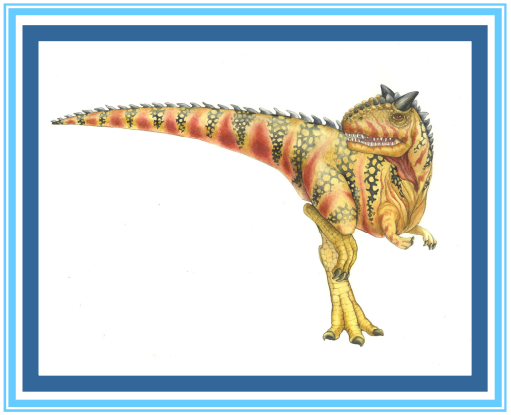
UNIVERSIDAD TECNICA
FEDERICO SANTA MARIA

DEPARTAMENTO DE
ELECTRONICA

Teoría de Sistemas Operativos

ELO 321 – Clase 02

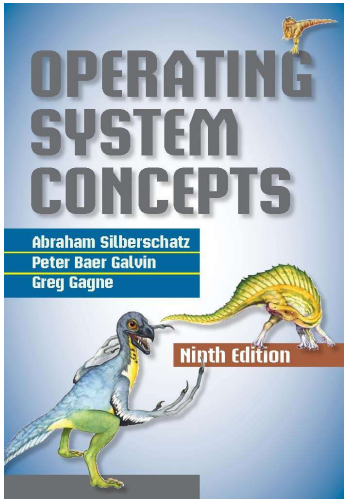
Dr. Ioannis Vourkas
Oficina B320, Email: ioannis.vourkas@usm.cl



10 de septiembre, 2020, Valparaíso, Chile

Course Information

- **Evaluation**
  - **2** Certamenes (75%) y **2** tareas (homework) (25%)…
- **Course book**: A. Silberschatz, P. Galvin, y G. Gagne (2012). *Operating System Concepts*, Wiley, 9th Ed. (http://www.os-book.com)



Process Management

- A process is a program in execution. It is a unit of work within the system. Program is a ***passive entity***, process is an ***active entity***.
- Process needs resources to accomplish its task
  - CPU, memory, I/O, files
  - Initialization data
- These resources are either given to the process when it is created or allocated to it while it is running.
  - When the process terminates, the operating system will reclaim any reusable resources.
- Single-threaded process has one **program counter** specifying location of next instruction to execute
  - Process executes instructions sequentially, one at a time, until completion
- Multi-threaded process has one **program counter** per thread
- Typically system has many processes, **running concurrently** on one or more CPUs
  - **Concurrency** by multiplexing the CPUs among the processes / threads

Process Management

- The operating system is **responsible for** the following activities in connection with process management:
  - Scheduling processes and threads on the CPUs
  - Creating and deleting both user and system processes
  - Suspending and resuming processes
  - Providing mechanisms for process synchronization
  - Providing mechanisms for process communication

Memory Management

- Main memory is a large array of bytes. *Each byte has its own address*.
- The main memory is generally **the only large storage device that the CPU is able to address and access directly**.
  - For example, for the CPU to process data from disk, those data must first be transferred to main memory by I/O calls.
- To execute a program all (or part) of the instructions must be in memory
- All (or part) of the data that is needed by the program must be in memory.
- **Memory management** determines what is in memory and when
- Many different memory management schemes are used
  - Each algorithm requires its own hardware support

Memory Management

- The operating system is **responsible for** the following activities in connection with memory management:
  - Keeping track of which parts of memory are currently being used and by whom
  - Deciding which processes (or parts thereof) and data to move into and out of memory
  - Allocating and de-allocating memory space as needed

Storage Management

- To make the computer system convenient for users, the operating system provides a uniform, **logical view of information storage**.
- The operating system *abstracts from the physical properties of its storage devices* to define *a logical storage unit*, **the file**.
  - Each storage medium is controlled by device (i.e. disk drive)
  - Each storage medium has its own characteristics and physical organization.
    - Varying properties include access *speed, capacity, data-transfer rate, access method* (sequential or random)
- The concept of a file is an extremely general one.
  - The operating system implements the abstract concept of a file by *managing mass-storage media*
- **File-System management**
  - Files usually organized into directories
  - Access control on most systems to determine who can access what

Mass-Storage Management

- As we have already seen, the computer system must provide **secondary storage** to back up main memory.
- Usually disks are used to store data that does not fit in main memory or data that must be kept *for a "long" period of time*
  - Proper mass-storage management is of central importance
- Most programs are stored on a disk until loaded into memory.
  - They then use the disk as both the source and destination of their processing.
- Because secondary storage is used frequently, it must be used efficiently.
  - Entire **speed of computer operation** hinges on disk subsystem and its algorithms

Storage Hierarchy and Caching

- **Caching** (*storing parts of data in faster storage for performance*) is an important principle of computer systems

- Here's **how it works**:
  - Information is normally kept in some storage system (such as *main memory*).
  - As it is used, it is *copied into a faster storage system*—**the cache**—on a temporary basis.
  - When we need a particular piece of information, we first check whether it is in the cache.
    - If it is, we use the information directly from the cache.
    - If it is not, we use the information from the source, putting a copy in the cache.

Storage Hierarchy and Caching

- **Caching** (storing parts of data in faster storage for performance) is an important principle of computer systems
  - Hardware-only caches are outside the control of the operating system.
  - In a hierarchical storage structure, the **same data** may appear in different levels of the storage system.
    - Main memory can be viewed as a fast cache for secondary storage

Storage Hierarchy and Caching

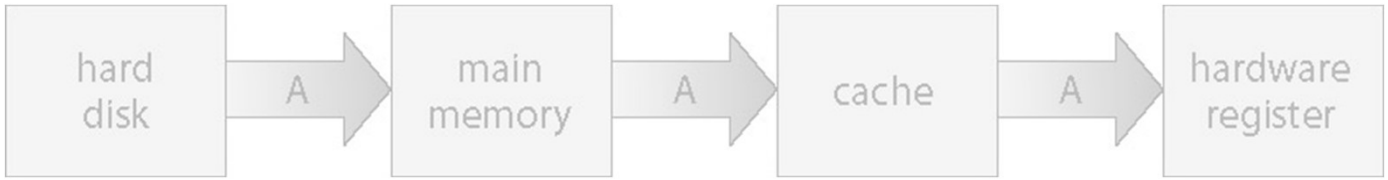
Level	1	2	3	4	5
Name	registers	cache	main memory	solid state disk	magnetic disk
Typical size	< 1 KB	< 16MB	< 64GB	< 1 TB	< 10 TB
Implementation technology	custom memory with multiple ports CMOS	on-chip or off-chip CMOS SRAM	CMOS SRAM	flash memory	magnetic disk
Access time (ns)	0.25 - 0.5	0.5 - 25	80 - 250	25,000 - 50,000	5,000,000
Bandwidth (MB/sec)	20,000 - 100,000	5,000 - 10,000	1,000 - 5,000	500	20 - 150
Managed by	compiler	hardware	operating system	operating system	operating system
Backed by	cache	main memory	disk	disk	disk or tape

