OS - 2005

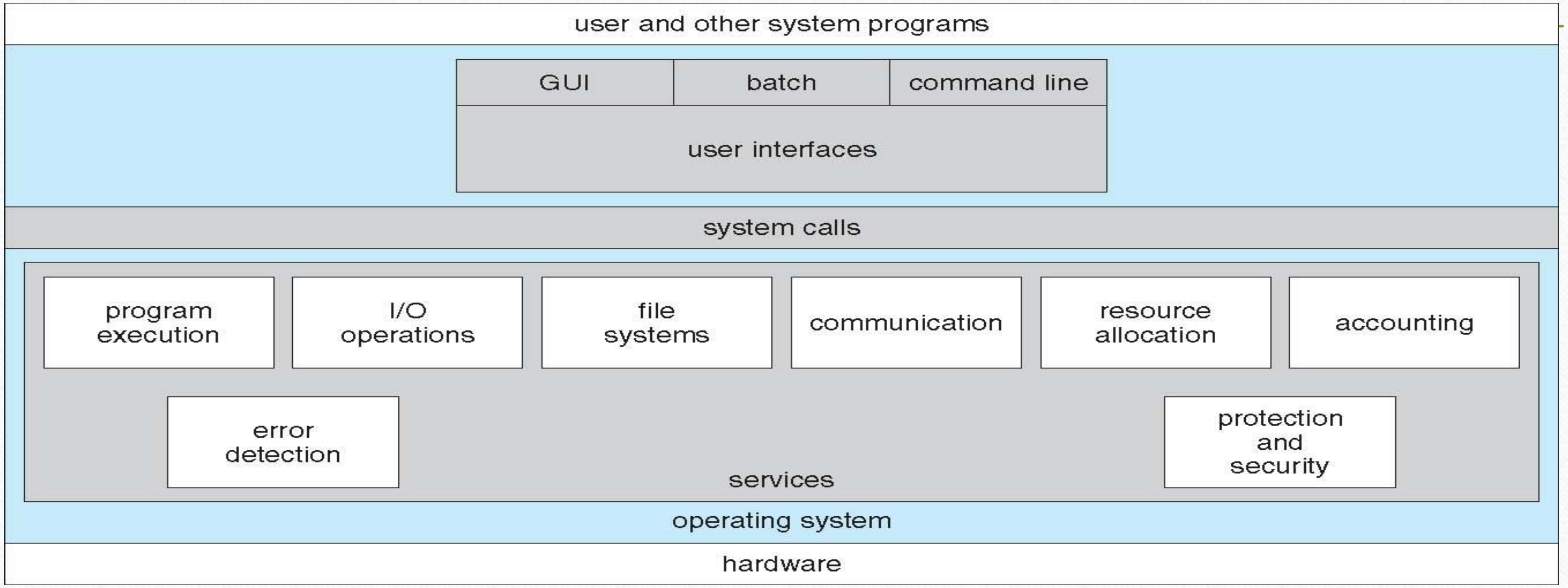
System Calls – System/Application Call Interface

Protection User/Kernel modes - Interrupts

Introduction

- Operating System Services
- User Operating System Interface
- System Calls
- Types of System Calls
- System Programs
- Operating System Design and Implementation
- Interrupts

A View of Operating System Services



Source: “Operating System Concepts” by Avi Silberschatz and Peter Galvin (9th Edition)

User Operating System Interface - CLI

- Command Line Interface (CLI) or **command interpreter** allows direct command entry
    - Sometimes implemented in kernel, sometimes by systems program
    - Sometimes multiple flavors implemented – **shells**
    - Primarily fetches a command from user and executes it
        - Sometimes commands built-in, sometimes just names of programs
            - If the latter, adding new features doesn't require shell modification

User Operating System Interface - GUI

- User-friendly **desktop** metaphor interface
  - Usually mouse, keyboard, and monitor
  - **Icons** represent files, programs, actions, etc
  - Various mouse buttons over objects in the interface cause various actions (provide information, options, execute function, open directory (known as a **folder**)
  - Invented at Xerox PARC
- Many systems now include both CLI and GUI interfaces
  - Microsoft Windows is GUI with CLI "command" shell
  - Apple Mac OS X as "Aqua" GUI interface with UNIX kernel underneath and shells available
  - Solaris is CLI with optional GUI interfaces (Java Desktop, KDE)

Protection users/kernel mode

- From a programmer's point of view, the system is the CPU used to execute instructions plus the memory used to hold instructions and data.
- This simplistic view might have been true when one person owned a microcomputer which ran one program at a time, but this is no longer the case.
- If every program had unfettered access to the CPU, main memory and the peripheral devices, all concepts of separation of programs and the data in memory, on disk etc. would not exist.

Protection users/kernel mode

- A program could look at all memory locations, including that of other programs, as well as read all the data on all of the attached disks, and read all the data being sent across the network.
- To prevent this, we need the CPU to have at least two privilege levels.

