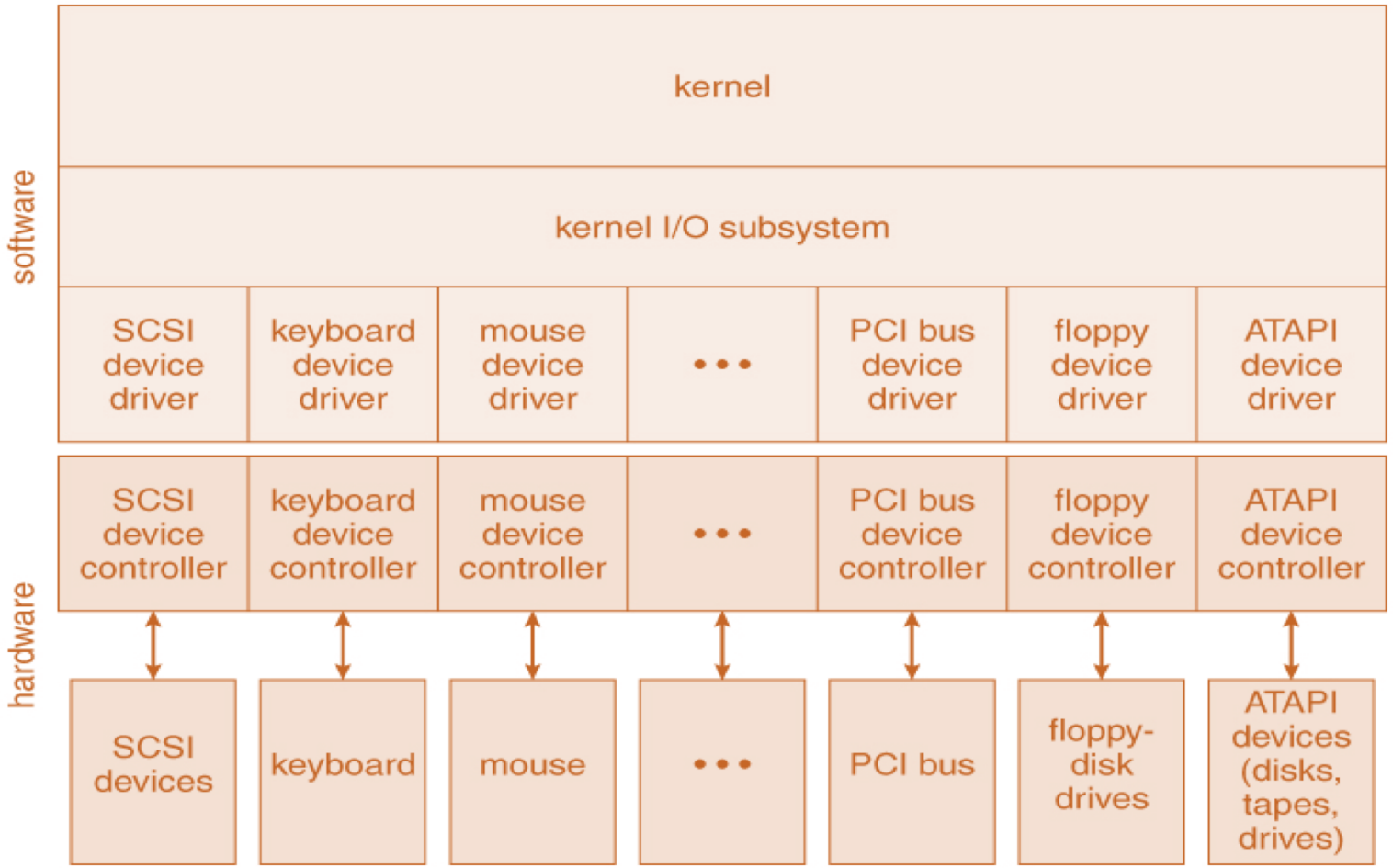
- Important principle, performed at many levels in a computer (in hardware, operating system, software)

