Module 2: L1

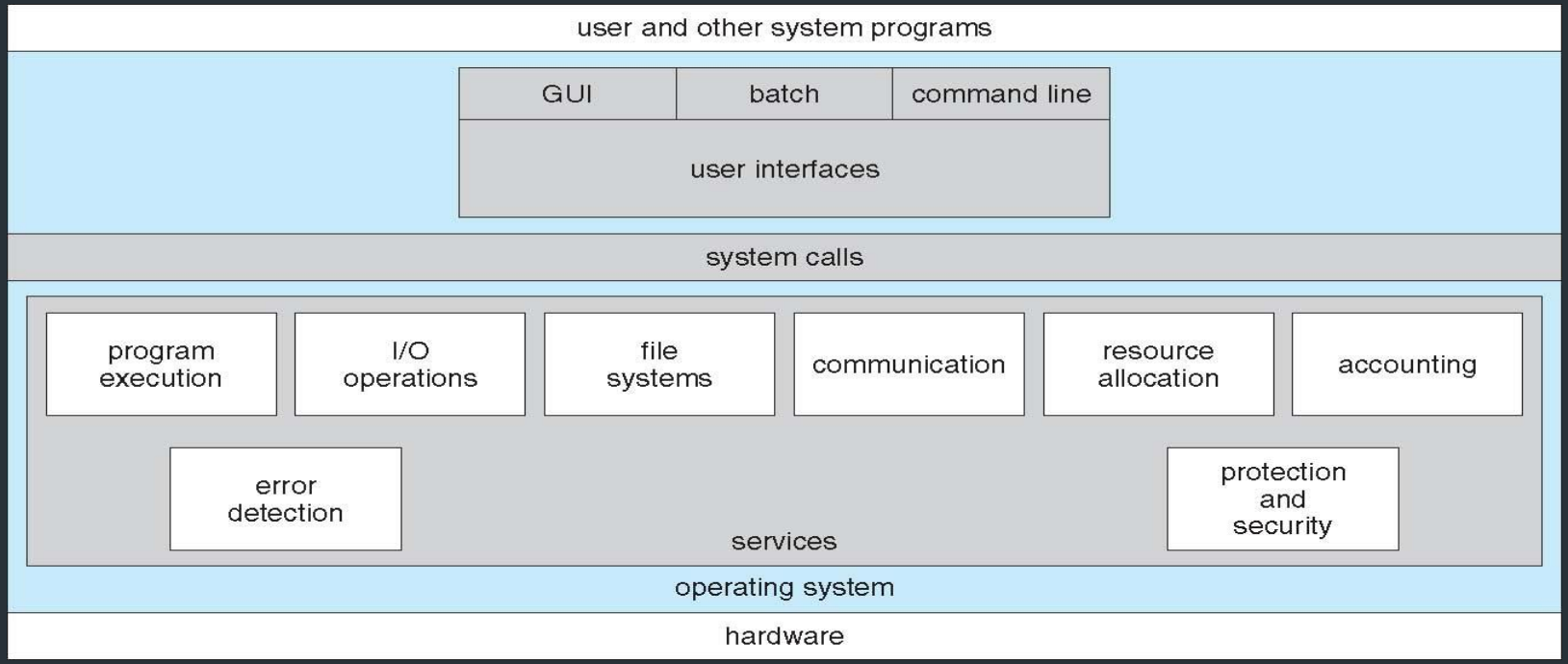
System Calls – System/Application Call Interface Protection User/Kernel modes - Interrupts

Dr. Rishikeshan C A
SCOPE, VIT Chennai

Contents

- 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: 9th Edition" by Silberschatz Galvin Gagne

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)

Operating-System Operations

- **Dual-mode** operation allows OS to protect itself and other system components
  - **User mode** and **kernel mode**
  - **Mode bit** provided by hardware
    - Provides ability to distinguish when system is running user code or kernel code
    - Some instructions designated as **privileged**, only executable in kernel mode
    - System call changes mode to kernel, return from call resets it to user
- Increasingly CPUs support multi-mode operations
  - i.e. **virtual machine manager** (**VMM**) mode for guest **VMs**