Kernel mode (privileged mode)

User mode (non- privileged mode)

Kernel Mode

- Kernel mode, also referred to as system mode.
- Mainly for Restriction/ Protection from unauthorized user application.
- When the CPU is in kernel mode, it is assumed to be executing trusted software, and thus it can execute any instructions and reference any memory addresses (i.e., locations in memory).
- All other programs(user applications) are considered untrusted software.
- Thus, all user mode software must request use of the kernel by means of a system call in order to perform privileged instructions, such as process creation or input/output operations

User Mode

- User mode is the normal mode of operating for programs.
- They don't interact directly with the kernel, instead, they just give instructions on what needs to be done, and the kernel takes care of the rest.
- Code running in user mode must delegate to system APIs to access hardware or memory.
- Due to the protection afforded by this sort of isolation, crashes in user mode are always recoverable.

User Mode

- In user mode, some parts of RAM cannot be addressed, some instructions can’t be executed, and I/O ports can’t be accessed.

- When a user-mode process wants to use a service that is provided by the kernel, it must switch temporarily into kernel mode.
- The standard procedure to switch from user mode to kernel mode is to call software interrupt.

Protection user/kernel mode

Thus,

- In **kernel mode**, the CPU has instructions to manage memory and how it can be accessed, plus the ability to access peripheral devices like disks and network cards.
- In **user mode**, access to memory is limited to only some memory locations, and access to peripheral devices is denied.

Now, all programs will be run in user mode, and this prevents them from accessing the data in other programs, as well as preventing the disk etc.

Switching from User to Kernel Mode

- The only way an user space application can explicitly initiate a switch to kernel mode during normal operation is by making an system.
- Whenever a user application calls these system call APIs with appropriate parameters, a **software interrupt/exception(SWI)** is triggered.
- As a result of this SWI, the control of the code execution jumps from the user application to a predefined location in the Interrupt Vector Table [IVT] provided by the OS.

Switching from User to Kernel Mode

- This IVT contains an address for the SWI exception handler routine, which performs all the necessary steps required to switch the user application to kernel mode and start executing kernel instructions on behalf of user process.
- An "**interrupt vector table**" (IVT) is a data structure that associates a list of **interrupt** handlers with a list of **interrupt** requests in a **table** of **interrupt vectors**. Each entry of the **interrupt vector table**, called an **interrupt vector**, is the address of an **interrupt** handler.

Interrupts

- An interrupt is a signal to the operating system that an event has occurred, and it results in changes in the sequence of instructions that are executed by the CPU.
- They are the part of the operating system and run in kernel mode.
- A hardware interrupt, the signal originates from a hardware device such as a keyboard, mouse or system clock.
- A software interrupt is an interrupt that originates in software, usually by a program in user mode.

System Calls

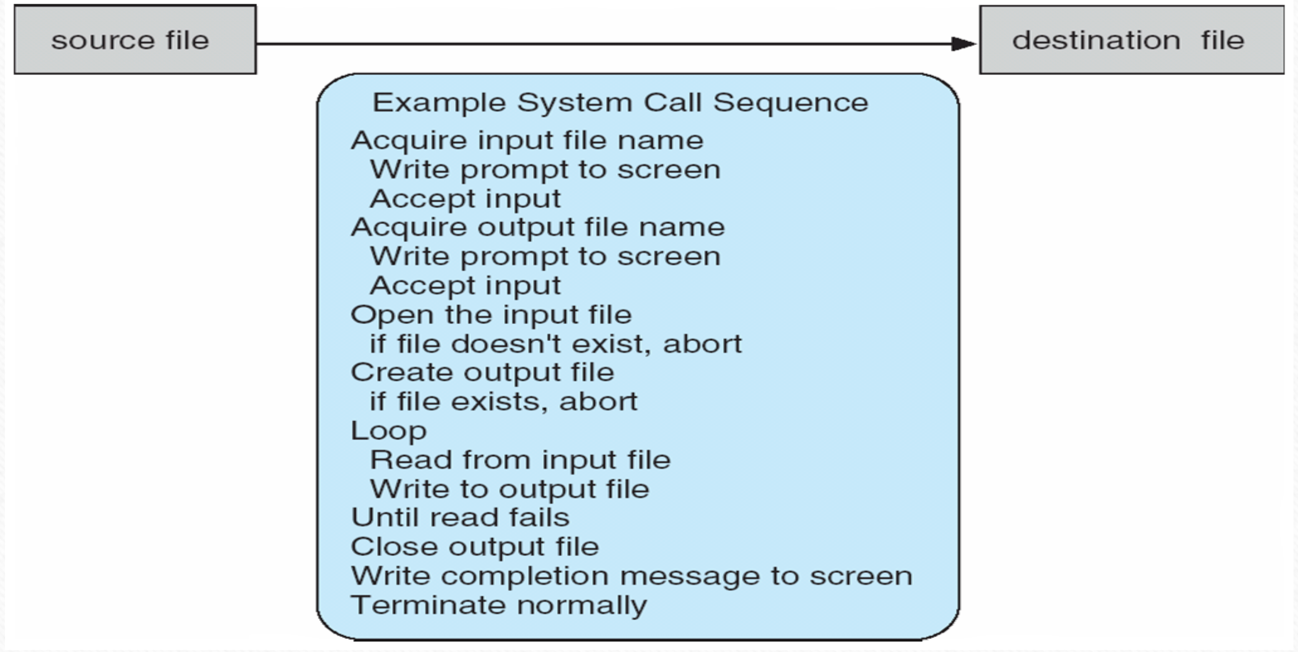
- Programming interface to the services provided by the OS
- Typically written in a high-level language (C or C++)
- Mostly accessed by programs via a high-level **Application Program Interface (API)** rather than direct system call use.
- Three most common APIs are Win32 API for Windows, POSIX API for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

