Computer System Operation

System Startup

- On power up
  - everything in system is in random, unpredictable state
  - special hardware circuit raises RESET pin of CPU
    - sets the program counter to 0xfffffff0
      - this address is mapped to ROM (Read-Only Memory)
- BIOS (Basic Input/Output Stream)
  - set of programs stored in ROM
  - some OS's use only these programs
    - MS DOS
  - many modern systems use these programs to load other system programs
    - Windows, Unix, Linux

BIOS

- General operations performed by BIOS
  1) find and test hardware devices
     - POST (Power-On Self-Test)
  2) initialize hardware devices
     - creates a table of installed devices
  3) find *boot sector*
     - may be on floppy, hard drive, or CD-ROM
  4) load boot sector into memory location 0x00007c00
  5) sets the program counter to 0x00007c00
     - starts executing code at that address

Boot Loader

- Small program stored in boot sector
- Loaded by BIOS at location 0x00007c0
- Configure a basic file system to allow system to read from disk
- Loads kernel into memory
- Also loads another program that will begin kernel initialization

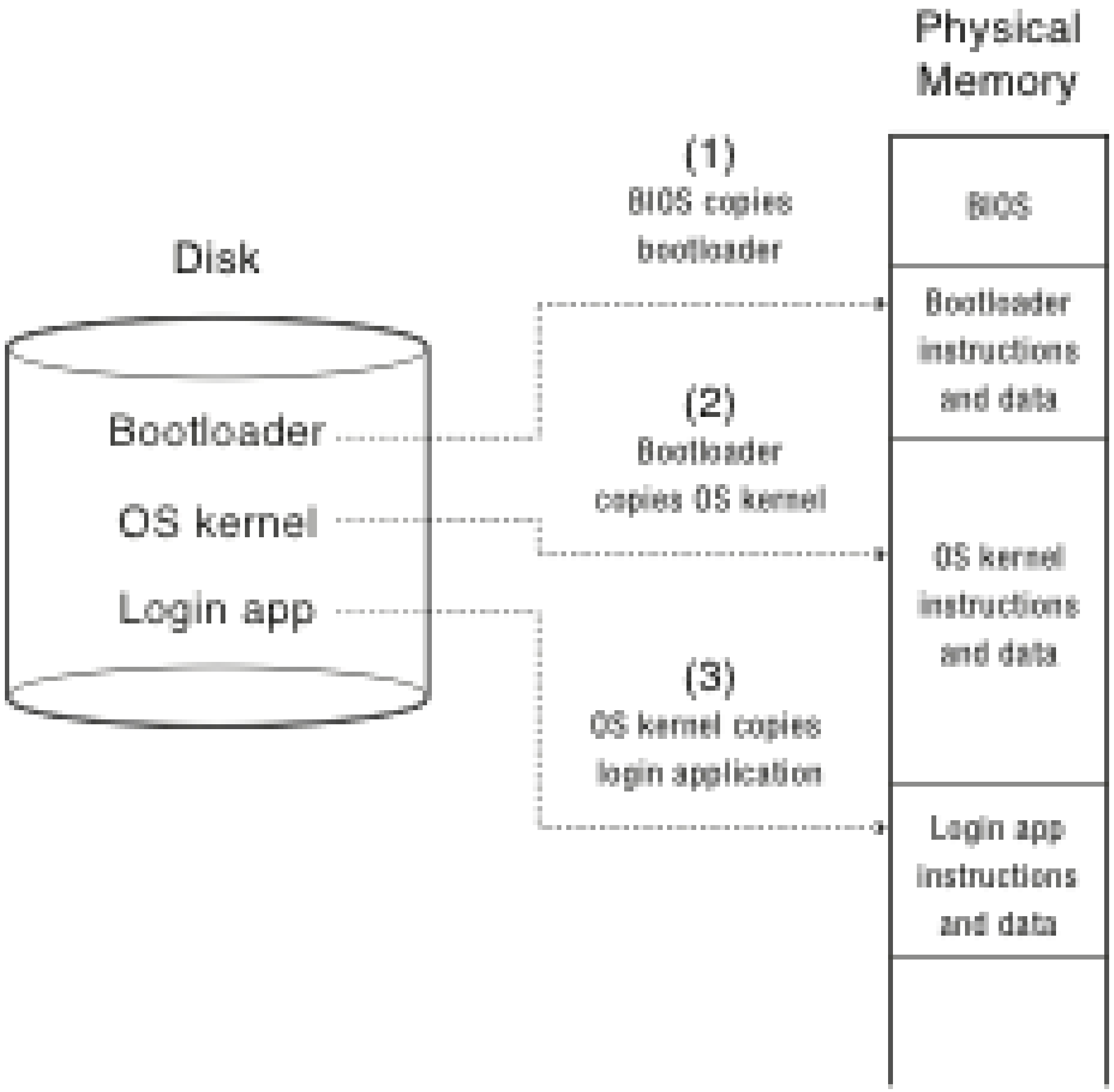
Initial Kernel Program

- Determines amount of RAM in system
  - uses a BIOS function to do this
- Configures hardware devices
  - video card, mouse, disks, etc.
  - BIOS may have done this but usually redo it
    - portability
- Switches the CPU from *real* to *protected* mode
  - real mode: fixed segment sizes, 1 MB memory addressing, and no segment protection
  - protected mode: variable segment sizes, 4 GB memory addressing, and provides segment protection
- Initializes paging (virtual memory)

Final Kernel Initialization

- Sets up page tables and segment descriptor tables
  - these are used by virtual memory and segmentation hardware
- Sets up interrupt vector and enables interrupts
- Initializes all other kernel data structures (Linked lists, Binary search trees, Bitmaps etc.)
- Creates initial process and starts it running
  - *init* in Linux
  - *smss* (Session Manager SubSystem) in NT

Booting



Physical Memory
Disk
Bootloader
OS kernel
Login app
(1)
BIOS copies bootloader
(2)
Bootloader copies OS kernel
(3)
OS kernel copies login application
BIOS
Bootloader instructions and data
OS kernel instructions and data
Login app instructions and data

System Programs

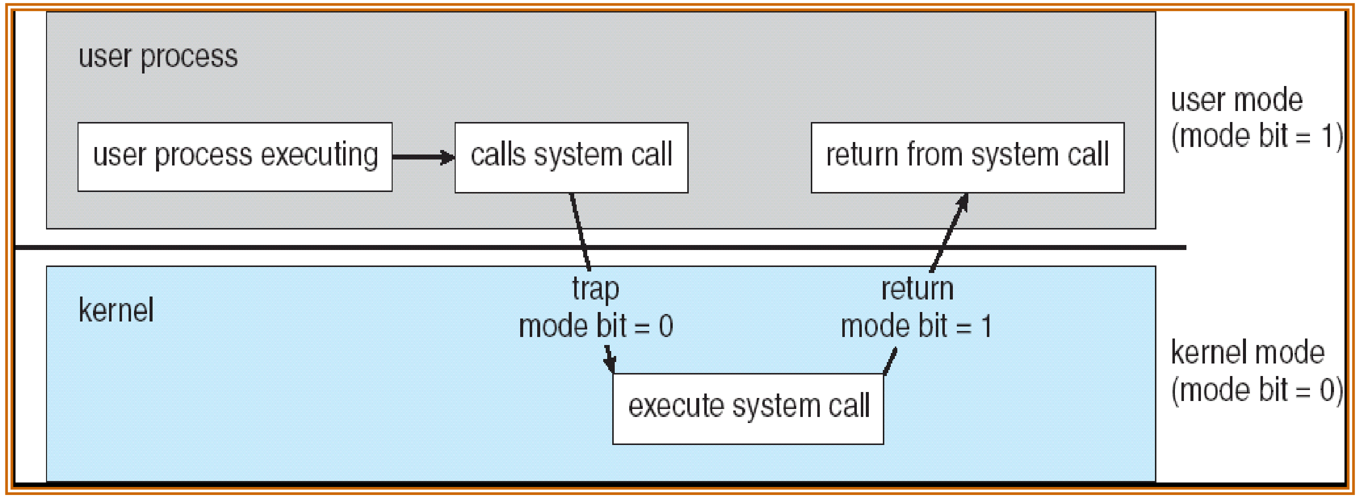
- Application programs included with the OS
- Highly trusted programs
- Perform useful work that most users need
  - listing and deleting files, configuring system
  - ls, rm, Windows Explorer and Control Panel
  - may include compilers and text editors
- Not part of the OS
  - run in user space
- Very useful

