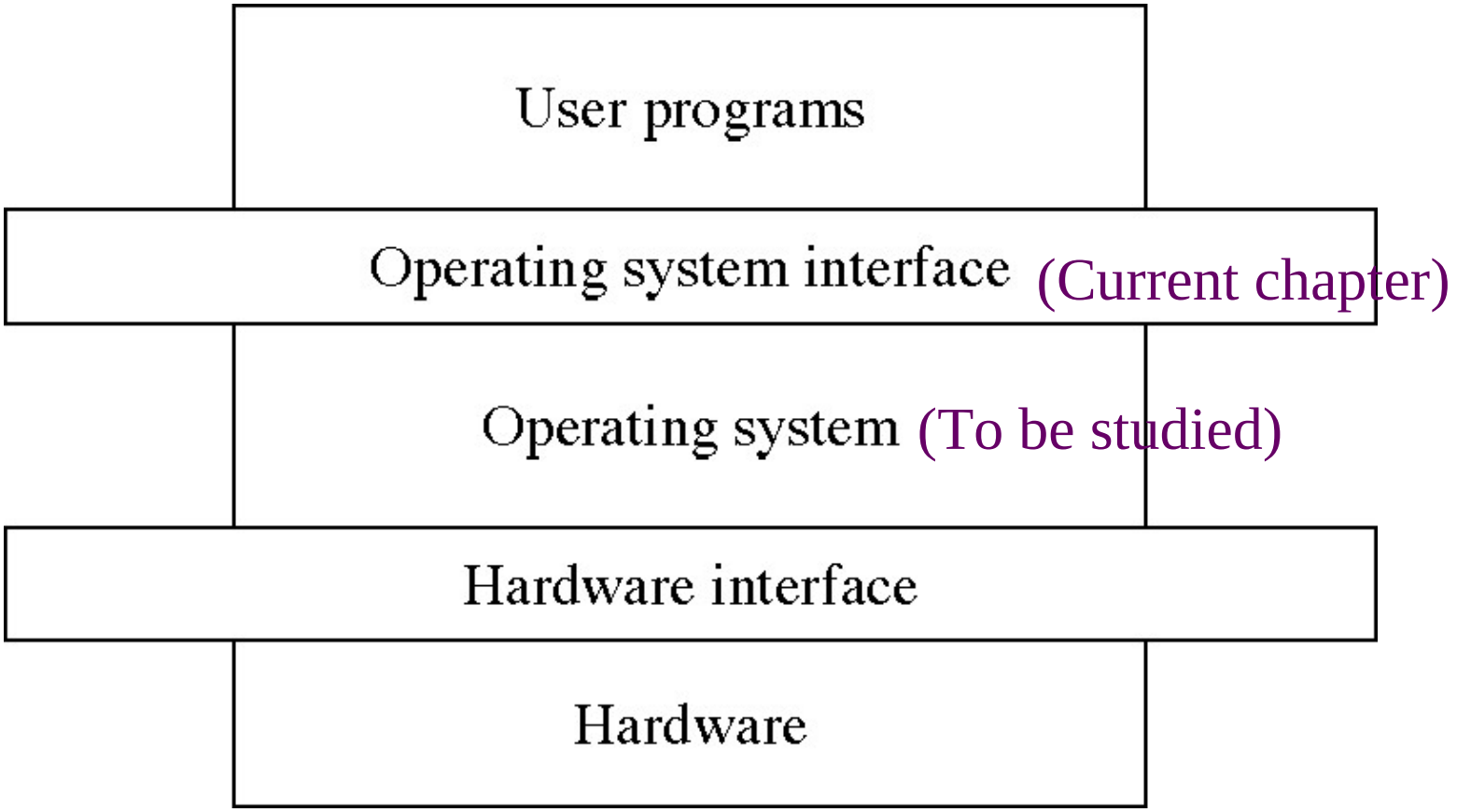
Chapter 2: System Structures

Reference: Chapter 2 of Silberschatz et al book, Operating System Concepts, 8th edition.