Why use APIs rather than system calls?

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

Example of System Calls

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



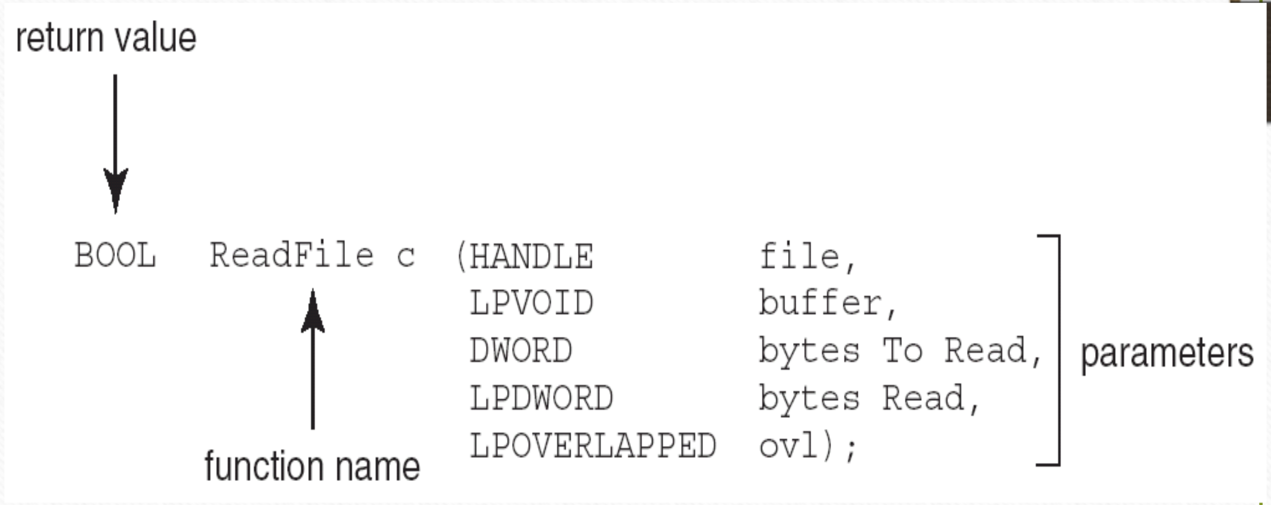
Example System Call Sequence

```
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
```