- 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**

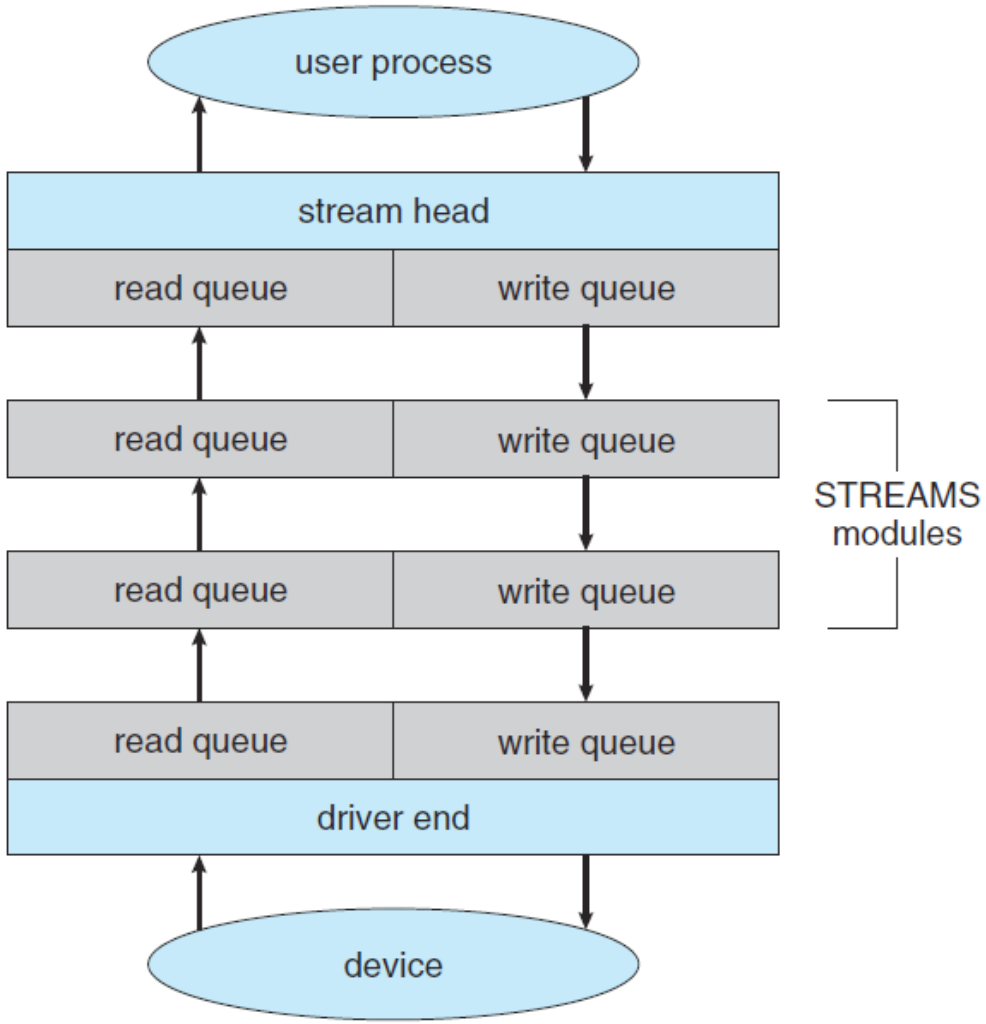
I/O request and hardware

- 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.