Operating-System Operations

- Interrupt driven by hardware
- Software error or request creates **exception** or **trap**
  - Division by zero, request for operating system service
- Other process problems include infinite loop, processes modifying each other or the operating system
- **Dual-mode** operation allows OS to protect itself and other system components
- Hardware provides at least two modes:
  - “Kernel” mode (or “supervisor” or “protected”)
  - “User” mode: Normal programs executed
  - **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

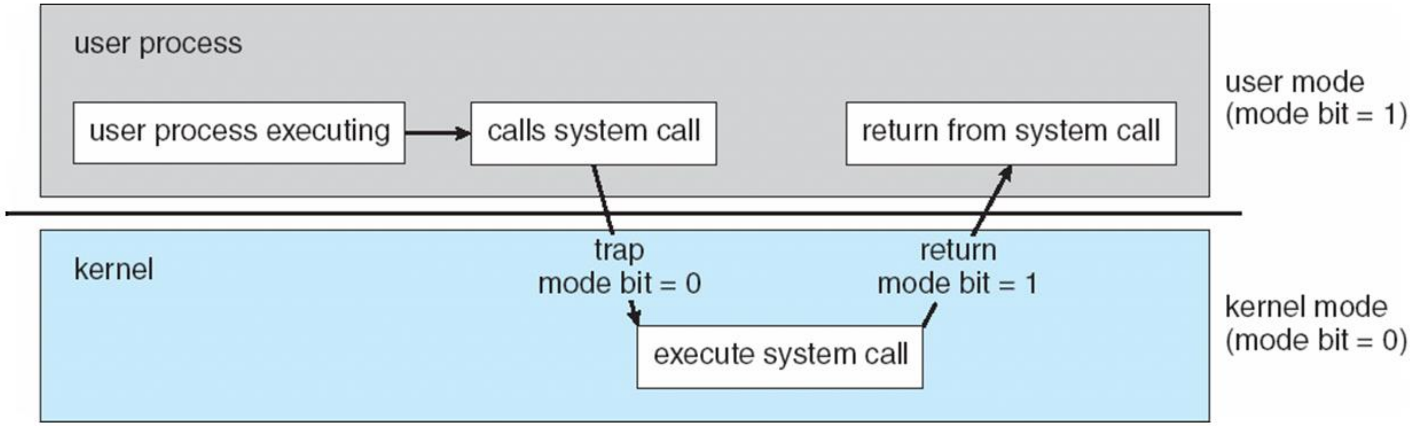
Dual Mode Operation

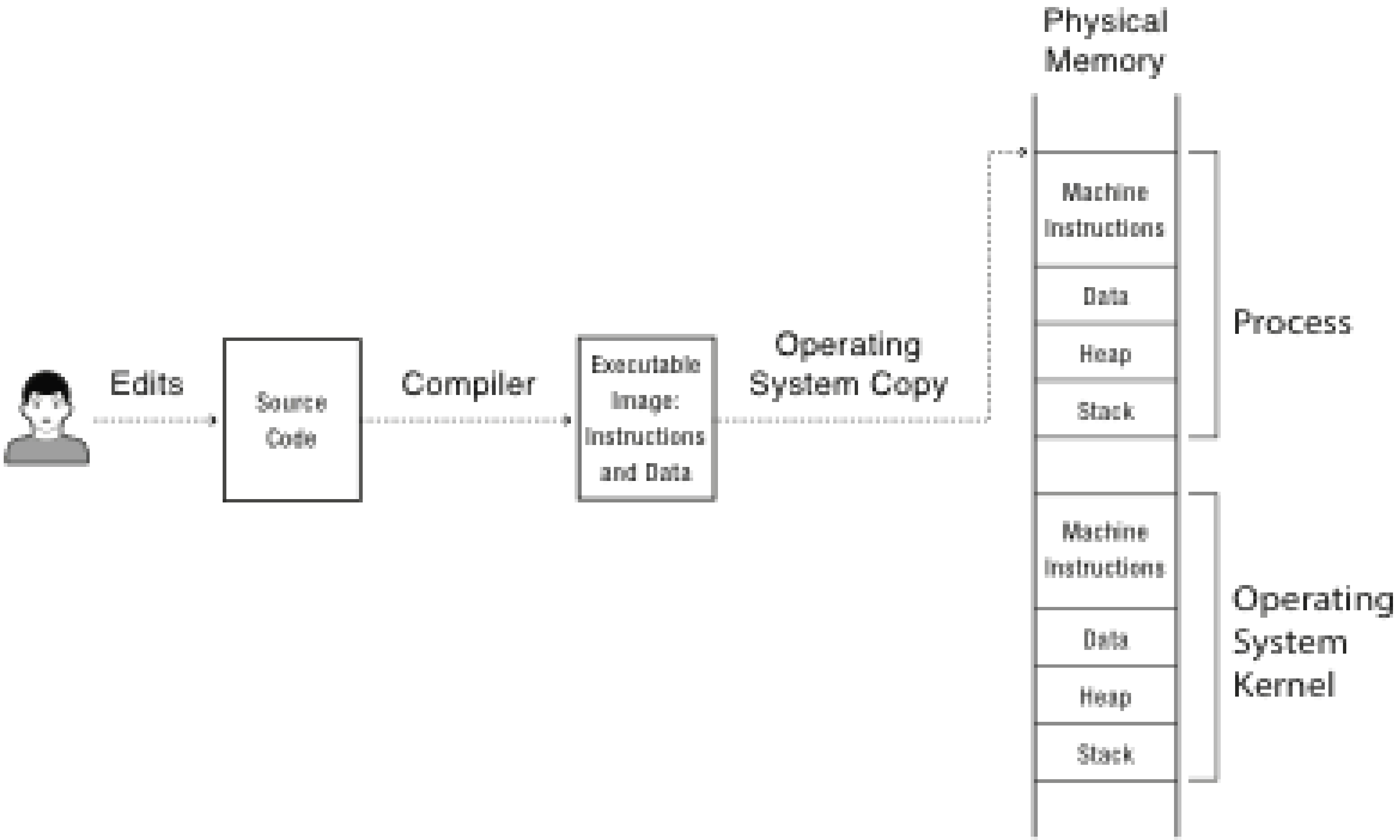
- Some instructions/ops prohibited in user mode:
  - Example: cannot modify page tables in user mode
    - Attempt to modify $\Rightarrow$ Exception generated
- Transitions from user mode to kernel mode:
  - System Calls, Interrupts, Other exceptions