Performance of different levels of storage

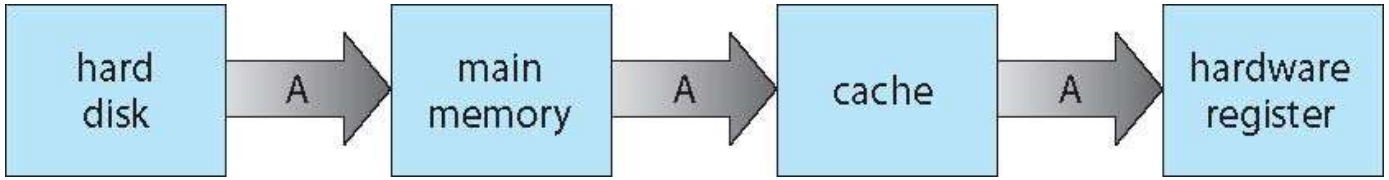
Migration of data “A” from Disk to Register

- The movement of information between levels of a storage hierarchy may be either **explicit** or **implicit**
  - data transfer from cache to CPU and registers is usually a hardware function, with no operating-system intervention.
  - transfer of data from disk to memory is usually controlled by the operating system.

- In a hierarchical storage structure, **the same data may appear in different levels of the storage system**.
  - *For example, suppose that an integer A that is to be incremented by 1, is located in file B in hard disk*

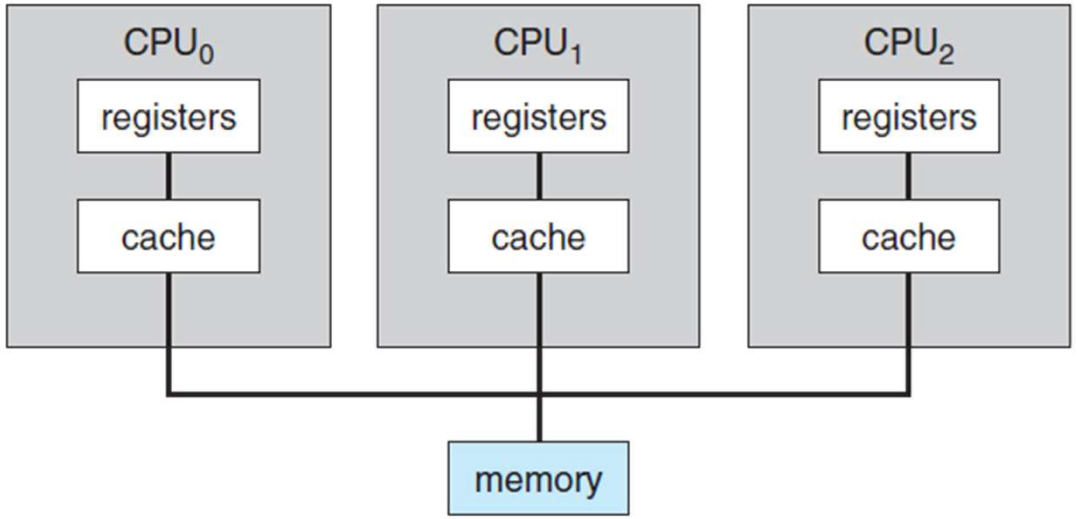


Migration of data “A” from Disk to Register



- The increment operation proceeds by first issuing an I/O operation to copy the disk block on which A resides to main memory.
- This operation is followed by copying A to the cache and to an internal register.
  - Thus, the **copy of A appears in several places**
- In **multitasking** environments (CPU is switched among various processes) care must be taken to ensure that each of these processes will obtain the most recent value, no matter where it is stored in the storage hierarchy

Migration of data “A” from Disk to Register



- In a **multiprocessor** environment, a copy of A may exist simultaneously in several caches.
  - It must provide **cache coherency** in hardware, such that **all CPUs have the most recent value** in their cache → HARDWARE issue

I/O Subsystem

- One purpose of OS is to hide peculiarities of hardware devices from the user
- The I/O subsystem consists of several components:
  - **Memory management** of I/O including:
    - buffering (storing data temporarily while it is being transferred)
    - caching (*storing parts of data in faster storage for performance*), etc.
  - ***Drivers*** for specific hardware devices
    - *Only the device driver knows the peculiarities of the specific device to which it is assigned*

Protection and Security

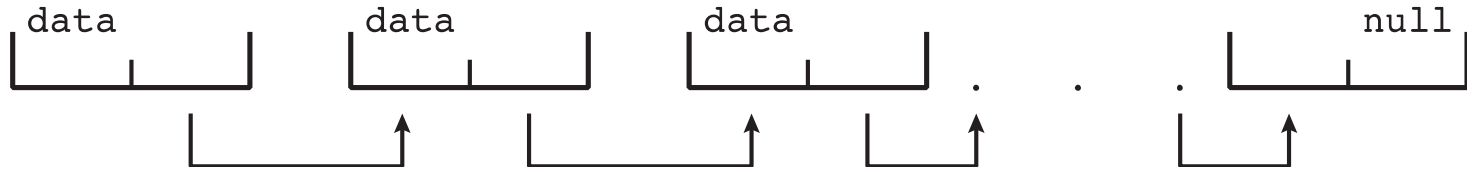
- **Protection** – any mechanism for controlling access of processes or users to resources defined by the OS
  - For example, *memory-addressing hardware* ensures that a process can execute only within its own address space.
  - However, a system can have adequate protection but still be prone to failure and allow inappropriate access
- **Security** – defense of the system against internal and external attacks
- Systems generally first distinguish among users, to determine who can do what
  - Most operating systems maintain a list of user names and associated user identifiers (**user IDs**).
    - These numerical IDs are unique, **one per user**.
  - User ID then *associated with all files, processes of that user to determine access control*
  - Group identifier (**group ID**) allows set of users to be defined and controls managed, then also associated with each process, file
  - **Privilege escalation** allows user to change to **effective ID** with more rights

Kernel Data Structures

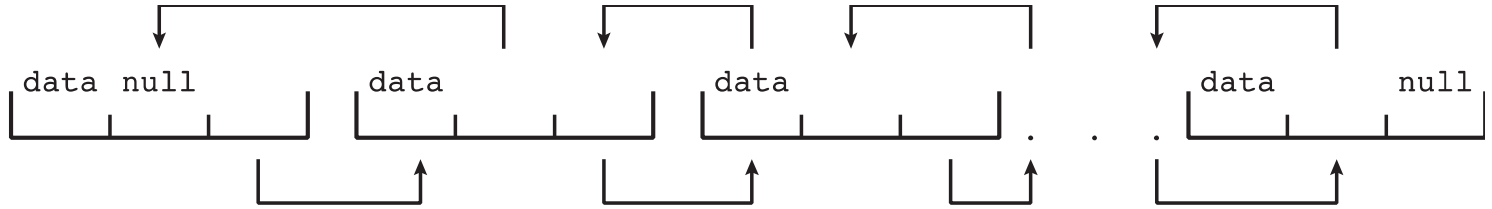
- We turn next to a topic central to operating-system implementation: the way data are structured in the system.
  - We briefly describe several fundamental data structures used extensively in operating systems.
- **Array**:
  - An array is a data structure in which each element can be accessed **directly**. For example, main memory is constructed as an array.
    - But what about storing an item *whose size may vary*???
    - In such situations, arrays give way to other data structures!
  - After arrays, **lists** are the most fundamental data structures in computer science.
    - *the items in a list must be accessed in a particular order*.
    - A list represents a collection of data values ***as a sequence***.