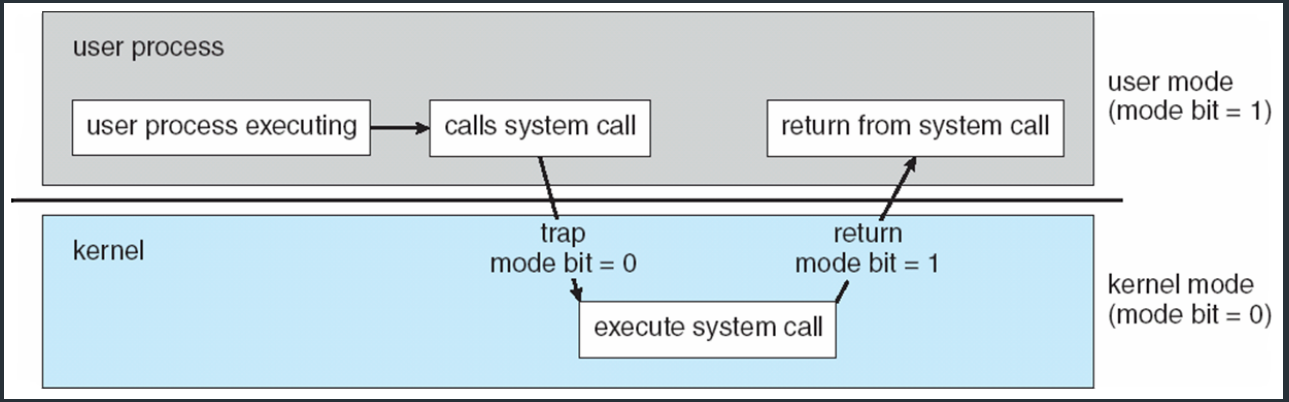
Protection users/kernel mode

- 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)

Transition from User to Kernel Mode

- The transition from user mode to kernel mode occurs when the application requests the help of operating system or an interrupt or a system call occurs. The mode bit is set to 1 in the user mode. It is changed from 1 to 0 when switching from user mode to kernel mode.
- In the fig. below, the user process executes in the user mode until it gets a system call. Then a system trap is generated and the mode bit is set to zero. The system call gets executed in kernel mode. After the execution is completed, again a system trap is generated and the mode bit is set to 1. The system control returns to kernel mode and the process execution continues.



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.
- 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 in 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

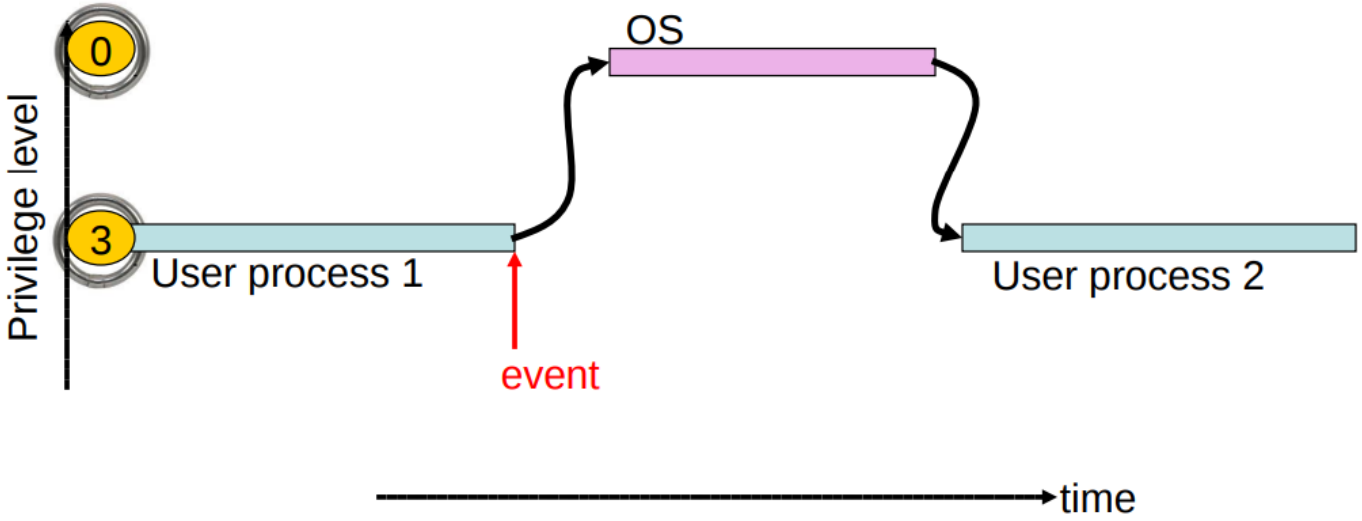
- The only way an user space application can explicitly initiate a switch to kernel mode during normal operation is by making an system.
- Whenever a user application calls these system call APIs with appropriate parameters, a **software interrupt/exception(SWI)** is triggered.
- As a result of this 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.
- 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