Source: "Operating System Concepts: 6th Edition" by Silberschatz Galvin Gagne

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

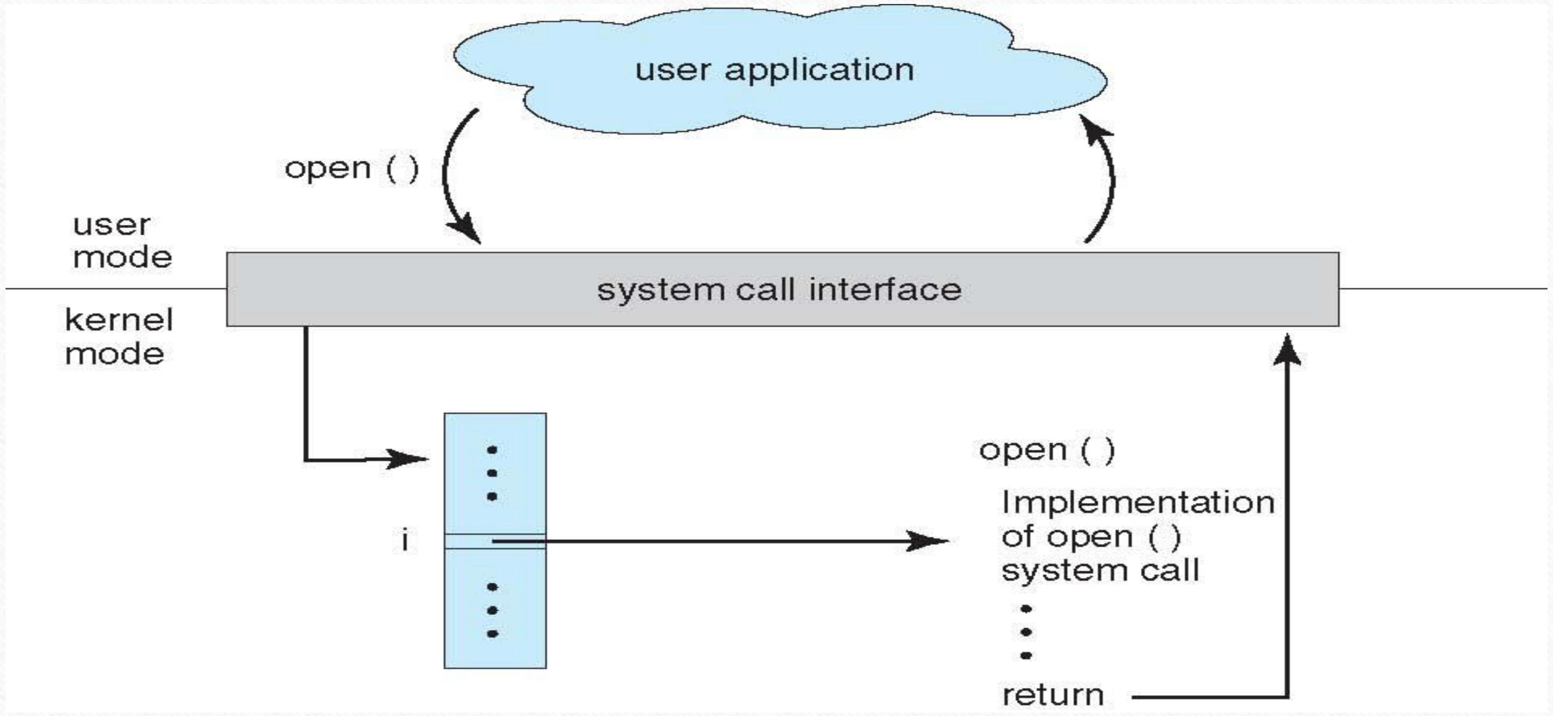


```
BOOL   ReadFile c  (HANDLE        file,
                    LPVOID        buffer,
                    DWORD         bytes To Read,
                    LPDWORD       bytes Read,
                    LPOVERLAPPED  ovl);
```

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)