Kernel Data Structures

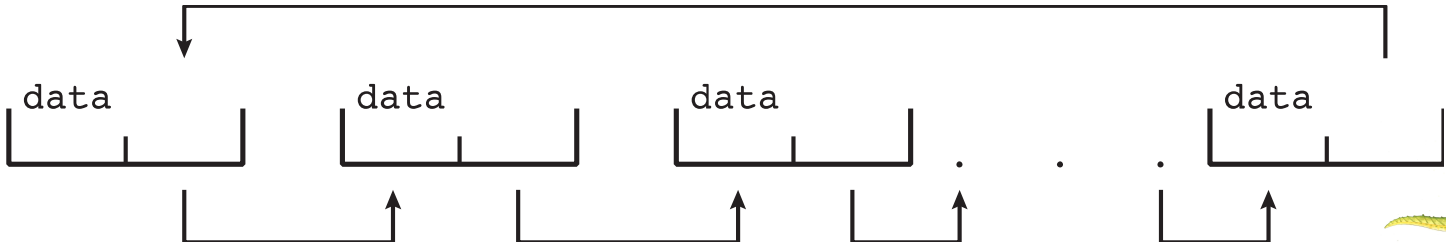
- Linked lists accommodate items of varying sizes and allow easy insertion and deletion of items.
- ***Singly linked list***



- ***Doubly linked list***

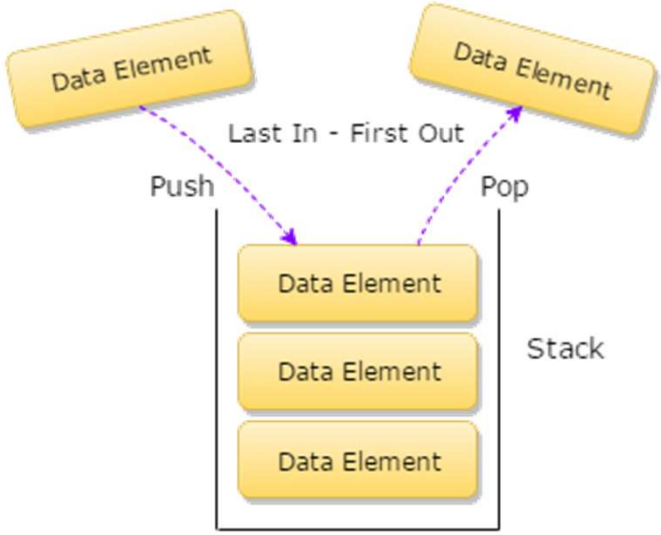


- ***Circular linked list***



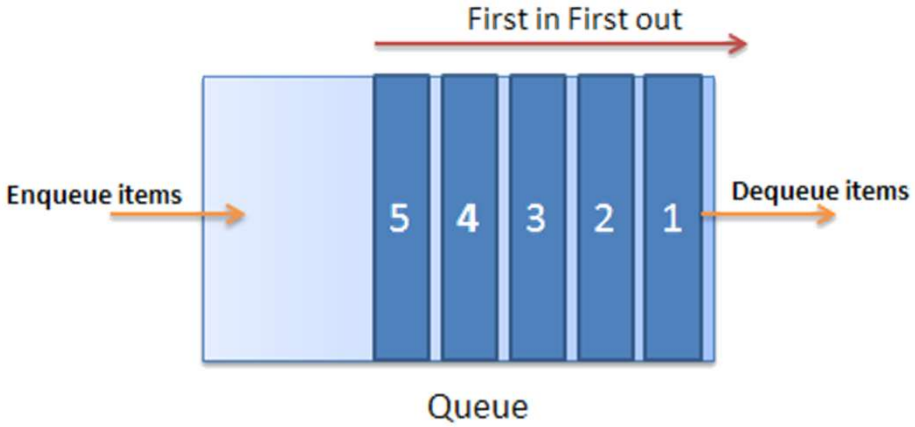
Kernel Data Structures

- ***Stack* (*pila*)**
  - A stack is a sequentially ordered data structure that uses the last in, first out (**LIFO**) principle, meaning that the last item placed onto a stack is the first item removed.
  - The operations for inserting and removing items from a stack are known as *push* and *pop*, respectively.
  - An operating system often **uses a stack when invoking function calls**.

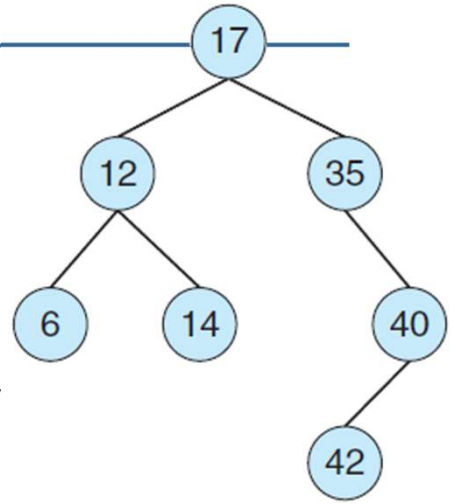


Kernel Data Structures

- ***Queue* (cola)**
  - A queue, is a sequentially ordered data structure that uses the first in, first out (**FIFO**) principle: items are removed from a queue in the order in which they were inserted.
    - Queues are quite common in operating systems
    - ***jobs that are sent to a printer*** *are typically printed in the order in which they were submitted.*
    - *tasks that are* ***waiting to be run on an available CPU*** *are often organized in queues*