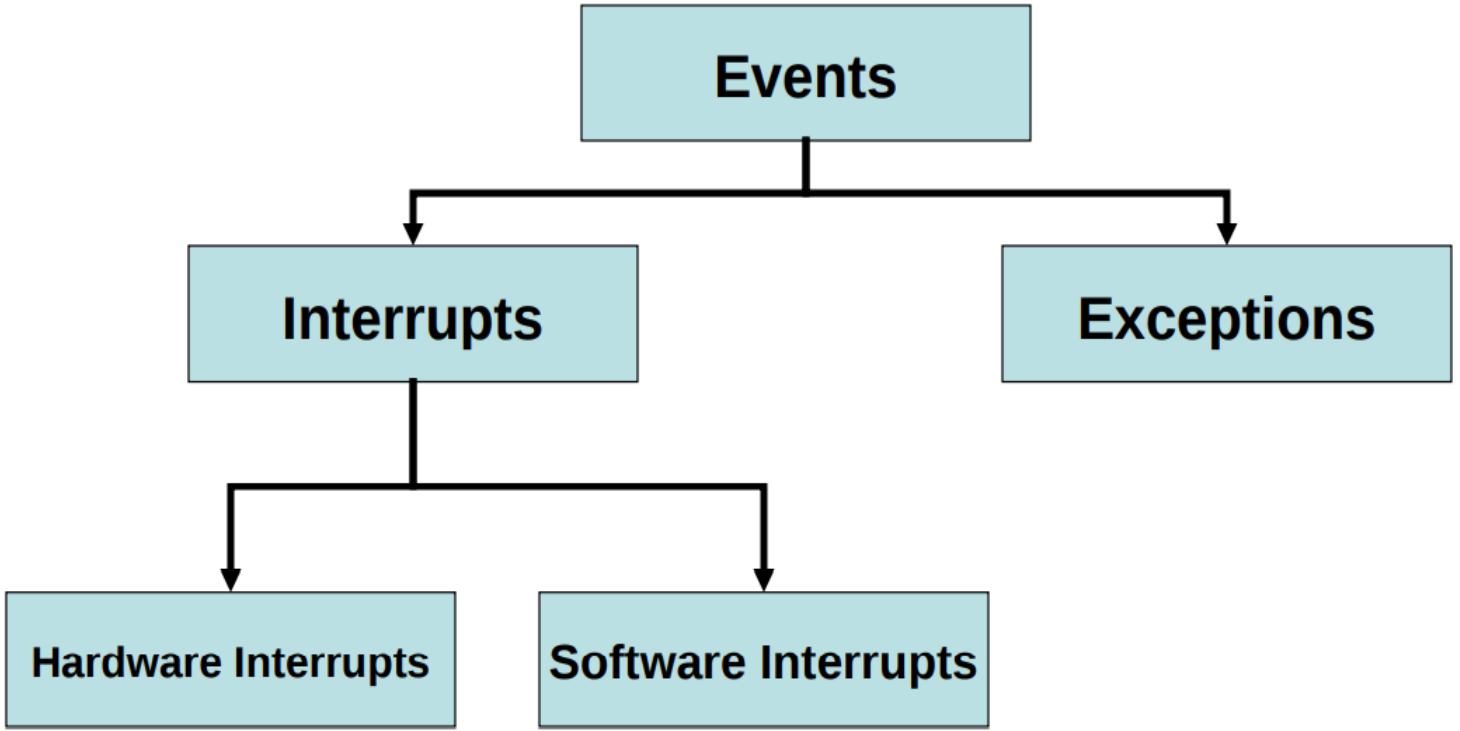
- An interrupt is a signal to the operating system that an event has occurred, and it results in changes in the sequence of instructions that are executed by the CPU.
- They are the part of the operating system and run in kernel mode.
- A hardware interrupt, the signal originates from a hardware device 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.