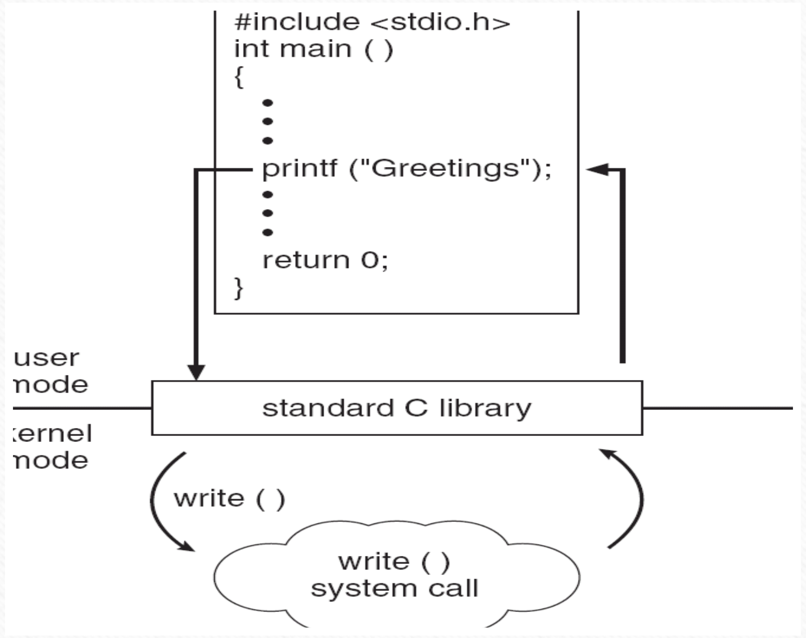
API – System Call – OS Relationship



Souce: “Operating System Concepts: 7th Edition” by Silberschatz Galvin Gagne

Standard C Library Example

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

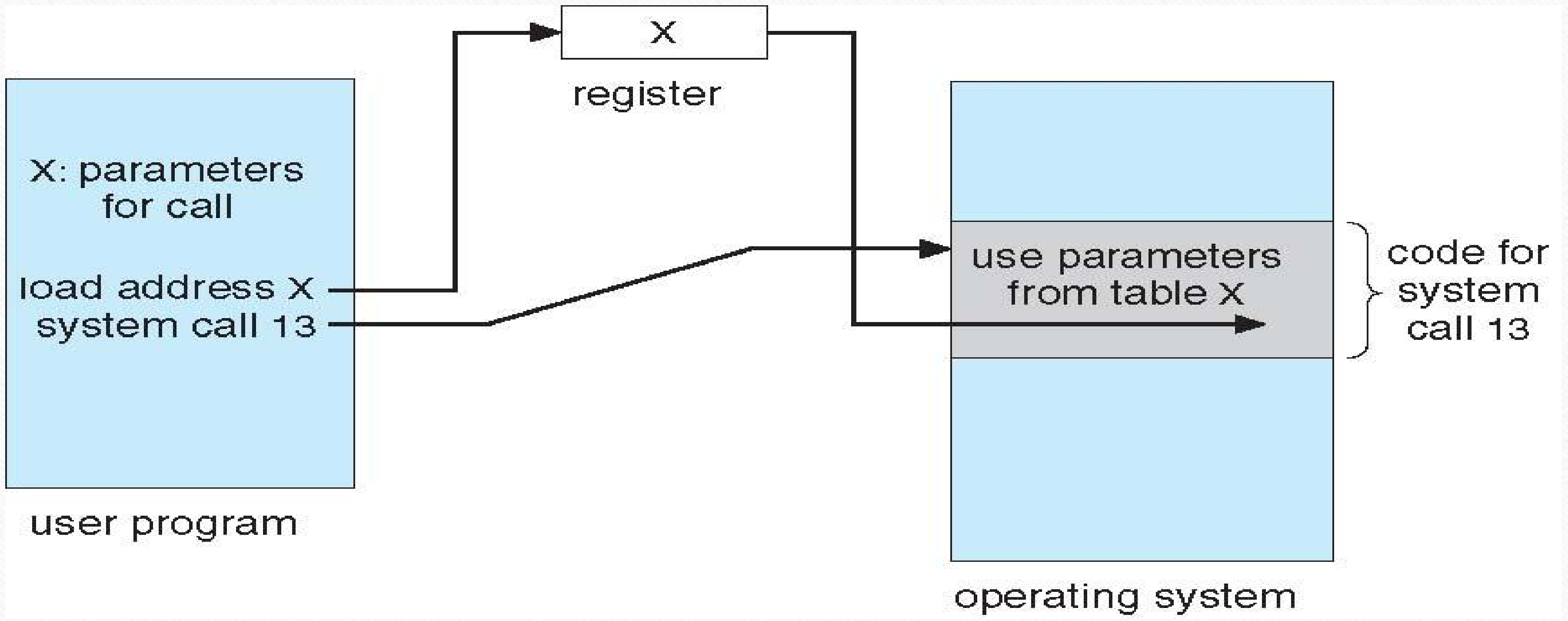


```
#include <stdio.h>
int main ( )
{
   •
   •
   •
   printf ("Greetings");
   •
   •
   •
   return 0;
}
```