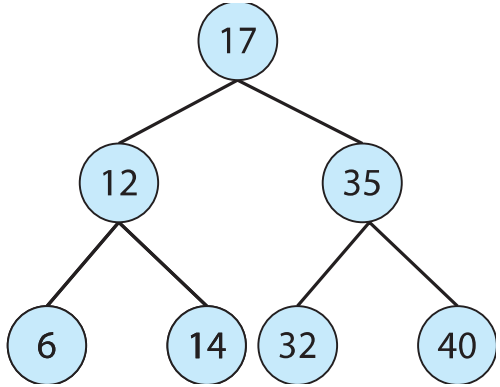


Kernel Data Structures



Binary search tree

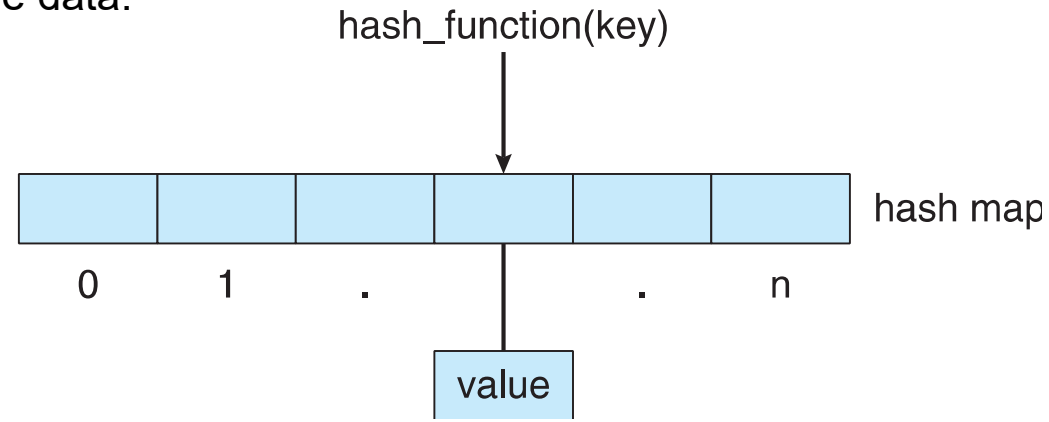
- A tree is a data structure that can be used *to represent data hierarchically*. Data values in a tree structure are linked through **parent–child relationships**.
- In a general tree, a parent may have an unlimited number of children. In a binary tree, a parent may have at most two children, which we term the *left* child and the *right* child.
- A **binary search tree** additionally requires an ordering between the parent's two children in which left_child <= right_child.
  - When we search for an item in a binary search tree, **the worst-case performance is O(n)** (consider how this can occur).
  - However, a **balanced binary search tree** containing *n* items, has at most log*n* levels, thus ensuring **worst-case performance of O(log*n*)**.



Kernel Data Structures

Hash function

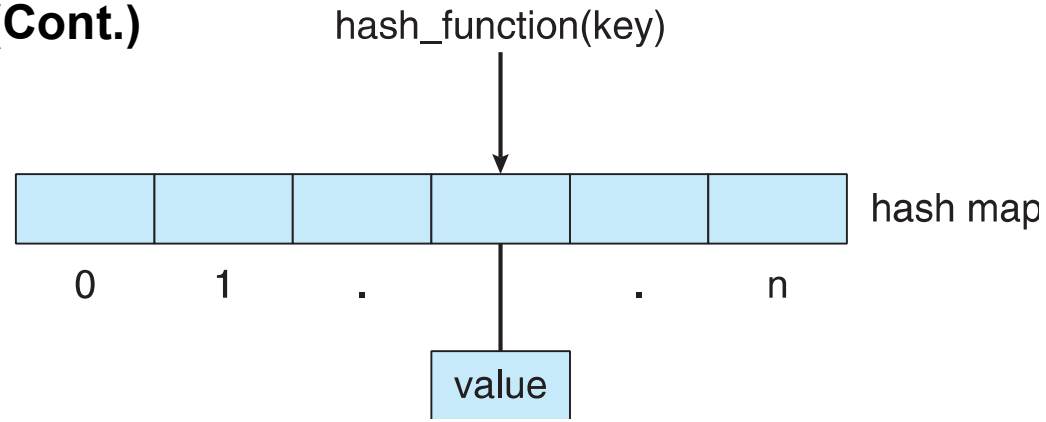
- A Hash function takes data as its input, performs a numeric operation on this data, and returns a numeric value.
  - This numeric value **can then be used as an index into an array** to quickly retrieve the data.



- Using a hash function for retrieving data from table can be as good as **O(1) in the worst case**.
- However, many inputs can result in the same output value—that is, they can link to the same table location.
  - We can accommodate this hash collision by having a linked list at that table location that contains all of the items with the same hash value.

Kernel Data Structures

Hash function (Cont.)



Example of use:

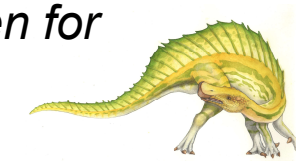
- One use of a hash function is **to implement a hash map**, which associates (or maps) [key:value] pairs
- Suppose that a user name is mapped to a password.
  - The hash function is applied to the user name to retrieve the password.
  - The retrieved password is compared with the password entered by the user for authentication
- Linux Data structures defined in: <linux/list.h>, <linux/kfifo.h>, <linux/rbtree.h>

Computing Environments - Virtualization

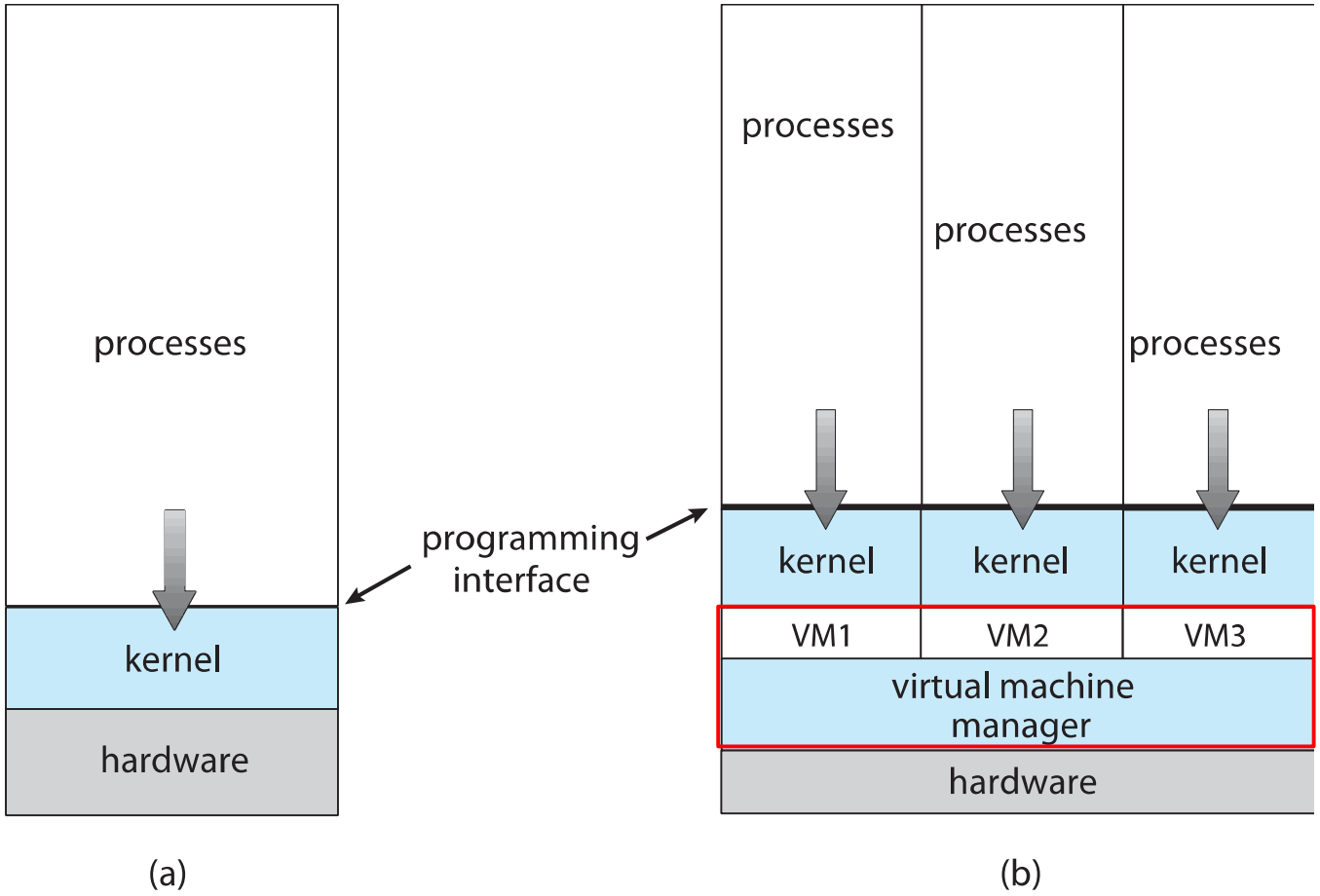
- **Virtualization** is a technology that allows operating systems to *run as applications* within other operating systems.
  - At first blush, there seems to be little reason for such functionality...
- Virtualization also includes **emulation**.
  - Emulation is used when the source CPU type is different from the target CPU type.
  - Example: Allow applications compiled for the IBM CPU to run on the Intel CPU.
    - That same concept can be extended to allow an entire operating system written for one platform to run on another.
- Emulation comes at a heavy price, however.
  - Every machine-level instruction that runs natively on the source system must be translated to the equivalent function on the target system, *frequently resulting in several target instructions*.
  - The emulated code can run much slower than the native code.