OS & Events

- OS is event driven
  - i.e. executes only when there is an interrupt, trap, or system call



Event Types


Events
Interrupts
Exceptions
Hardware Interrupts
Software Interrupts

Events

- Interrupts : raised by hardware or programs to get OS attention
  - Types
    - Hardware interrupts : raised by external hardware devices
    - Software Interrupts : raised by user programs
- Exceptions : due to illegal operations

Exception & Interrupt Vectors

Event occured **What to execute next?**

- Each interrupt/exception provided a number
- Number used to index into an Interrupt descriptor table (IDT)
- IDT provides the entry point into a interrupt/exception handler
- 0 to 255 vectors possible
  - 0 to 31 used internally
  - Remaining can be defined by the OS

Vector No.	Mnemonic	Description	Type	Error Code	Source
0	#DE	Divide Error	Fault	No	DIV and IDIV instructions.
1	#DB	RESERVED	Fault/ Trap	No	For Intel use only.
2	—	NMI Interrupt	Interrupt	No	Nonmaskable external interrupt.
3	#BP	Breakpoint	Trap	No	INT 3 instruction.
4	#OF	Overflow	Trap	No	INTO instruction.
5	#BR	BOUND Range Exceeded	Fault	No	BOUND instruction.
6	#UD	Invalid Opcode (Undefined Opcode)	Fault	No	UD2 instruction or reserved opcode.[1]
7	#NM	Device Not Available (No Math Coprocessor)	Fault	No	Floating-point or WAIT/FWAIT instruction.
8	#DF	Double Fault	Abort	Yes (zero)	Any instruction that can generate an exception, an NMI, or an INTR.
9		Coprocessor Segment Overrun (reserved)	Fault	No	Floating-point instruction.[2]
10	#TS	Invalid TSS	Fault	Yes	Task switch or TSS access.
11	#NP	Segment Not Present	Fault	Yes	Loading segment registers or accessing system segments.
12	#SS	Stack-Segment Fault	Fault	Yes	Stack operations and SS register loads.
13	#GP	General Protection	Fault	Yes	Any memory reference and other protection checks.
14	#PF	Page Fault	Fault	Yes	Any memory reference.
15	—	(Intel reserved. Do not use.)		No	
16	#MF	x87 FPU Floating-Point Error (Math Fault)	Fault	No	x87 FPU floating-point or WAIT/FWAIT instruction.
17	#AC	Alignment Check	Fault	Yes (Zero)	Any data reference in memory.[3]
18	#MC	Machine Check	Abort	No	Error codes (if any) and source are model dependent.[4]
19	#XM	SIMD Floating-Point Exception	Fault	No	SSE/SSE2/SSE3 floating-point instructions[5]
20	#VE	Virtualization Exception	Fault	No	EPT violations[6]
21-31	—	Intel reserved. Do not use.			
32-255	—	User Defined (Non-reserved) Interrupts	Interrupt		External interrupt or INT *n* instruction.

Why Hardware Interrupts?