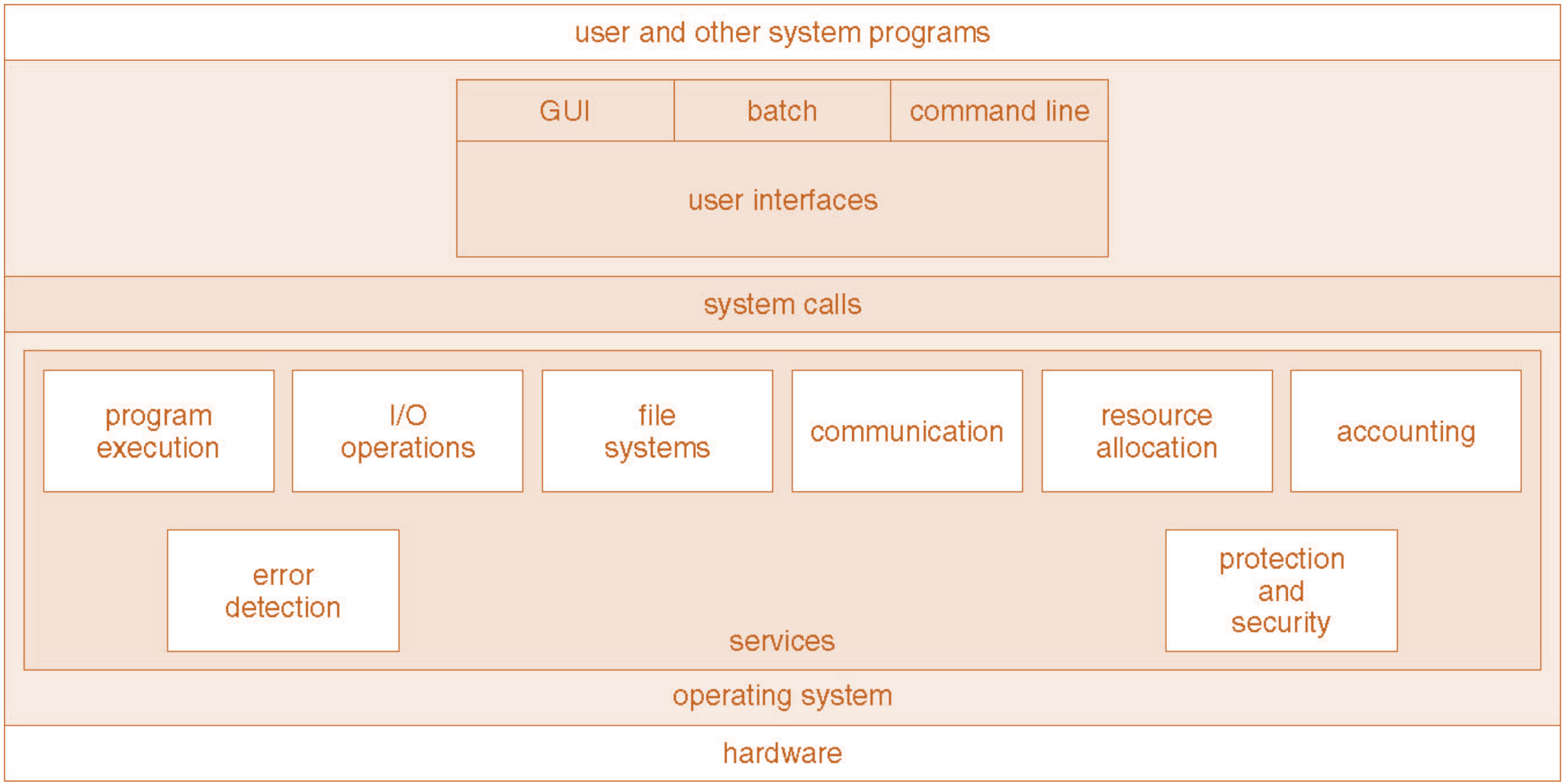
Unix Streams



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