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



Souce: “Operating System Concepts: 7th Edition” by Silberschatz Galvin Gagne

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

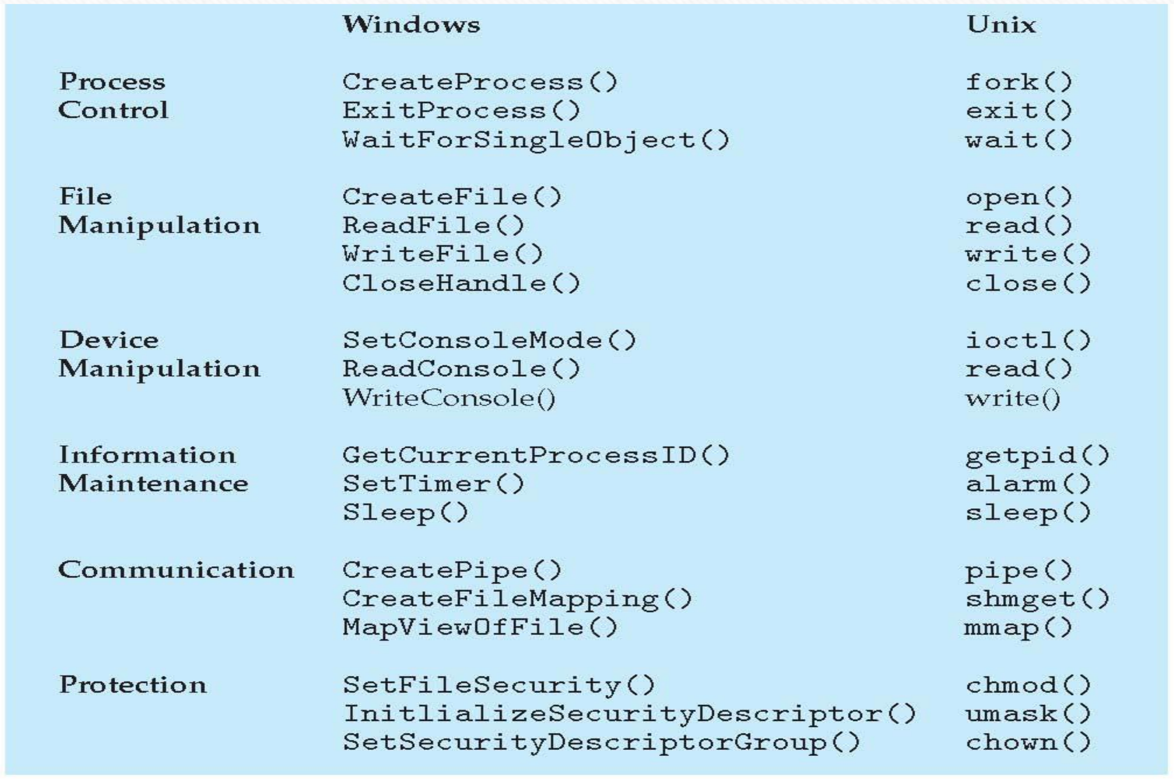
Types of System Calls

- **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

Types of System Calls (Cont.)

- **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

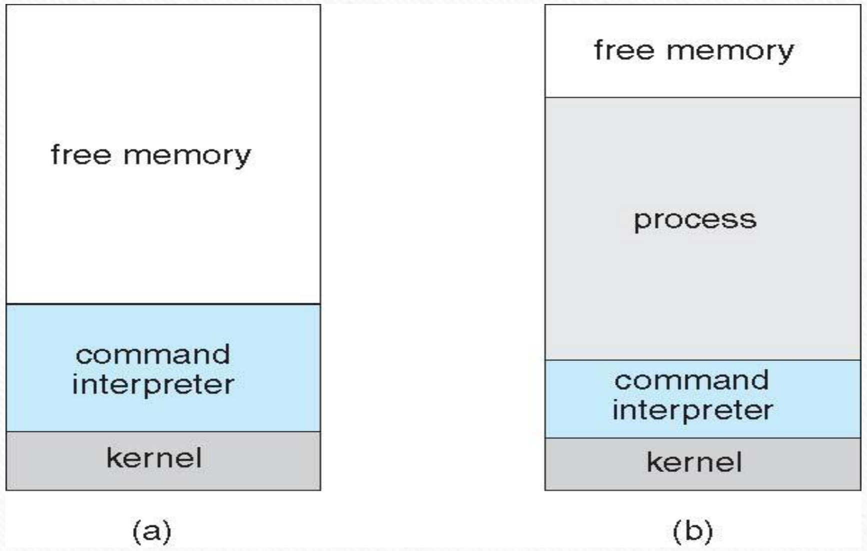


	Windows	Unix
Process Control	CreateProcess()	fork()
	ExitProcess()	exit()
	WaitForSingleObject()	wait()
File Manipulation	CreateFile()	open()
	ReadFile()	read()
	WriteFile()	write()
	CloseHandle()	close()
Device Manipulation	SetConsoleMode()	ioctl()
	ReadConsole()	read()
	WriteConsole()	write()
Information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

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



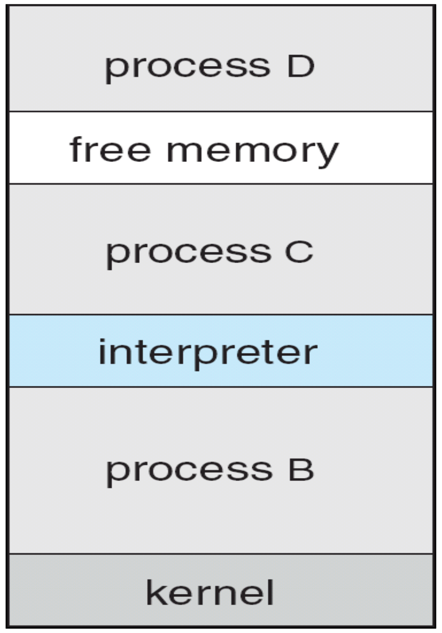
(a) At system startup (b) running a program

Souce: “Operating System Concepts: 7th Edition” by Silberschatz Galvin Gagne

Example: FreeBSD

- Unix variant
- Multitasking
- User login -> invoke user's choice of shell
- Shell executes fork() system call to create process
  - Executes exec() to load program into process
  - Shell waits for process to terminate or continues with user commands
- Process exits with code of 0 – no error or > 0 – error code

FreeBSD Running Multiple Programs



System Programs

- System programs provide a convenient environment for program development and execution. The can be divided into:
  - File manipulation
  - Status information
  - File modification
  - Programming language support
  - Program loading and execution
  - Communications
  - Application programs
- Most users' view of the operation system is defined by system programs, not the actual system calls

System Programs

- Provide a convenient environment for program development and execution
  - Some of them are simply user interfaces to system calls; others are considerably more complex

- **File management** - Create, delete, copy, rename, print, dump, list, and generally manipulate files and directories
- **Status information**
  - Some ask the system for info - date, time, amount of available memory, disk space, number of users
  - Others provide detailed performance, logging, and debugging information
  - Typically, these programs format and print the output to the terminal or other output devices
  - Some systems implement a registry - used to store and retrieve configuration information

System Programs (Cont.)