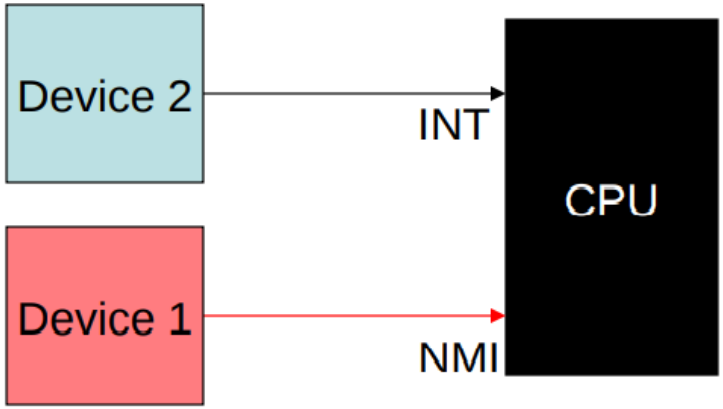
- Several devices connected to the CPU
  - eg. Keyboards, mouse, network card, etc.
- These devices occasionally need to be serviced by the CPU
  - eg. Inform CPU that a key has been pressed
- These events are asynchronous i.e. we cannot predict when they will happen.
- Need a way for the CPU to determine when a device needs attention

Possible Solution : Polling

- CPU periodically queries device to determine if they need attention
- Useful when device often needs to send information
  - For example in data acquisition systems
- If device does not need attention often,
  - Polling wastes CPU time

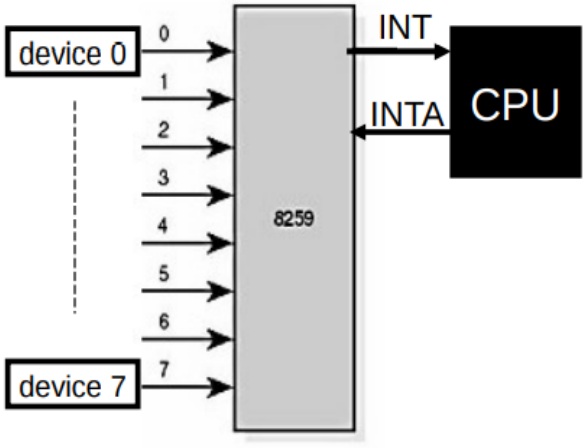
Interrupts

- Each device signals to the CPU that it wants to be serviced
- Generally CPUs have 2 pins
  - INT : Interrupt
  - NMI : Non maskable – for very critical signals
- How to support more than two interrupts?



8259 Programmable Interrupt Controller

- 8259 (Programmable interrupt controller) relays upto 8 interrupt to CPU
- Devices raise interrupts by an ‘interrupt request’ (IRQ)
- CPU acknowledges and queries the 8259 to determine which device interrupted
- Priorities can be assigned to each IRQ line
- 8259s can be cascaded to support more interrupts



System Calls

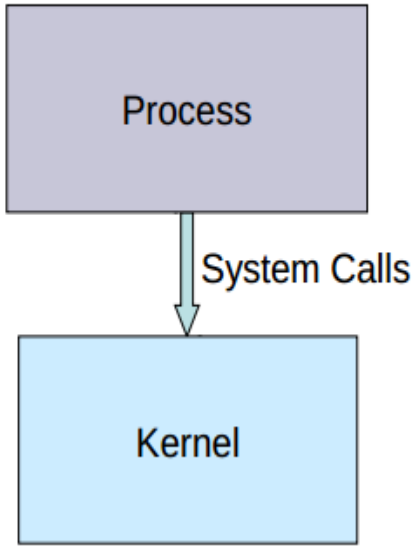
- Programming interface to the services provided by the OS
- Typically written in a high-level language (C or C++)
- Mostly accessed by programs via a high-level **Application Program Interface (API)** rather than direct system call use.
- Three most common APIs are Win32 API for Windows, POSIX API for 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)

Communicating with the OS (System Calls)