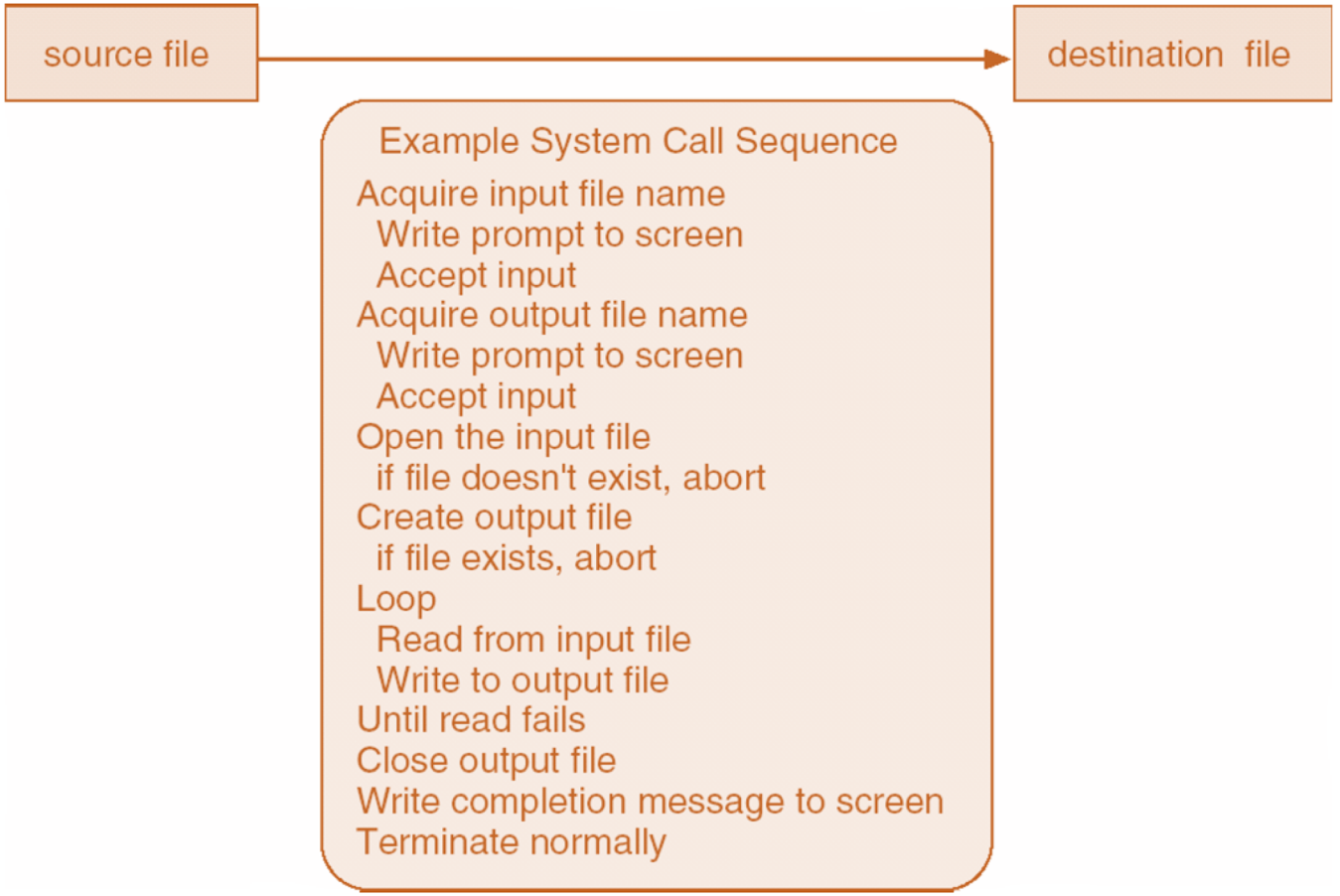
System Calls

- 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)