Transition from User to Kernel Mode

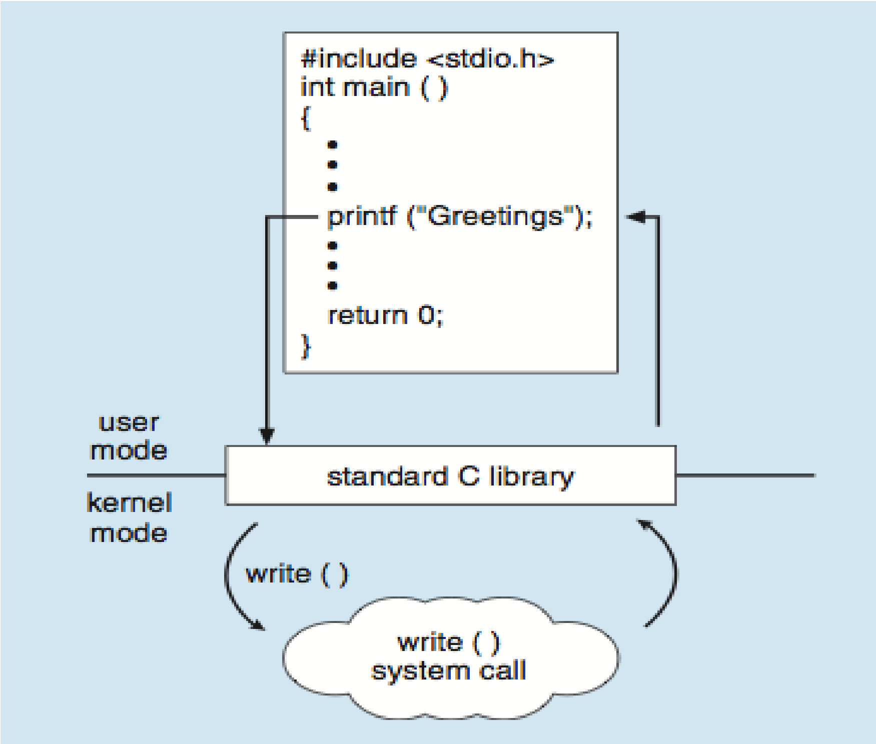
- Timer to prevent infinite loop / process hogging resources
  - Set interrupt after specific period
  - Operating system decrements counter
  - When counter zero generate an interrupt
  - Set up before scheduling process to regain control or terminate program that exceeds allotted time




Edits
Source Code
Compiler
Executable Image: Instructions and Data
Operating System Copy
Physical Memory
Machine Instructions
Data
Heap
Stack
Process
Machine Instructions
Data
Heap
Stack
Operating System Kernel

Standard C Library Example

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



System Calls

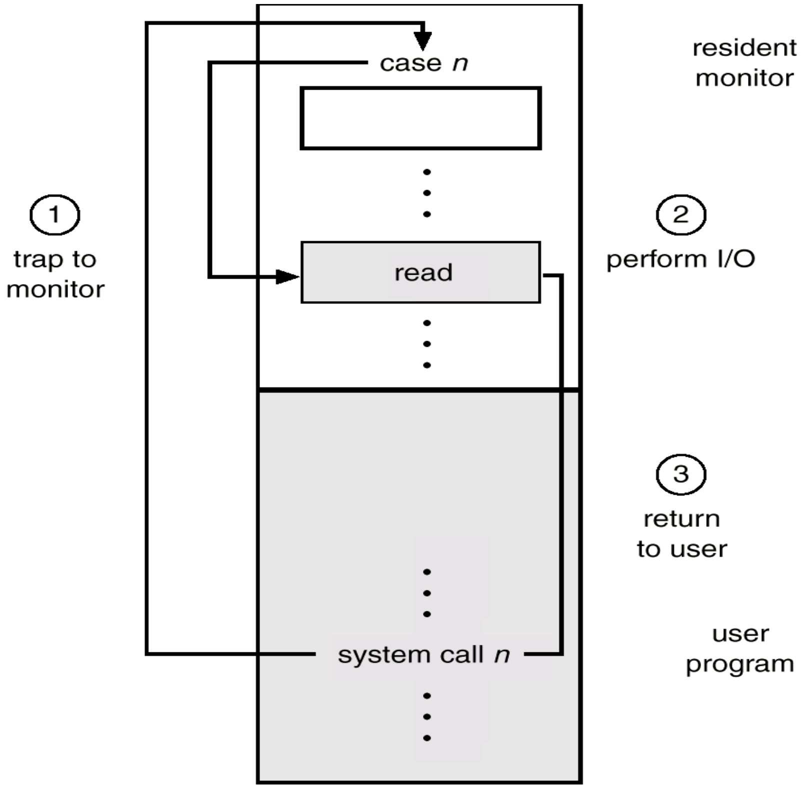
- System calls provide the interface between a running program and the operating system.
  - Generally available as assembly-language instructions.
  - Languages have been defined to replace assembly language for systems programming; allow system calls to be made directly (e.g., C, C++)
- Three general methods are used to pass parameters between a running program and the operating system.
  - 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.

System Calls

- System calls are routines run by the OS on behalf of the user
- Allow user to access I/O, create processes, get system information, etc.
- 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 Programming 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)

System Calls

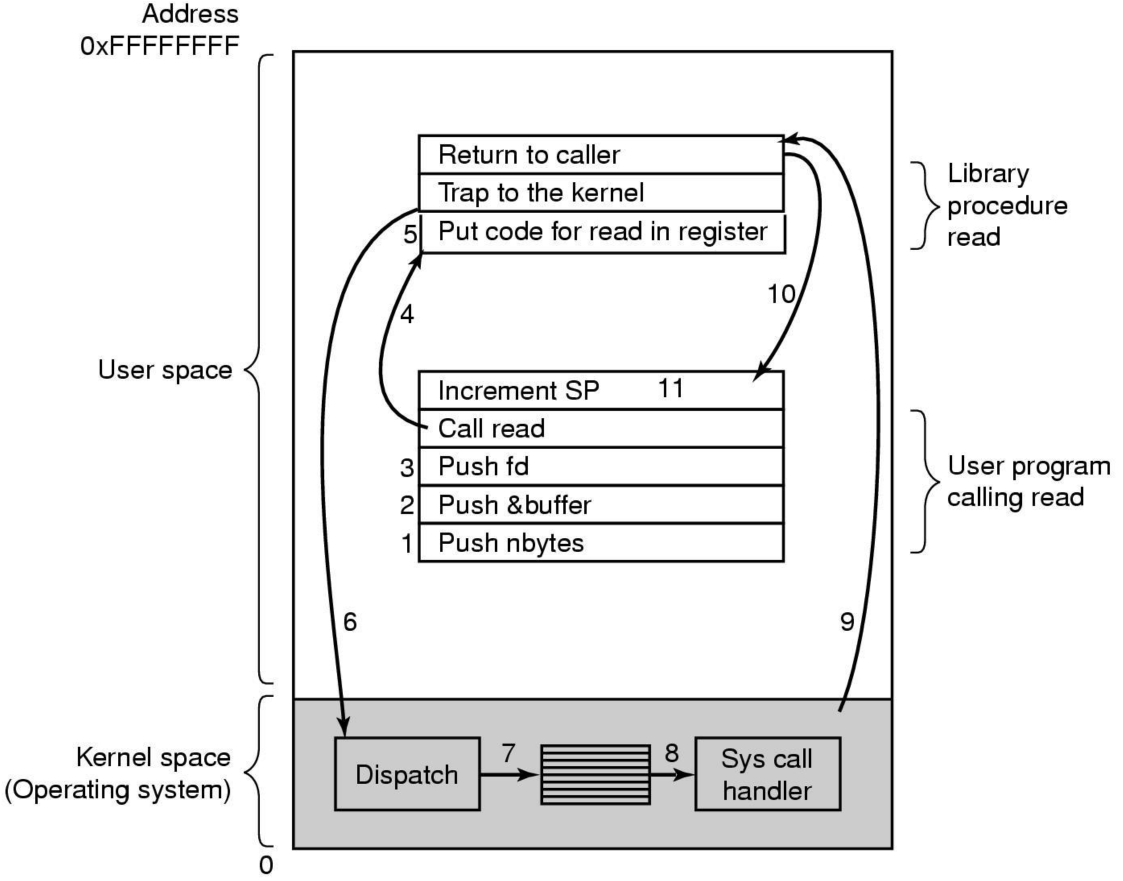
- A System Call is the main way a user program interacts with the Operating System.