Computing Environments - Virtualization

- A common example of emulation occurs when a computer language is not compiled to native code but instead is either executed in its *high-level* form or translated to an *intermediate form*.
  - This is known as **interpretation**.
- Interpretation is a form of emulation in that the *high-level language code is translated to native CPU instructions*, emulating a theoretical (virtual) machine on which that language could run natively.
  - We can run **Java** programs on “Java virtual machines”.
- With **virtualization**, an operating system that is natively compiled for a particular CPU architecture runs within another operating system also native to that CPU.
  - *A virtual machine manager VMM allows the user to install multiple operating systems for exploration or to run applications written for operating systems other than the native host.*



Computing Environments - Virtualization



The VMM runs the guest operating systems, manages their resource use, and protects each guest from the others

Computing Environments – Real-Time Embedded Systems

- **Embedded computers** are the most prevalent form of computers in existence.
- They tend to have very specific tasks. The operating systems provide limited features → they have little or no user interface
- Embedded systems almost always run **real-time operating systems**.
- A **real-time system** is used when rigid time requirements have been placed on the operation of a processor or the flow of data.
  - *Medical imaging systems, industrial control systems, and certain display systems are real time systems.*
- A real-time system has well-defined, **fixed time constraints**. Processing must be done within the defined constraints, or the system will fail.
  - *Contrast this system with a time-sharing system…*

Chapter 2: Outline

- Operating System **Services**
- **User** Operating System **Interface**
- **System Calls**
  - Types of System Calls
- System **Programs**
- …
- Operating System **Structure**

Objectives

- To describe the services an operating system provides to users, processes, and other systems
- To discuss the various ways of structuring an operating system

Operating System Services

- We explore all three aspects of operating systems, showing the viewpoints of users, programmers, and operating system designers.

- Operating systems provide an environment for execution of programs and services to programs and users
- 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)**, **Graphical User Interface (GUI)**
  - **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. The operating system must provide a means to do I/O.

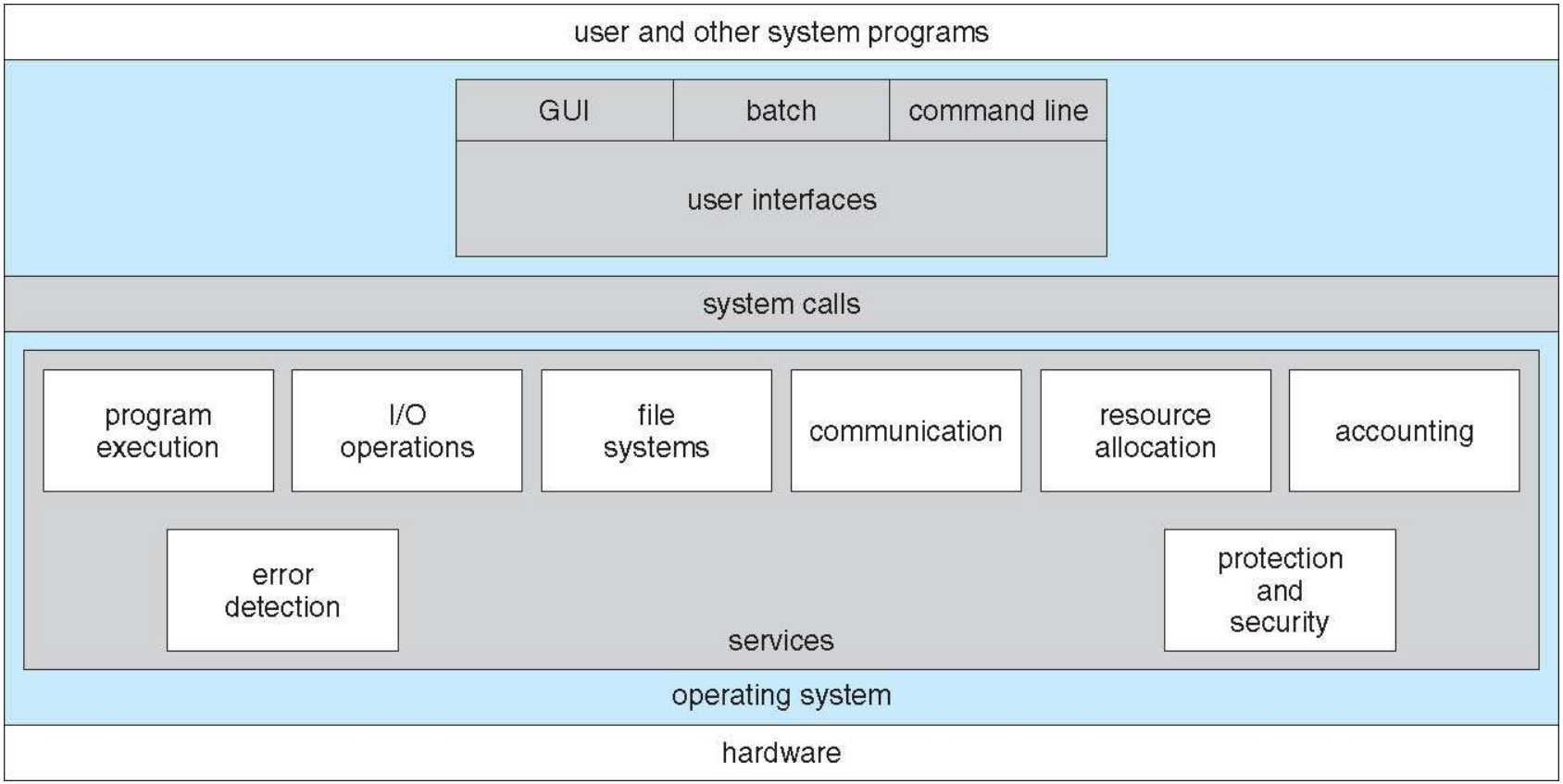
Operating System Services (*Cont.*)