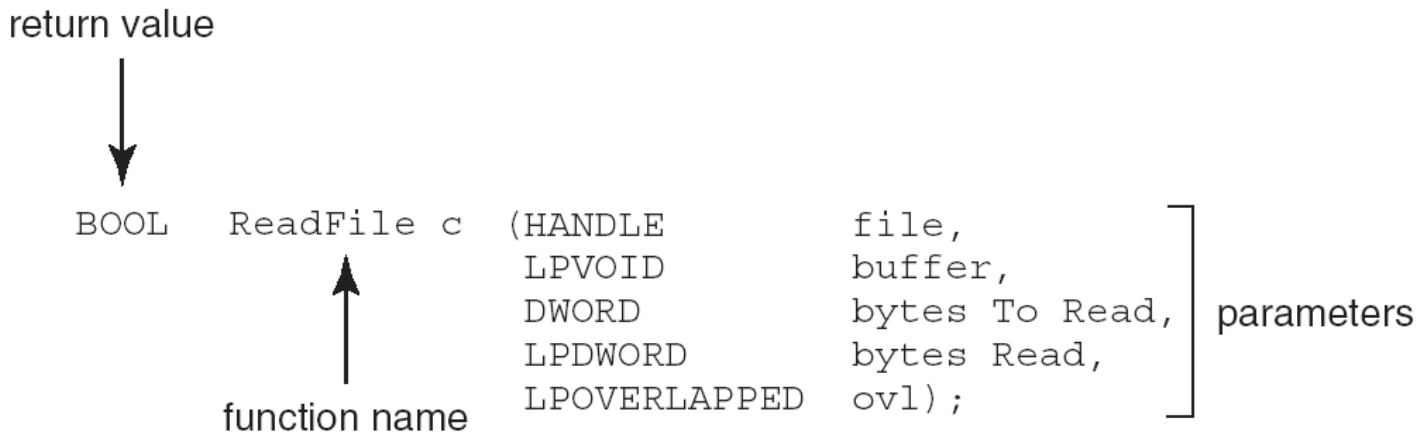
Example of System Calls

- 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

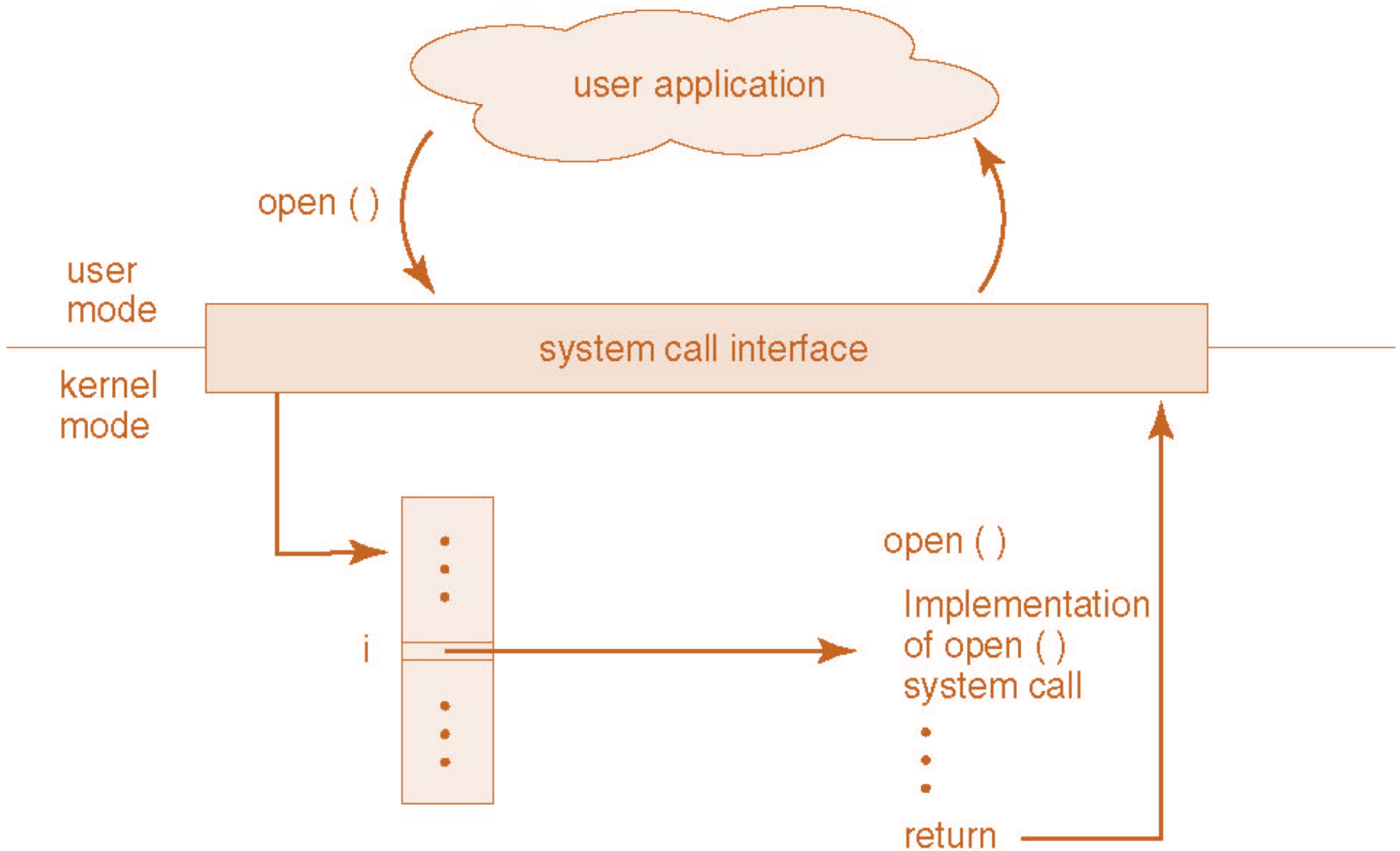
System Call Implementation

- 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

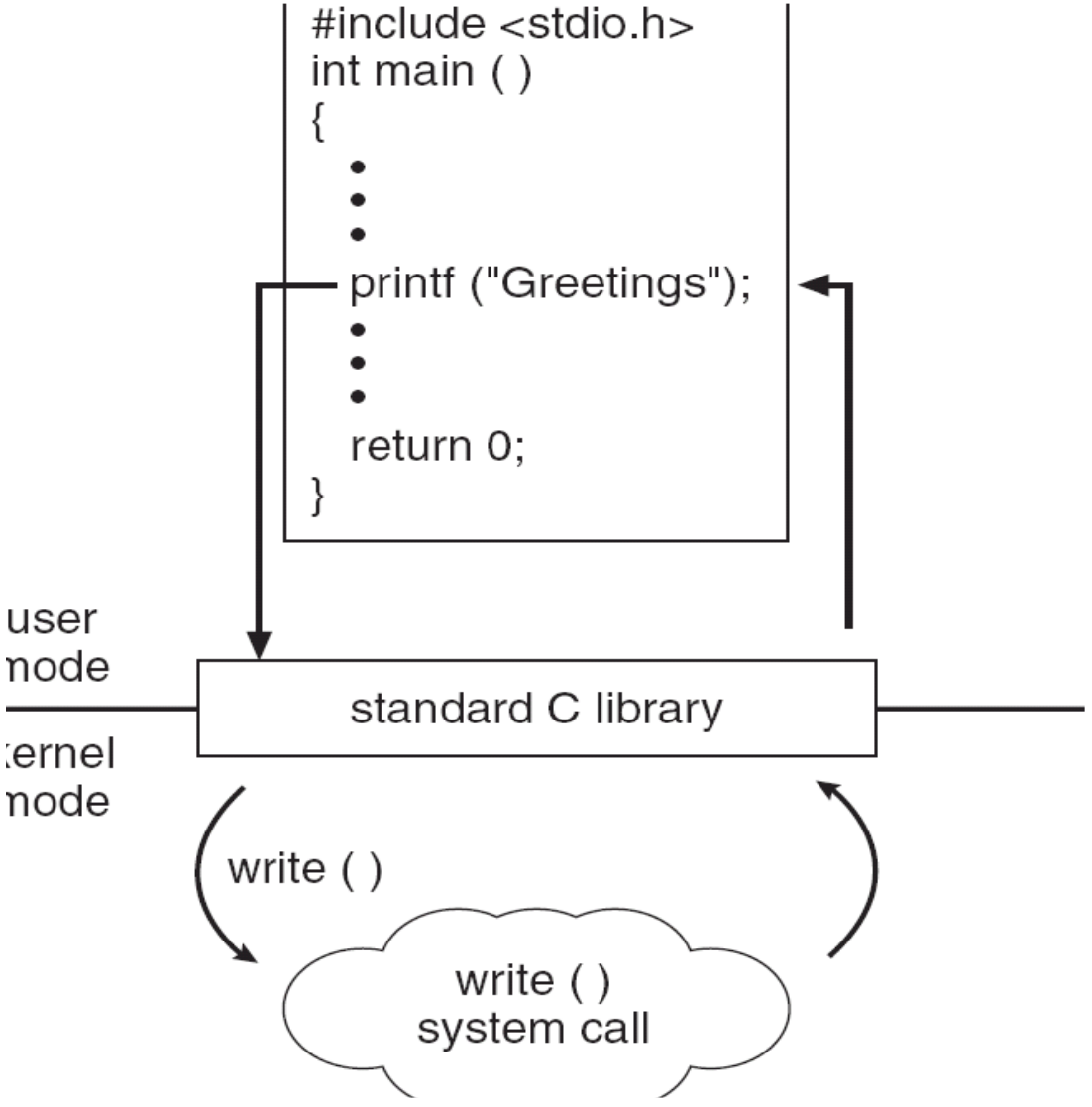
- The caller need know nothing about how the system call is implemented
  - Just needs to obey API and understand what OS will do as a result call
  - Most details of OS interface hidden from programmer by API
    - Managed by run-time support library (set of functions built into libraries included with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