System Calls

HOW A SYSTEM CALL WORKS

- Obtain access to system space
- Do parameter validation
- System resource collection (locks on structures)
- Ask device/system for requested item
- Suspend waiting for device
- Interrupt makes this thread ready to run
- Wrap-up
- Return to user



There are 11 (or more) steps in making the system call

read (fd, buffer, nbytes)

Linux API

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

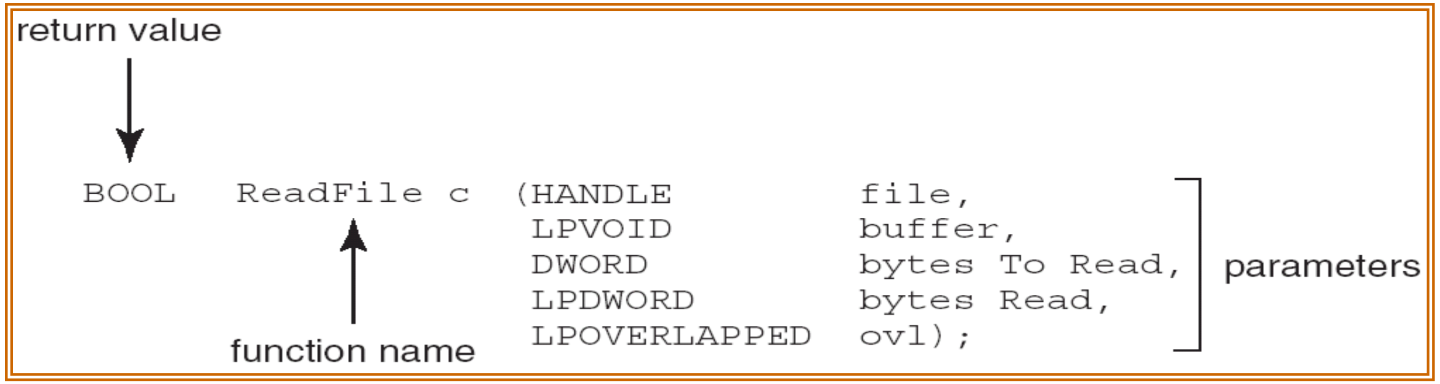
A program that uses the `read()` function must include the `unistd.h` header file, as this file defines the `ssize_t` and `size_t` data types (among other things). The parameters passed to `read()` are as follows:

- `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.

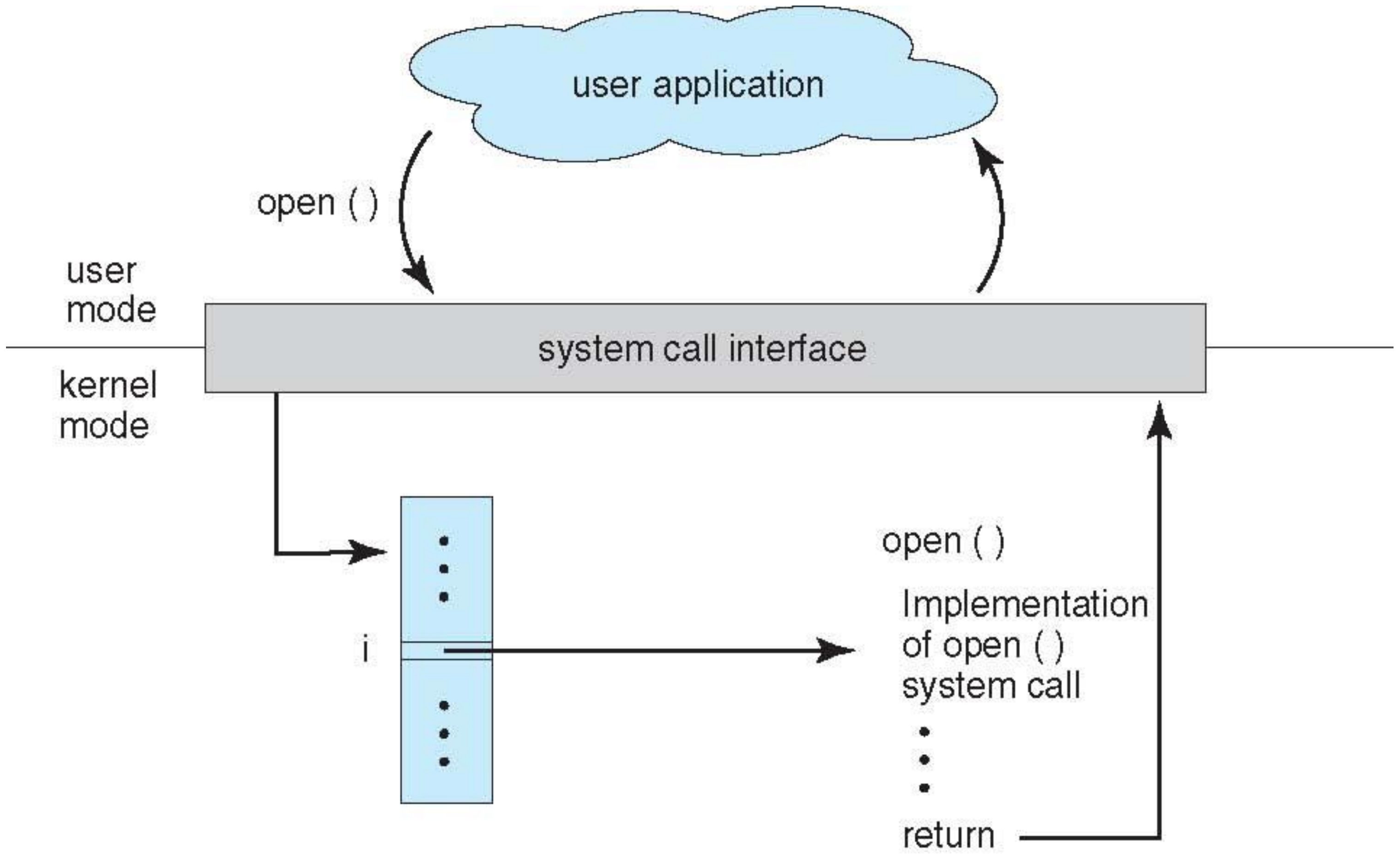
Example of Windows API

- Consider the ReadFile() function in the program
- 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