- **File modification**
  - Text editors to create and modify files
  - Special commands to search contents of files or perform transformations of the text
- **Programming-language support** - Compilers, assemblers, debuggers and interpreters sometimes provided
- **Program loading and execution**- Absolute loaders, relocatable loaders, linkage editors, and overlay-loaders, debugging systems for higher-level and machine language
- **Communications** - Provide the mechanism for creating virtual connections among processes, users, and computer systems
  - Allow users to send messages to one another's screens, browse web pages, send electronic-mail messages, log in remotely, transfer files from one machine to another

Operating System Design and Implementation

- Design and Implementation of OS not “solvable”, but some approaches have proven successful
- Internal structure of different Operating Systems can vary widely
- Start by defining goals and specifications
- Affected by choice of hardware, type of system
- ***User* goals** and ***System* goals**
    - User goals – operating system should be convenient to use, easy to learn, reliable, safe, and fast
    - System goals – operating system should be easy to design, implement, and maintain, as well as flexible, reliable, error-free, and efficient

Operating System Design and Implementation (Cont.)

- Important principle to separate

  Policy: What will be done?
  Mechanism: How to do it?

- Mechanisms determine how to do something, policies decide what will be done
  - The separation of policy from mechanism is a very important principle, it allows maximum flexibility if policy decisions are to be changed later

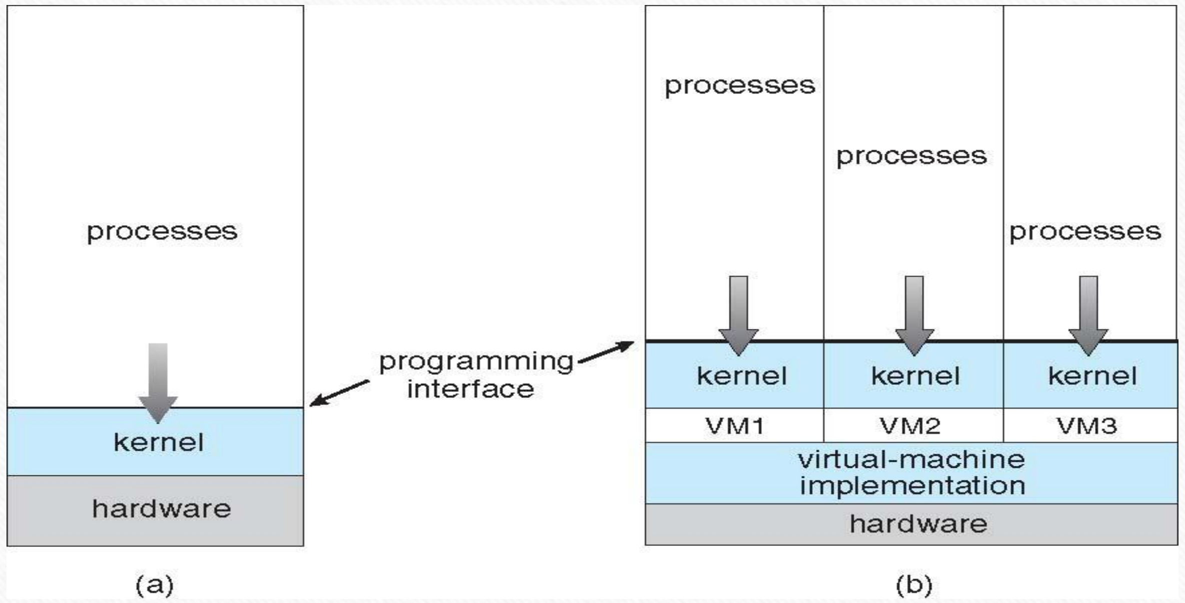
Virtual Machines

- A **virtual machine** takes the layered approach to its logical conclusion. It treats hardware and the operating system kernel as though they were all hardware.
- A virtual machine provides an interface *identical* to the underlying bare hardware.
- The operating system **host** creates the illusion that a process has its own processor and (virtual memory).
- Each **guest** provided with a (virtual) copy of underlying computer.