API – System Call – OS Relationship



Standard C Library Example

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

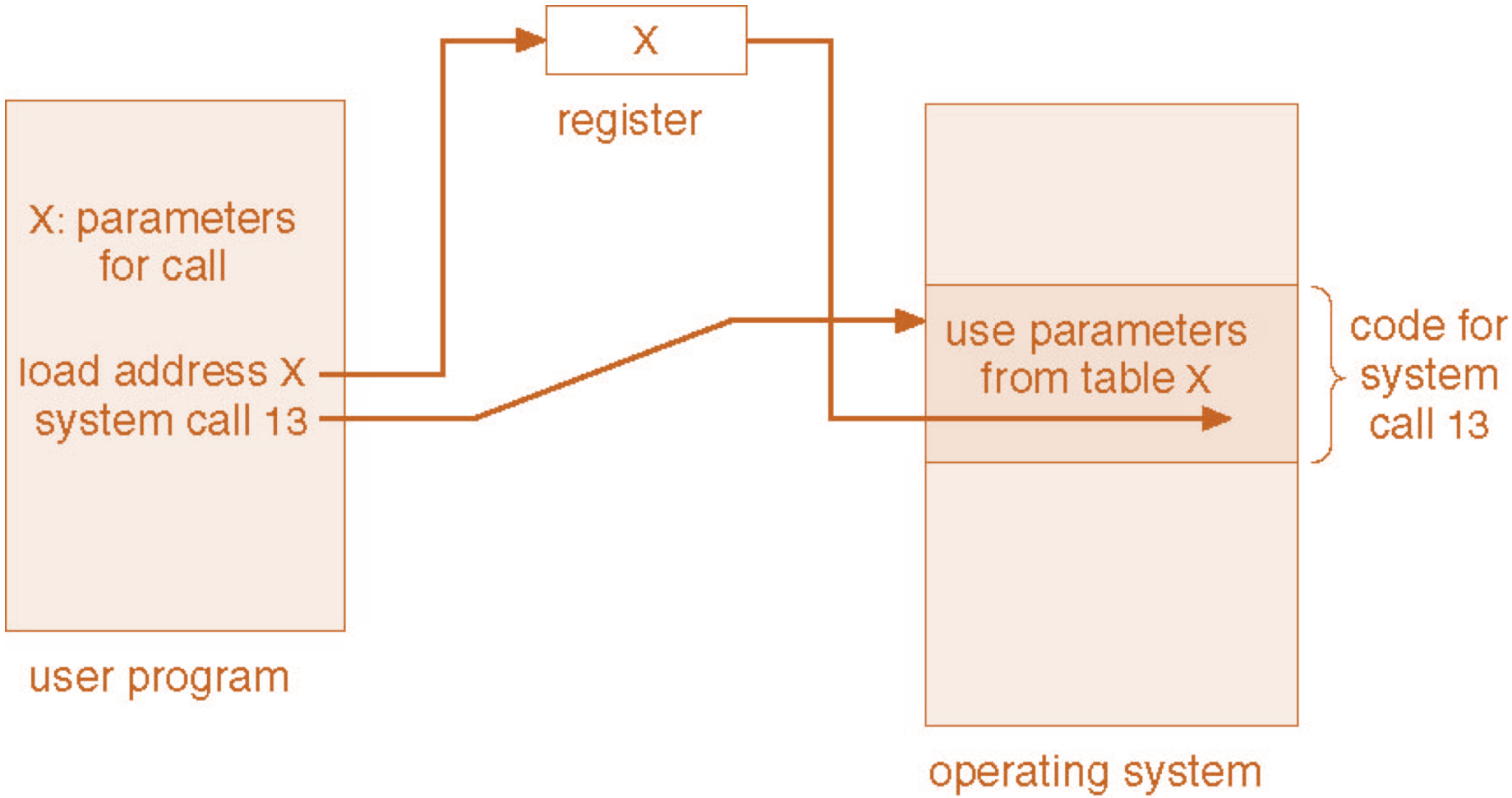


System Call Parameter Passing