The OS Level Structure



Chapter 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
- To explain how operating systems are installed and customized and how they boot

Outline

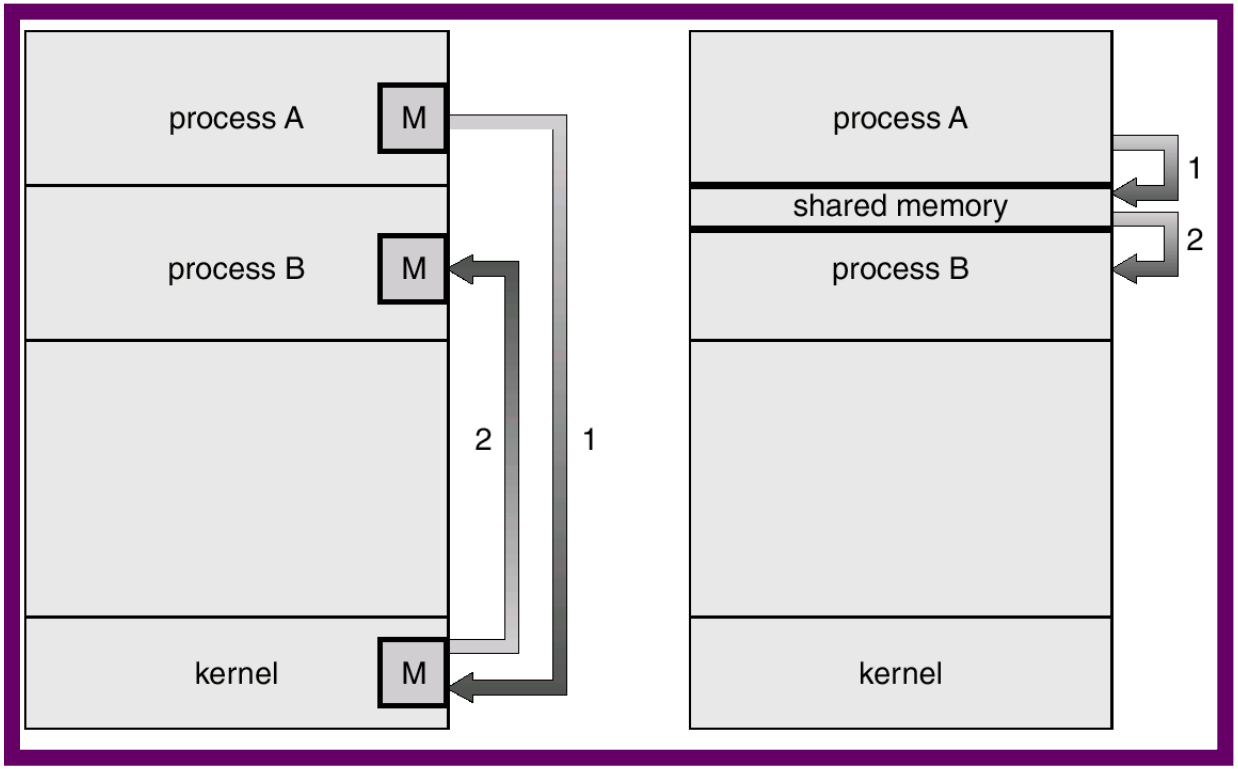
- **Operating System Services**
- User Operating System Interface
- System Calls
- Types of System Calls
- System Programs
- Operating System Design and Implementation
- Operating System Structure
- Virtual Machines
- Operating System Debugging
- Operating System Generation
- System Boot

Operating System Services

- 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)**, **Graphics User Interface (GUI)**, **Batch**
  - **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
  - **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.

Operating System Services (Cont.)

- **Communications** – Processes may exchange information, on the same computer or between computers over a **network**
  - Communications may be via shared memory or through message passing (packets moved by the OS)