- One set of operating-system services provides *functions that are helpful to the user* (Cont.):
  - **File-system manipulation** - The file system is of particular interest. 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
    - **2 ways** for communications: *via shared memory* or through *message passing* (via a common mailbox)
  - **Error detection** – OS needs to be constantly *aware of possible errors*
    - *Errors may occur in the CPU and memory hardware (such as a memory error or a power failure), in I/O devices (such as a parity error on disk, a connection failure on a network, etc), and in the user program (such as an arithmetic overflow, or an attempt to access an illegal memory location...)*
    - For each type of error, OS should take the appropriate action to ensure correct and consistent computing

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 - CPU cycles, main memory, file storage, I/O devices.
  - **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 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

A View of Operating System Services



User Operating System Interface - CLI

- There are several ways for users to interface with the operating system. Here, we discuss two fundamental approaches.
  - One provides a **command-line interface** (CLI) or command *interpreter*.
  - The other allows users to interface with the operating system via a *graphical user interface*
- On systems with multiple command interpreters to choose from, the interpreters are known as **shells**

- *The main function of the command interpreter is to get and execute the next user-specified command.*
- These commands can be implemented in **two general ways**.
  - In one approach, the *command interpreter itself contains the code* to execute the command.
  - An alternative approach implements *most commands through system programs* In this case, the command interpreter merely uses the command to identify a file to be loaded into memory and executed.

User Operating System Interface - GUI

- User-friendly **desktop** 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**)

- *Many systems now include both* **CLI** *and* **GUI** *interfaces*
  - Microsoft Windows is GUI with CLI "command" shell
  - Apple Mac OS X is "Aqua" GUI interface with UNIX kernel underneath and shells available
  - Unix and Linux have CLI with optional GUI interfaces (CDE, KDE, GNOME)

User Operating System Interface

Choice of Interface

- The choice of whether to use a command-line or GUI interface is mostly one of personal preference.
- System administrators who manage computers and power users who have *deep knowledge of a system* frequently use the command-line interface.
- On some systems, *only a subset of system functions is available via the GUI*, leaving the less common tasks to those who are command-line knowledgeable.
- Further, command-line interfaces usually *make repetitive tasks easier*, in part because they have their own programmability (shell scripts).

System Calls

- *System calls provide a programming interface to the services* provided by the OS
  - Generally available as **routines** written in C and C++, although certain low-level tasks may have to be written using **assembly**
- Even simple programs may make heavy use of the operating system.
  - Frequently, systems execute *thousands of system calls per second*. Most programmers never see this level of detail.
- Let's first use **an example** to illustrate how system calls are used: writing a simple program to read data from one file and copy them to another file...
  - Ask user for filenames
  - Open the files (check for error conditions)
  - Read from file and write to file (check for error conditions)
  - Close files

Example of System Calls

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

Example System Call Sequence

Acquire input file name
  Write prompt to screen
  Accept input
Acquire output file name
  Write prompt to screen
  Accept input
Open the input file
  if file doesn't exist, abort
Create output file
  if file exists, abort
Loop
  Read from input file
  Write to output file
Until read fails
Close output file
Write completion message to screen
Terminate normally

System Calls

- Application developers design programs according to an application programming interface (API).
  - The API specifies a set of *functions that are available* to an application programmer
  - The functions that make up an API typically **invoke the actual system calls** on behalf of the application programmer

- Types of System Calls can be grouped roughly into six major categories:
  - process control
  - file manipulation
  - device manipulation
  - information maintenance
  - communications
  - protection

System Calls

- 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)
  - Java API for the Java virtual machine (JVM)

Note that the system-call names used throughout this text are **generic**

- A programmer accesses an API via a library of code provided by the operating system.
  - *In the case of UNIX and Linux for programs written in the C language, the library is called **libc**.*
- **Why** would an application programmer prefer programming according to an API rather than invoking actual system calls?
- There are several reasons for doing so...
  - One benefit concerns program portability.
  - Actual system calls can often be more detailed and difficult to work with

Example of Standard API

EXAMPLE OF STANDARD API

As an example of a standard API, consider the `read()` function that is available in UNIX and Linux systems. The API for this function is obtained from the `man` page by invoking the command

```
man read
```

on the command line. A description of this API appears below:

```
#include <unistd.h>

ssize_t    read(int fd, void *buf, size_t count)
```

return value — function name — parameters

A program that uses the `read()` function must include the `unistd.h` header file, as this file defines the `ssize_t` and `size_t` data types (among other things). The parameters passed to `read()` are as follows:

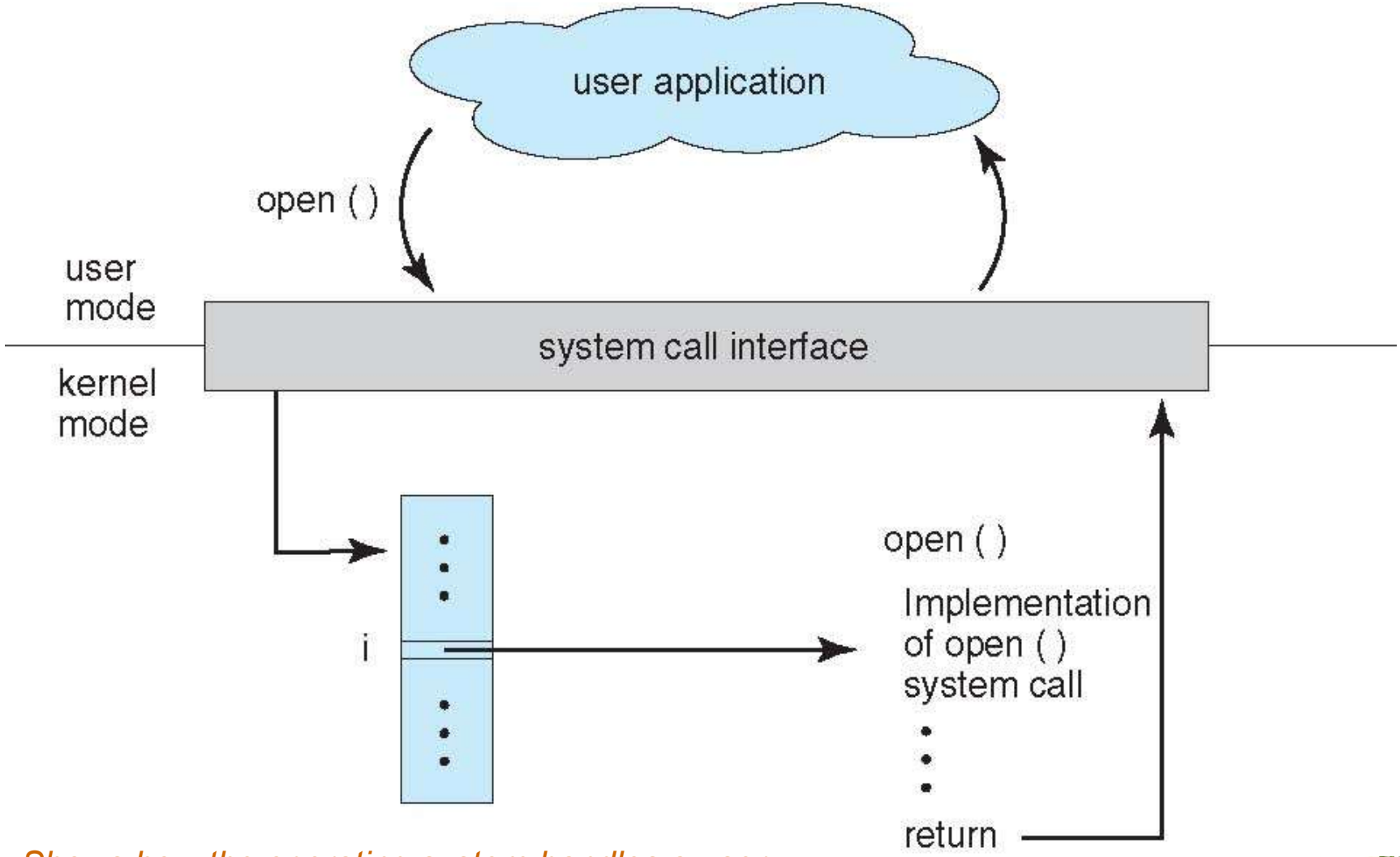
- `int fd`—the file descriptor to be read
- `void *buf`—a buffer where the data will be read into
- `size_t count`—the maximum number of bytes to be read into the buffer