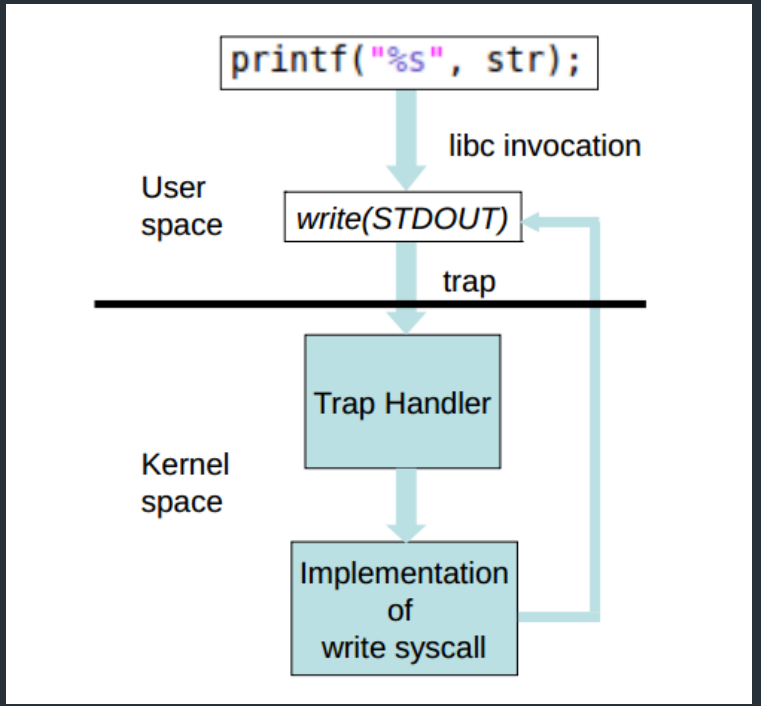
- System call invokes a function in the kernel using a Trap
- This causes
  - Processor to shift from user mode to privileged mode
- On completion of the system call, the execution gets transferred back to the user mode process



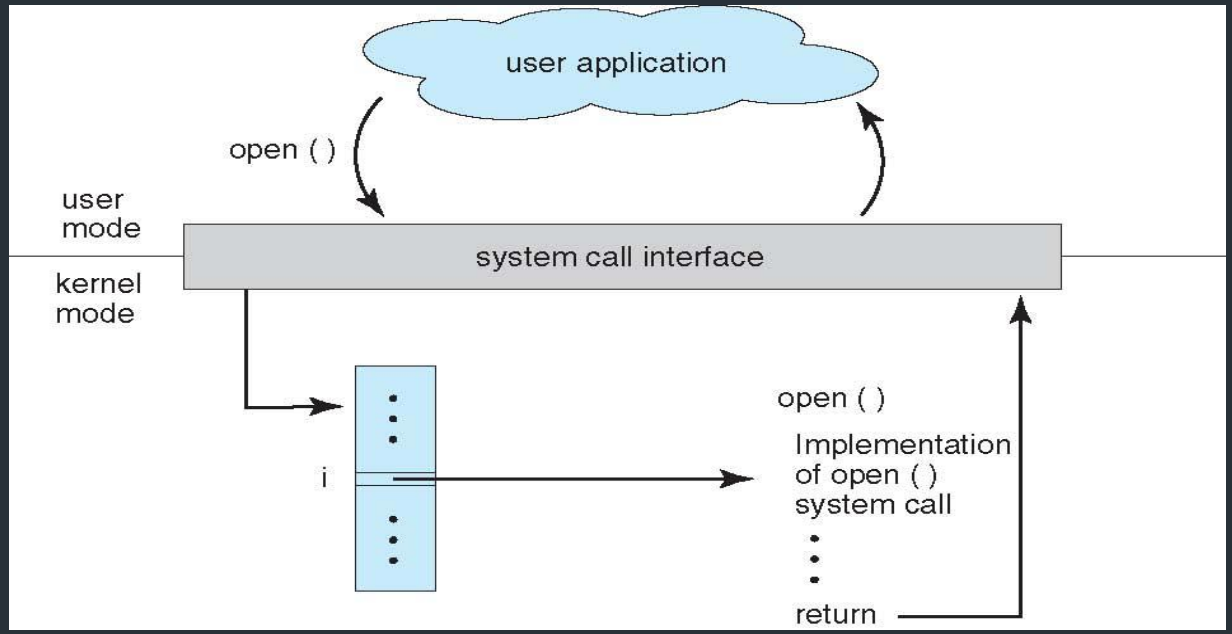
System Call Implementation

- 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)

Example (write system call)