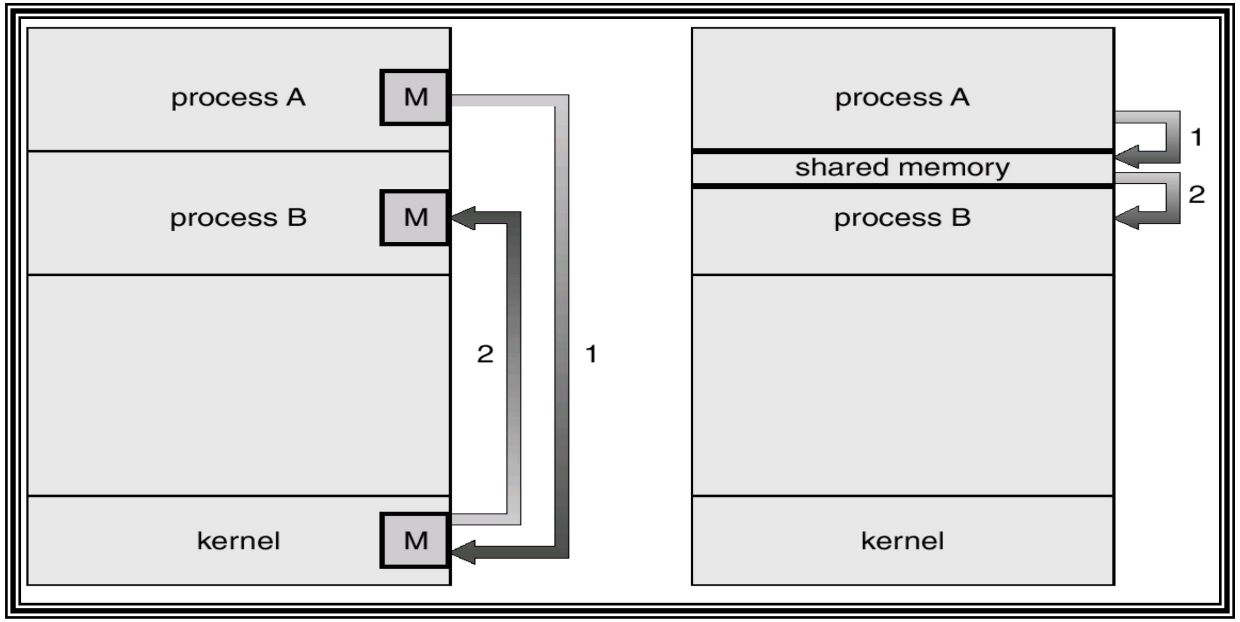
API – System Call – OS Relationship



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

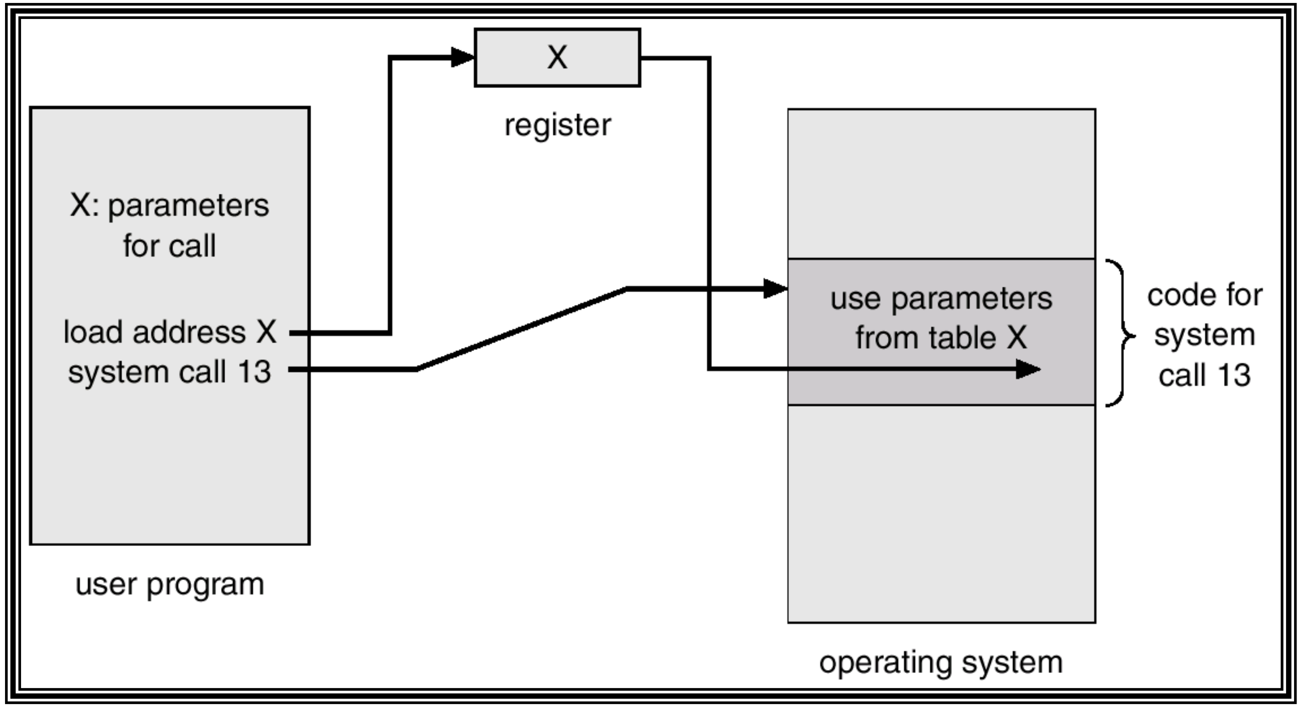
passing data between programs



Msg Passing

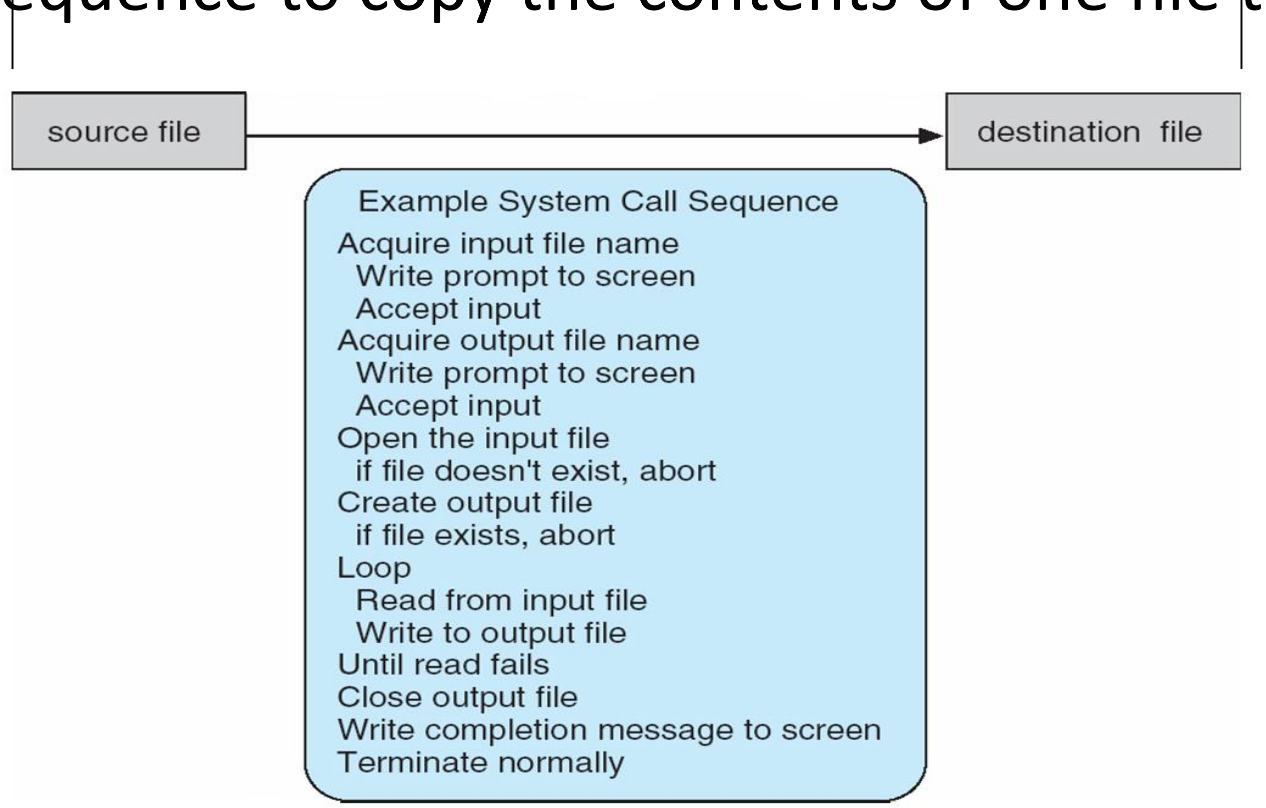
Shared Memory

Passing of Parameters As A Table



Example of System Calls

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



Send message from one processor to another

Operations to be performed:

- Check Permissions, Format Message
- Enforce forward progress,

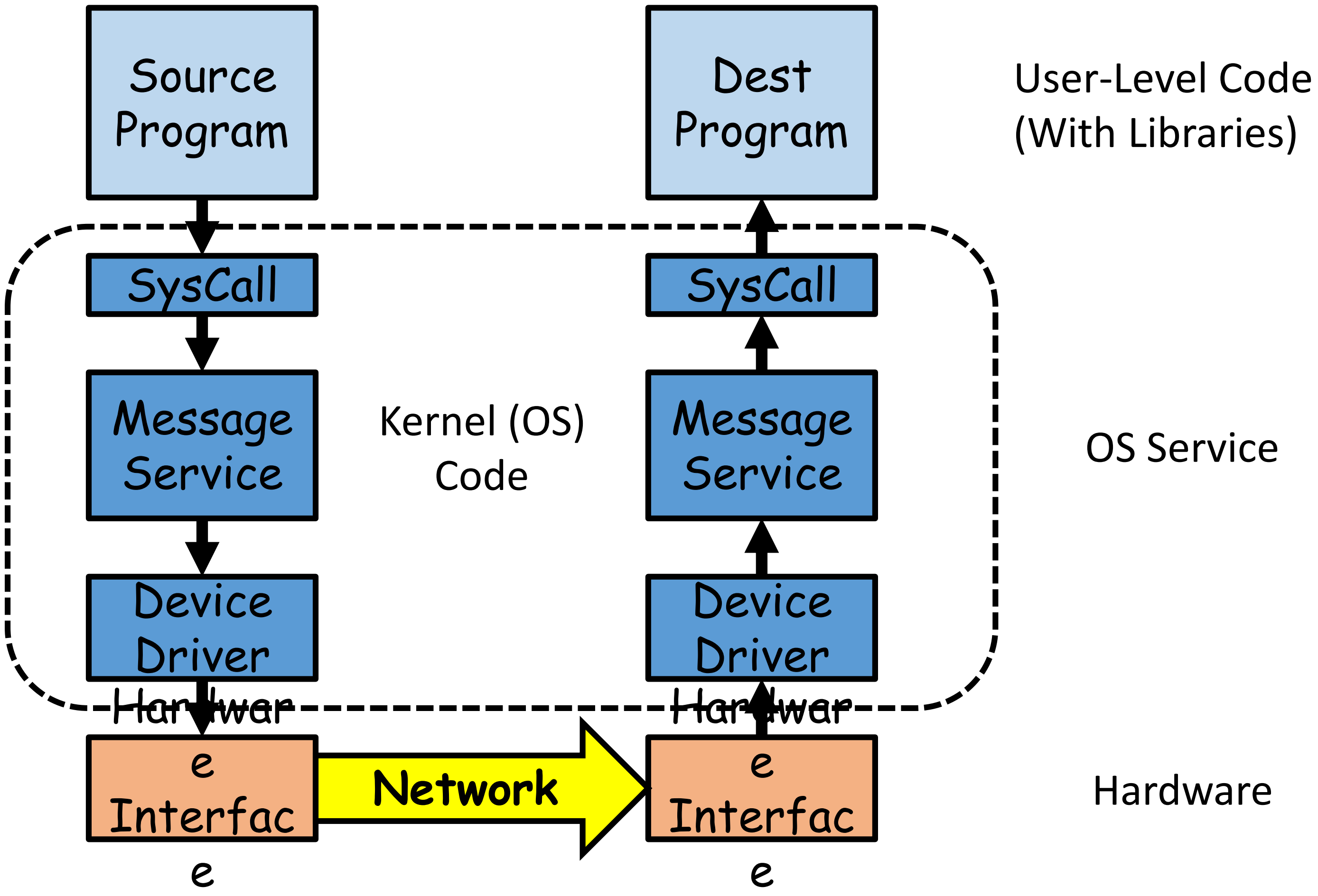
Handle interrupts

- Prevent Denial Of Service (DOS)

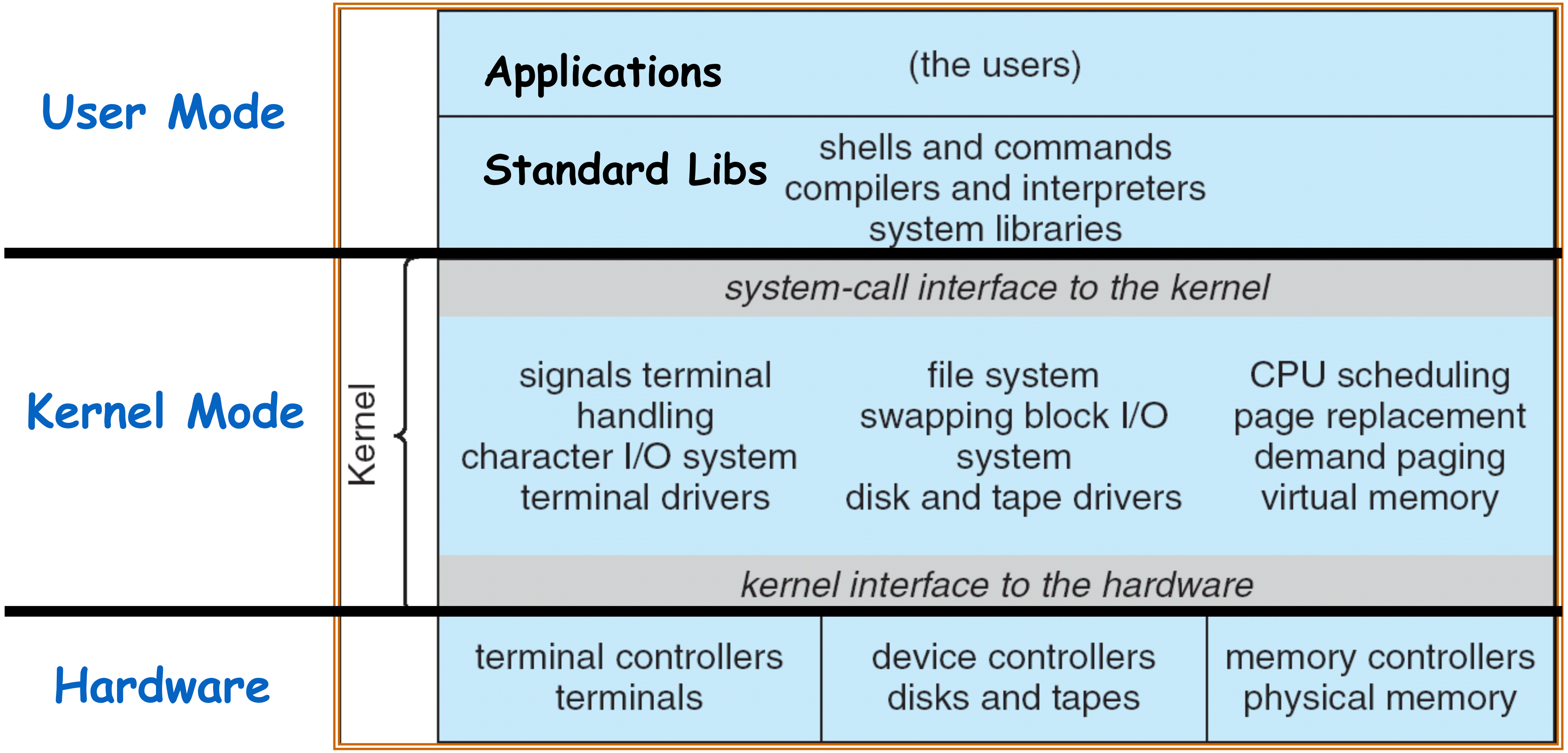
and/or Deadlock

Traditional Approach:

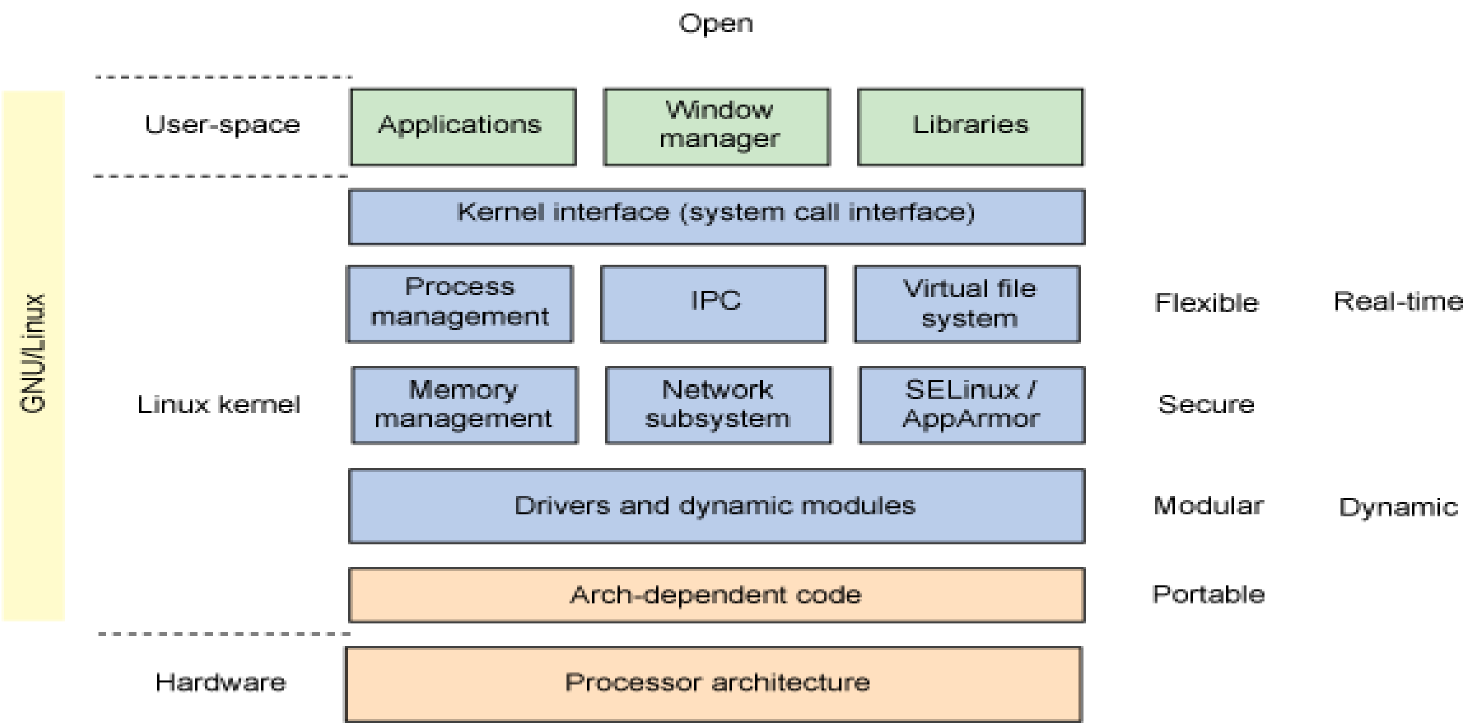
Use a system call + OS Service



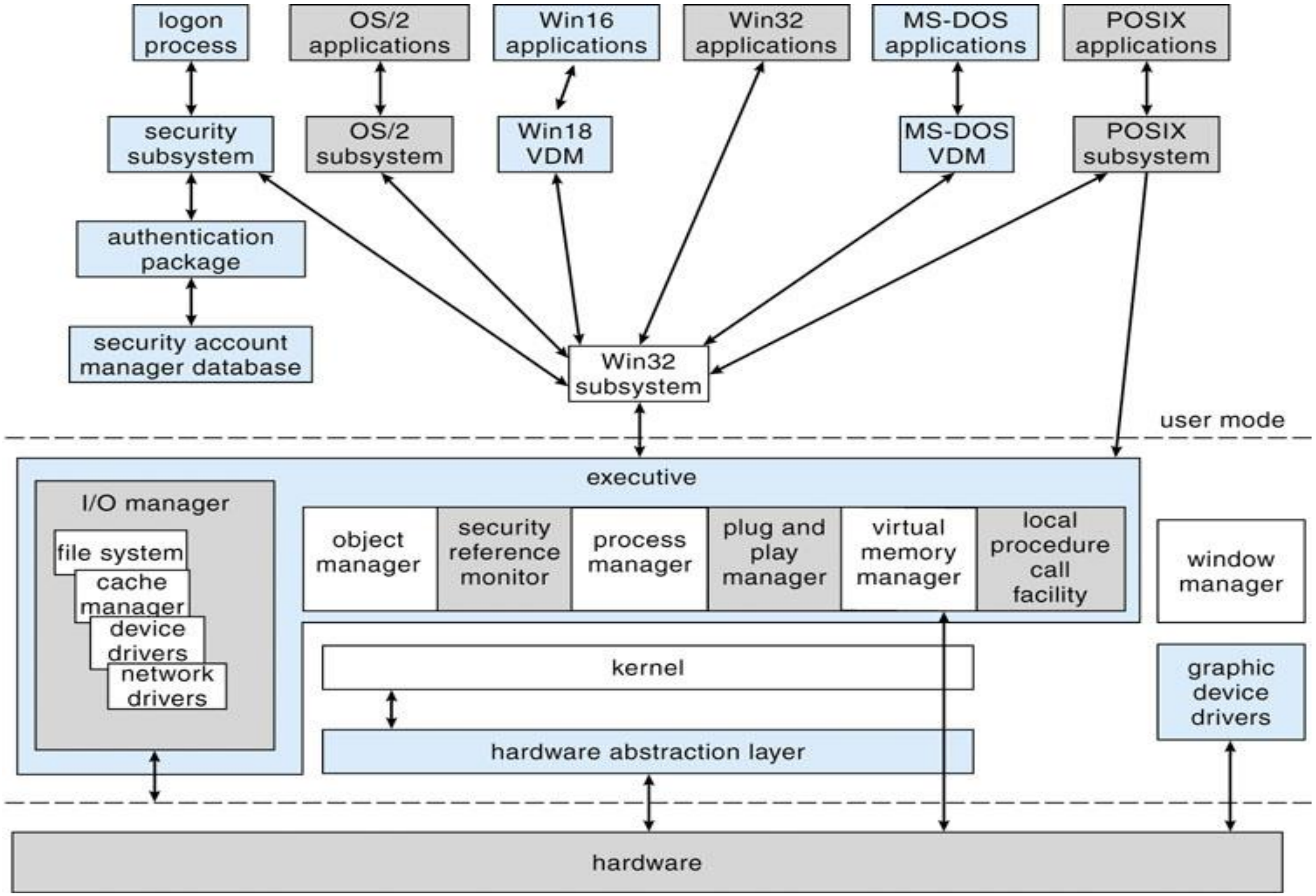
UNIX System Structure



Linux Structure



Microsoft Windows Structure



Major Windows Components

- Hardware Abstraction Layer
  - Hides hardware chipset differences from upper levels of OS
- Kernel Layer
  - Thread Scheduling
  - Low-Level Processors Synchronization
  - Interrupt/Exception Handling
  - Switching between User/Kernel Mode.
- Executive
  - Set of services that all environmental subsystems need
    - Object Manager
    - Virtual Memory Manager
    - Process Manager
    - Advanced Local Procedure Call Facility
    - I/O manager
    - Cache Manager
    - Security Reference Monitor
    - Plug-and-Plan and Power Managers
    - Registry
    - Booting
- Programmer Interface: Win32 API