Virtual Machines History and Benefits

- First appeared commercially in IBM mainframes in 1972
- Fundamentally, multiple execution environments (different operating systems) can share the same hardware
- Protect from each other
- Some sharing of file can be permitted, controlled
- Commutate with each other, other physical systems via networking
- Useful for development, testing
- **Consolidation** of many low-resource use systems onto fewer busier systems
- "Open Virtual Machine Format", standard format of virtual machines, allows a VM to run within many different virtual machine (host) platforms

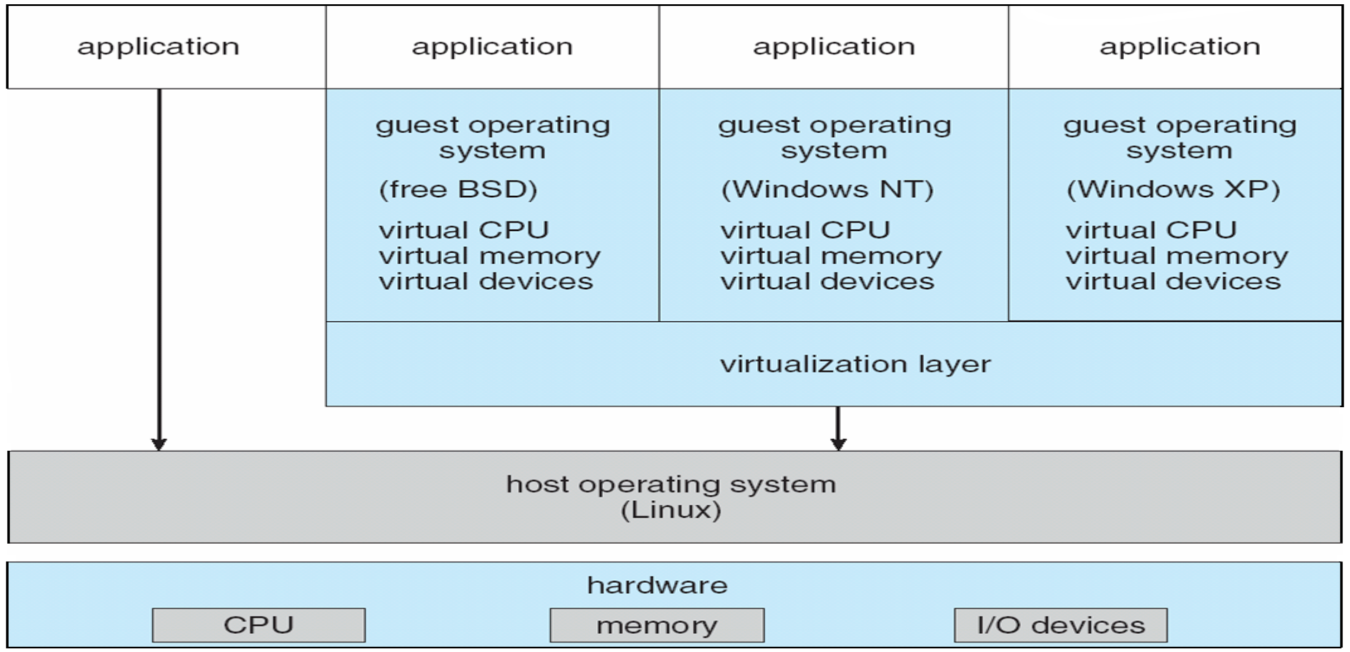
Virtual Machines (Cont.)



(a) Non Virtual machine (b) virtual machine

Souce: “Operating System Concepts: 7th Edition” by Silberschatz Galvin Gagne

VMware Architecture



Souce: “Operating System Concepts: 7th Edition” by Silberschatz Galvin Gagne

Summary - System Call

System Calls

- System calls provide the interface between a running program and the OS Think of it as a set of functions available to the program to call (but somewhat different from normal functions, we will see why)
- Generally available as assembly-language instructions.
- Most common languages (e.g., C, C++) have APIs that call system calls underneath

- Passing parameters to system calls Pass parameters in *registers*
- Store the parameters in a table in memory, and the table address is passed as a parameter in a register
- *Push* (store) the parameters onto the *stack* by the program, and *pop* off the stack by operating system

Summary

- Operating systems are designed to run on any of a class of machines; the system must be configured for each specific computer site
- SYSGEN program obtains information concerning the specific configuration of the hardware system
- *Booting* – starting a computer by loading the kernel
- *Bootstrap program* – code stored in ROM that is able to locate the kernel, load it into memory, and start its execution

References

1. N. Markovic, D. Nemirovsky, O. Unsal, M. Valero and A. Cristal, "Kernel-to-User-Mode Transition-Aware Hardware Scheduling," in IEEE Micro, vol. 35, no. 4, pp. 37-47, July-Aug. 2015, doi: 10.1109/MM.2015.80.
2. Operating System Concepts, 7th Edition" by Silberschatz Galvin Gagne
3. System Calls Analysis, F. Tchakounté, P. Dayang, *International Journal of Science and Technology Volume 2 No. 9, September, 2013*
4. Vaggelis Atlidakis, Jeremy Andrus, Roxana Geambasu, Dimitris Mitropoulos, and Jason Nieh. Posix abstractions in modern operating systems: The old, the new, and the missing. In Proceedings of the Eleventh European Conference on Computer Systems (EuroSys), pages 19:1–19:17.

   ACM, 2016