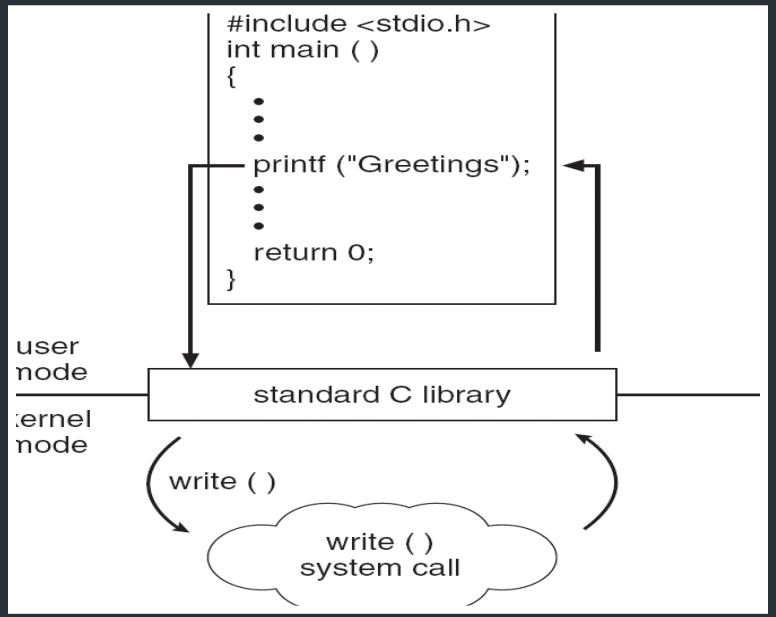
API – System Call – OS Relationship



Source: "Operating System Concepts: 7th Edition" by Silberschatz Galvin Gagne

Standard C Library Example

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



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

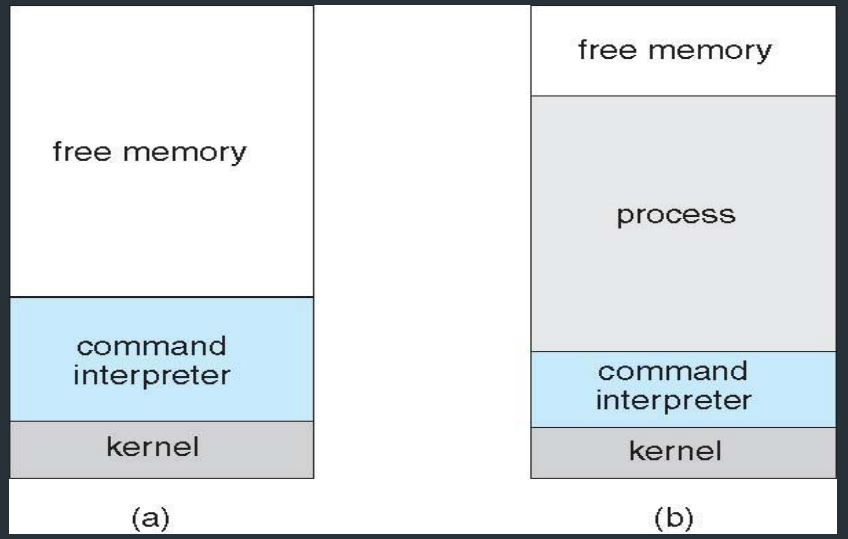
Examples of Windows and Unix System Calls

	Windows	Unix
Process Control	CreateProcess() 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()

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

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
- **File management** - Create, delete, copy, rename, print, dump, list, and generally manipulate files and directories

System Programs

- **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
- **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