On a successful read, the number of bytes read is returned. A return value of 0 indicates end of file. If an error occurs, `read()` returns −1.

System Call Implementation

- The caller need know nothing about how the system call is implemented or what it does during execution.
  - Thus, most of the *details of the operating-system interface are hidden from the programmer* **by the API** and are managed by the run-time support library
- For most programming languages, the run-time support system (*a set of functions built into libraries included with a compiler*) provides a **system call interface** that is the link to available system calls
- Typically, *a number is associated with each system call*
  - **System-call interface** maintains *a table indexed according to these numbers*
- The system call interface *invokes the intended system call in OS kernel* and returns status of the system call and any return values

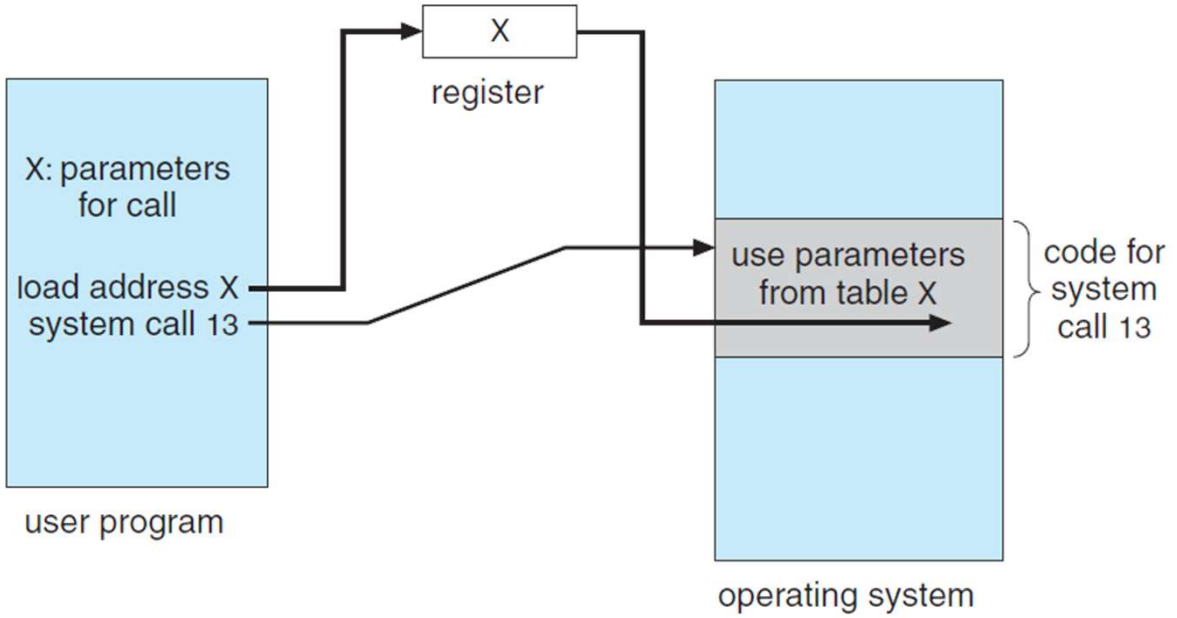
API – System Call – OS Relationship



Shows how the operating system handles a user application invoking the open() system call.

System Call Implementation

- System calls occur in different ways, depending on the computer in use.
  - Often, more information is required than simply the identity of the desired system call.
- **General methods used to pass parameters to the operating system**.
  - The simplest approach is to pass the parameters in registers. In some cases, there may be more parameters than registers.
  - So, parameters are then stored in a block, or table, in memory, and the address of the block is passed as a parameter in a register