- 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

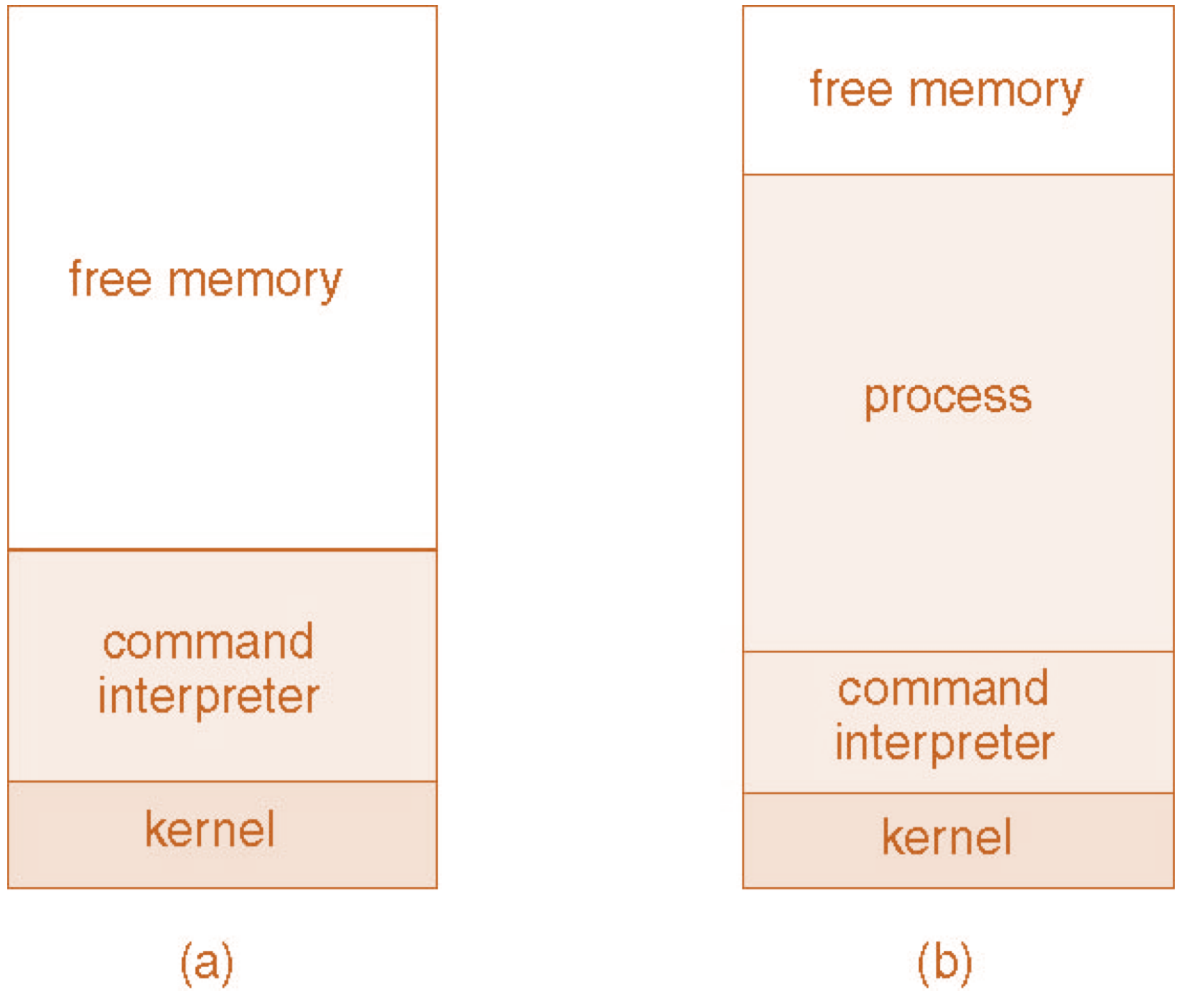
Examples of Windows and Unix System Calls

	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



THANK YOU