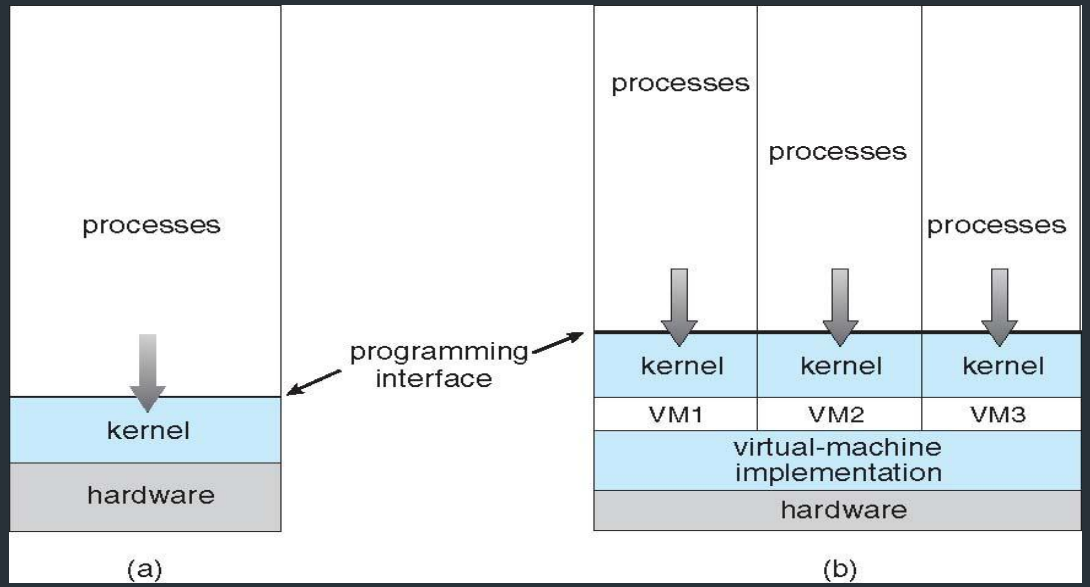
Virtual Machines

- 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

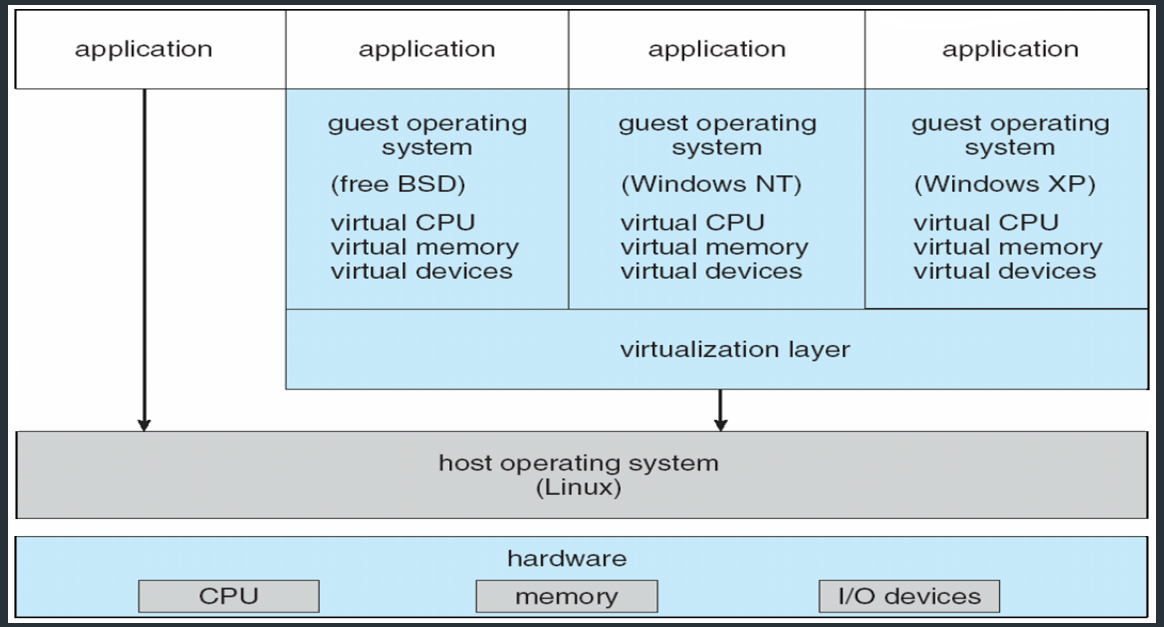
Virtual Machines (Cont.)



(a) Non Virtual machine (b) virtual machine

Source: "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

Thank You!