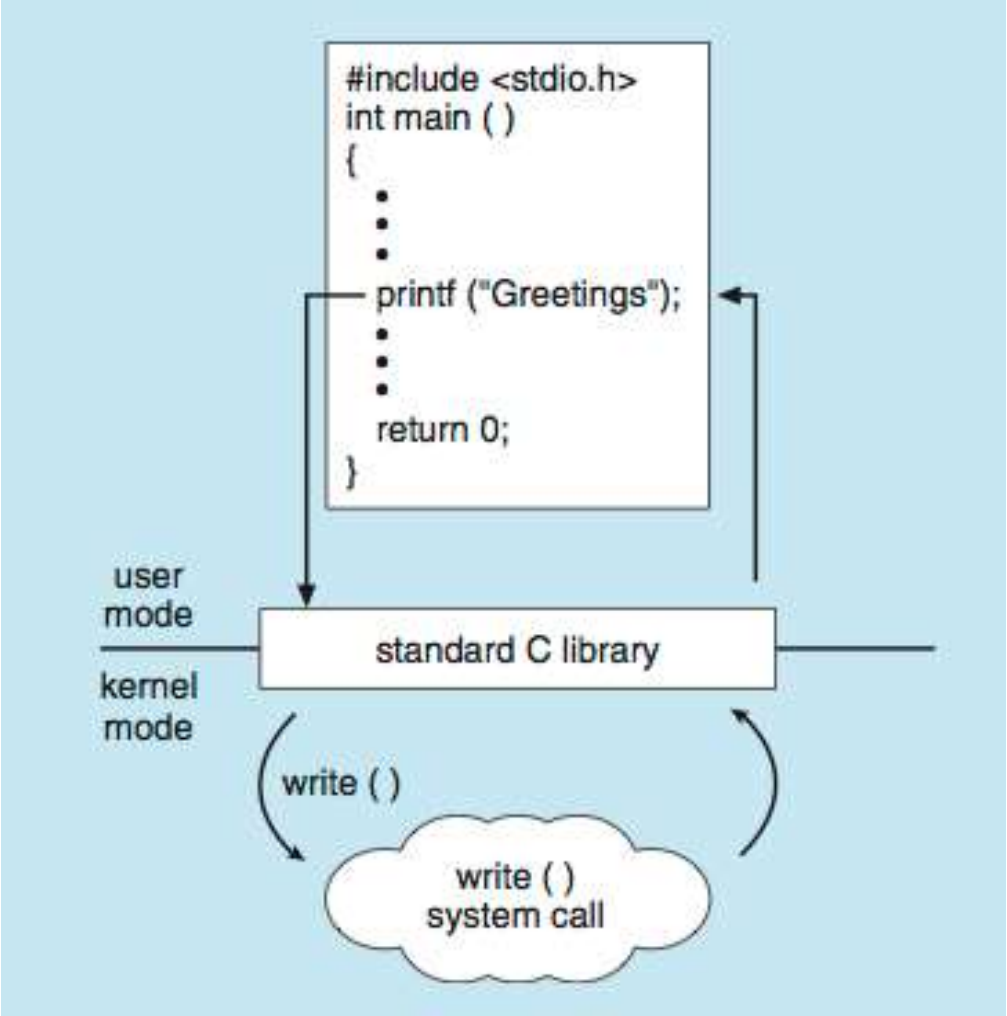
Types of System Calls

We briefly discuss the **types of system calls** that may be provided by an operating system

- **Process control**
  - A running program may need to *halt its execution* either normally (end()) or abnormally (abort())
  - A process executing one program may want to load() and execute() another program.
  - Determine and reset the attributes of a process, including priority, its maximum allowable execution time, etc.
  - Terminate a job or process that we created
  - Wait for processes to finish their execution; wait for a certain amount of time or a specific event to occur
  - Allow a process to **lock** shared data. Then, no other process can access the data until the lock is released. (acquire_lock(), release_lock())
  - Allocate and free memory

Standard C Library Example

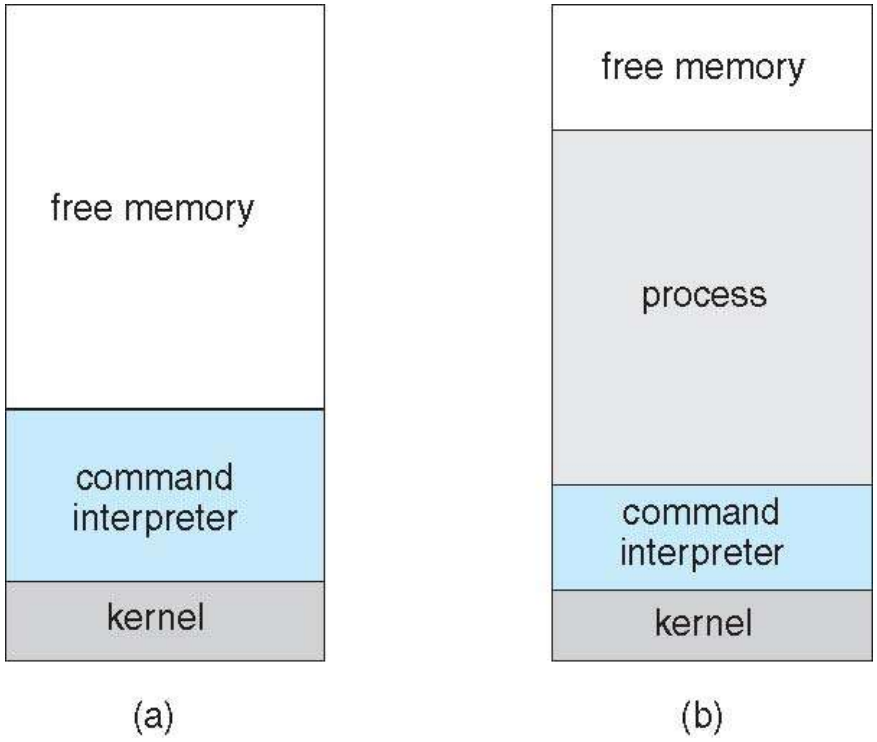
- Let's assume a C program invokes the printf() statement.
  - The C library intercepts this call and invokes the necessary system call



Example: MS-DOS

- We next use two examples—one involving a **single-tasking** system and the other a **multitasking** system—to clarify some concepts.

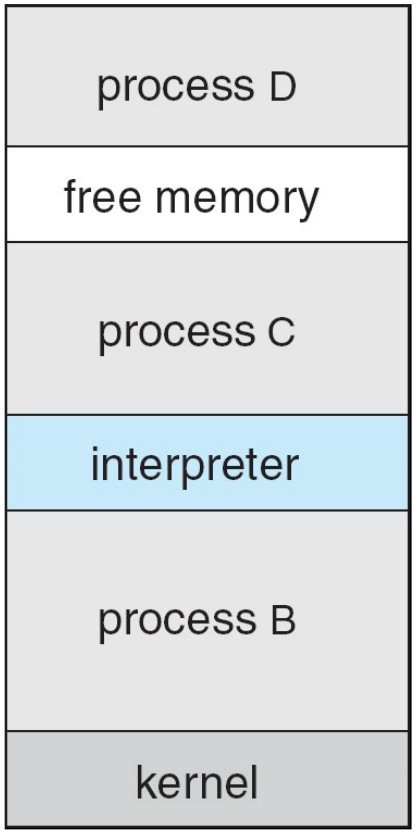
- **Single-tasking**
- Shell invoked when system booted
- Simple method to run program
  - No process created
- Single memory space
- Loads program into memory, overwriting most of itself except the kernel
- Program exit -> shell reloaded



At system startup running a program

Example: FreeBSD

- Unix variant
- **Multitasking**
- User login -> invoke user's choice of shell
- The command interpreter may continue running while another program is executed
- Shell executes **fork**() system call to create a new process
  - Executes **exec**() to load program into the process
  - Shell waits for process to terminate ***or*** continues with user commands (*runs the process in the background*)
- When the process is done, it executes an **exit**() system call to terminate



FreeBSD running multiple programs

Other Types of System Calls

- **File management**
  - create file, delete file
  - open, close file
  - read, write, reposition
  - get and set file attributes
- **Device management**
  - request device, release device
  - read, write, reposition
  - get device attributes, set device attributes
  - logically attach or detach devices

Other Types of System Calls (*Cont.*)

- **Information maintenance**
  - return information about the system
  - Info for debugging
    - get time or date, set time or date
    - get system data, set system data
    - get and set process, file, or device attributes
- **Communication**
  - create, delete communication connection
  - send, receive messages if **message passing model** to **host name** or **process name**
  - **Shared-memory model** create and gain access to memory regions
    - *Message passing is useful for exchanging smaller amounts of data*
  - attach and detach remote devices

Other Types of System Calls (*Cont.*)

- **Communication (Cont.)**
  - Note that normally, the operating system tries to prevent one process from accessing another process's memory.
    - Shared memory requires that two or more processes agree to remove this restriction.
    - They can then exchange information by reading and writing data in the shared areas.
    - The form of the data is not under the operating system's control.
    - The processes are also responsible for ensuring that they are not writing to the same location simultaneously.
- **Protection**
  - Control access to resources
  - Get and set permissions
  - Allow and deny user access

Examples of Windows and Unix System Calls

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

Fin de la Clase

Gracias por
su asistencia y atención

¿ Preguntas ?