Types of System Calls

- Process control
- File manipulation
- Device manipulation
- Information maintenance
- Communications
- Protection

Types of System Calls

- Process control
  - create process, terminate process
  - end, abort
  - load, execute
  - get process attributes, set process attributes
  - wait for time
  - wait event, signal event
  - allocate and free memory
  - Dump memory if error
  - **Debugger** for determining **bugs, single step** execution
  - **Locks** for managing access to shared data between processes

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

- 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 if **message passing model** to **host name** or **process name**
    - From **client** to **server**
  - **Shared-memory model** create and gain access to memory regions
  - transfer status information
  - attach and detach remote devices

Types of System Calls

- 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() 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()

UNIX	Win32	Description
fork	CreateProcess	Create a new process
waitpid	WaitForSingleObject	Can wait for a process to exit
execve	(none)	CreateProcess = fork + execve
exit	ExitProcess	Terminate execution
open	CreateFile	Create a file or open an existing file
close	CloseHandle	Close a file
read	ReadFile	Read data from a file
write	WriteFile	Write data to a file
lseek	SetFilePointer	Move the file pointer
stat	GetFileAttributesEx	Get various file attributes
mkdir	CreateDirectory	Create a new directory
rmdir	RemoveDirectory	Remove an empty directory
link	(none)	Win32 does not support links
unlink	DeleteFile	Destroy an existing file
mount	(none)	Win32 does not support mount
umount	(none)	Win32 does not support mount
chdir	SetCurrentDirectory	Change the current working directory
chmod	(none)	Win32 does not support security (although NT does)
kill	(none)	Win32 does not support signals
time	GetLocalTime	Get the current time

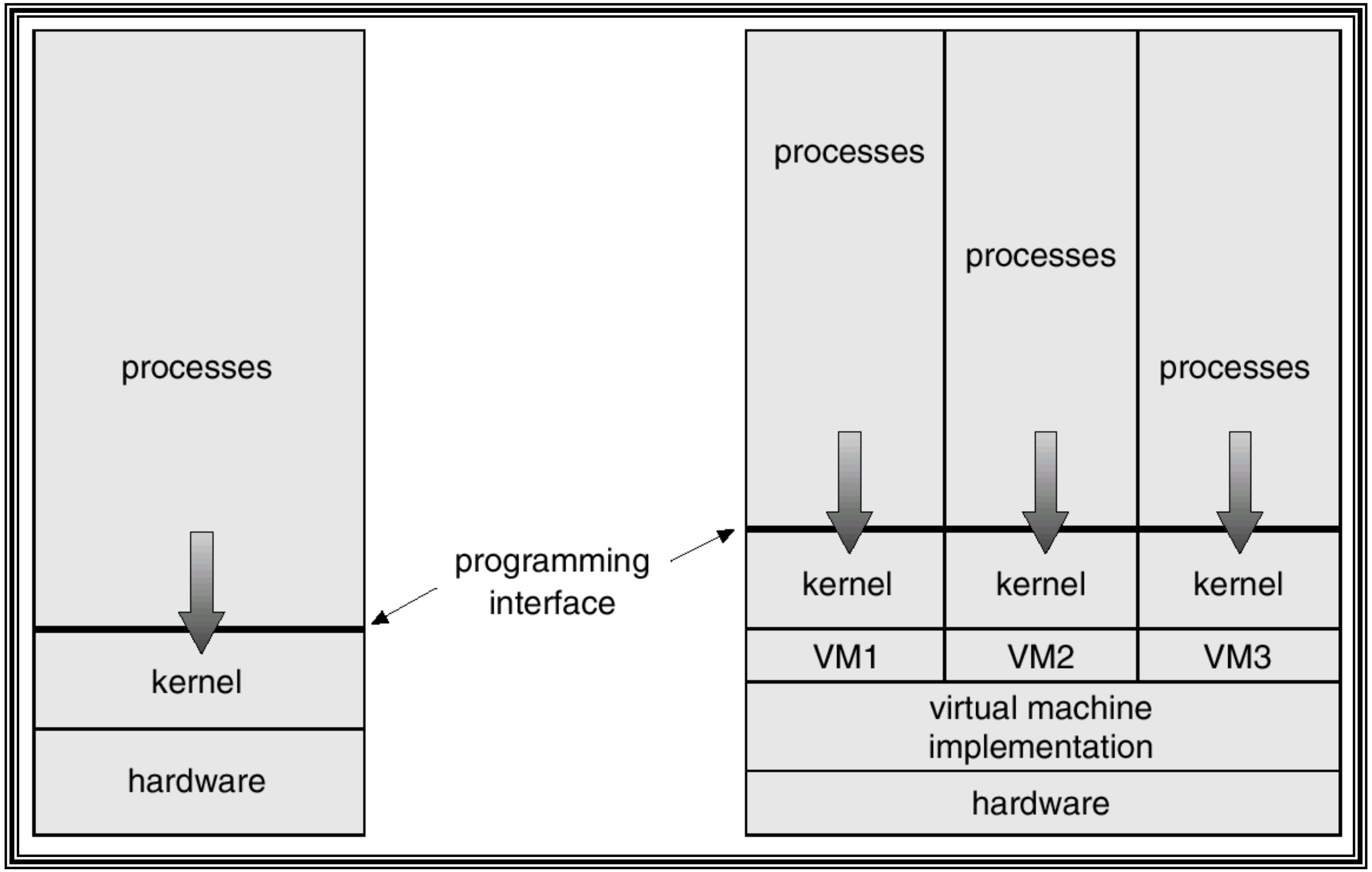
- The concept of modes of operation in operating system can be extended beyond the dual mode. This is known as the multimode system. In those cases the more than 1 bit is used by the CPU to set and handle the mode.
- An example of the multimode system can be described by the systems that support virtualisation. These CPU’s have a separate mode that specifies when the virtual machine manager (VMM) and the virtualisation management software is in control of the system.
- For these systems, the virtual mode has more privileges than user mode but less than kernel mode.

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 creates the illusion of multiple processes, each executing on its own processor with its own (virtual) memory.

- The resources of the physical computer are shared to create the virtual machines.
  - CPU scheduling can create the appearance that users have their own processor.
  - Spooling and a file system can provide virtual card readers and virtual line printers.
  - A normal user time-sharing terminal serves as the virtual machine operator’s console.

System Models



Advantages/Disadvantages of Virtual Machines

- The virtual-machine concept provides complete protection of system resources since each virtual machine is isolated from all other virtual machines. This isolation, however, permits no direct sharing of resources.
- A virtual-machine system is a perfect vehicle for operating-systems research and development. System development is done on the virtual machine, instead of on a physical machine and so does not disrupt normal system operation.
- The virtual machine concept is difficult to implement due to the effort required to provide an *exact* duplicate to the underlying machine.