Message Passing

Shared Memory

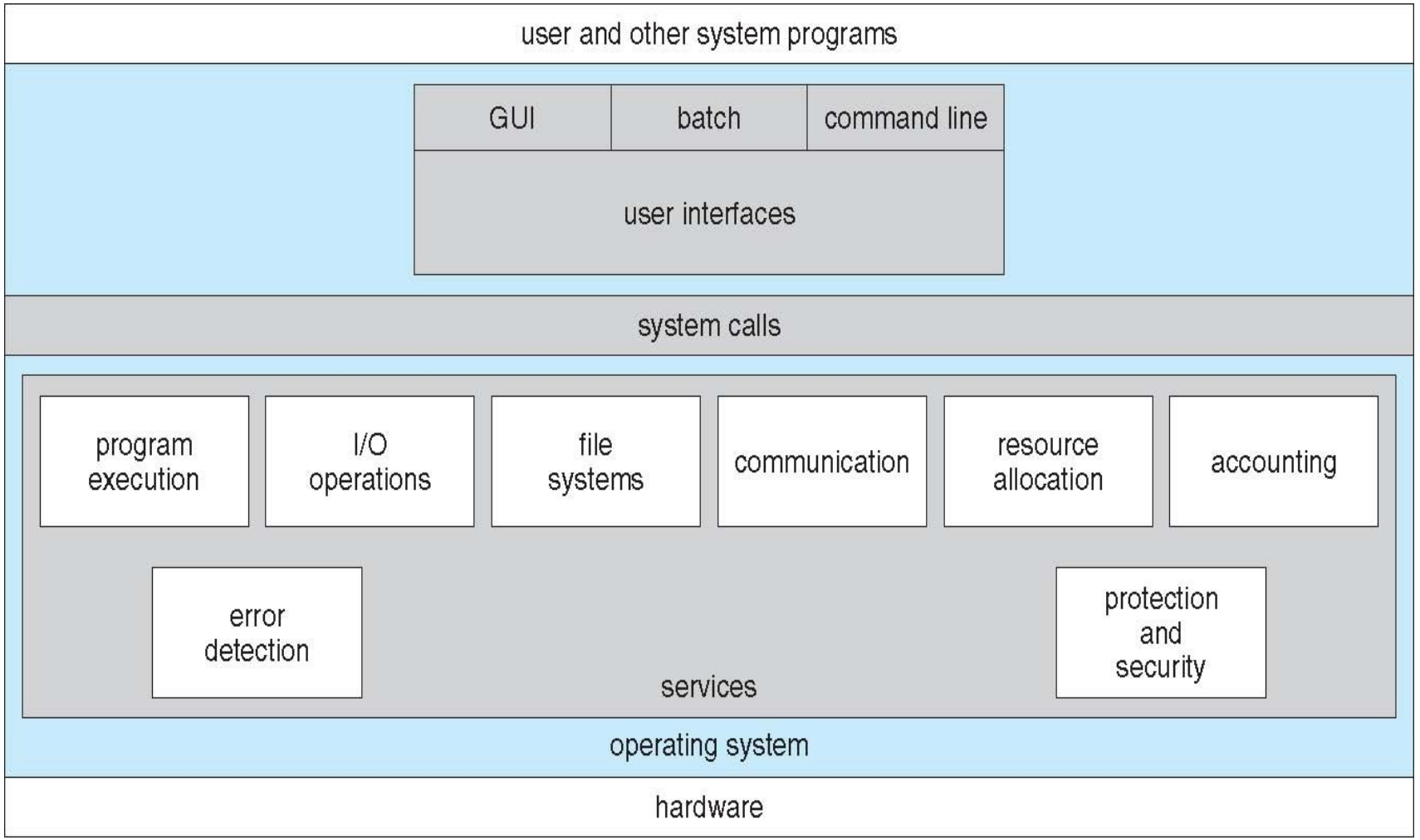
Operating System Services (Cont.)

- **Error detection** – OS needs to be constantly aware of possible errors
  - May occur in the CPU and memory hardware, in I/O devices, in user program
  - For each type of error, OS should take the appropriate action to ensure correct and consistent computing
  - Debugging facilities can greatly enhance the user's and programmer's abilities to efficiently use the system

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 - Some (such as CPU cycles, main memory, and file storage) may have special allocation code, others (such as I/O devices) may have general request and release code
  - **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 system 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
    - If a system is to be protected and secure, precautions must be instituted throughout it. A chain is only as strong as its weakest link.

A View of Operating System Services



Outline

- Operating System Services
- **User Operating System Interface**
- System Calls
- Types of System Calls
- System Programs
- Operating System Design and Implementation
- Operating System Structure
- Virtual Machines
- Operating System Debugging
- Operating System Generation
- System Boot

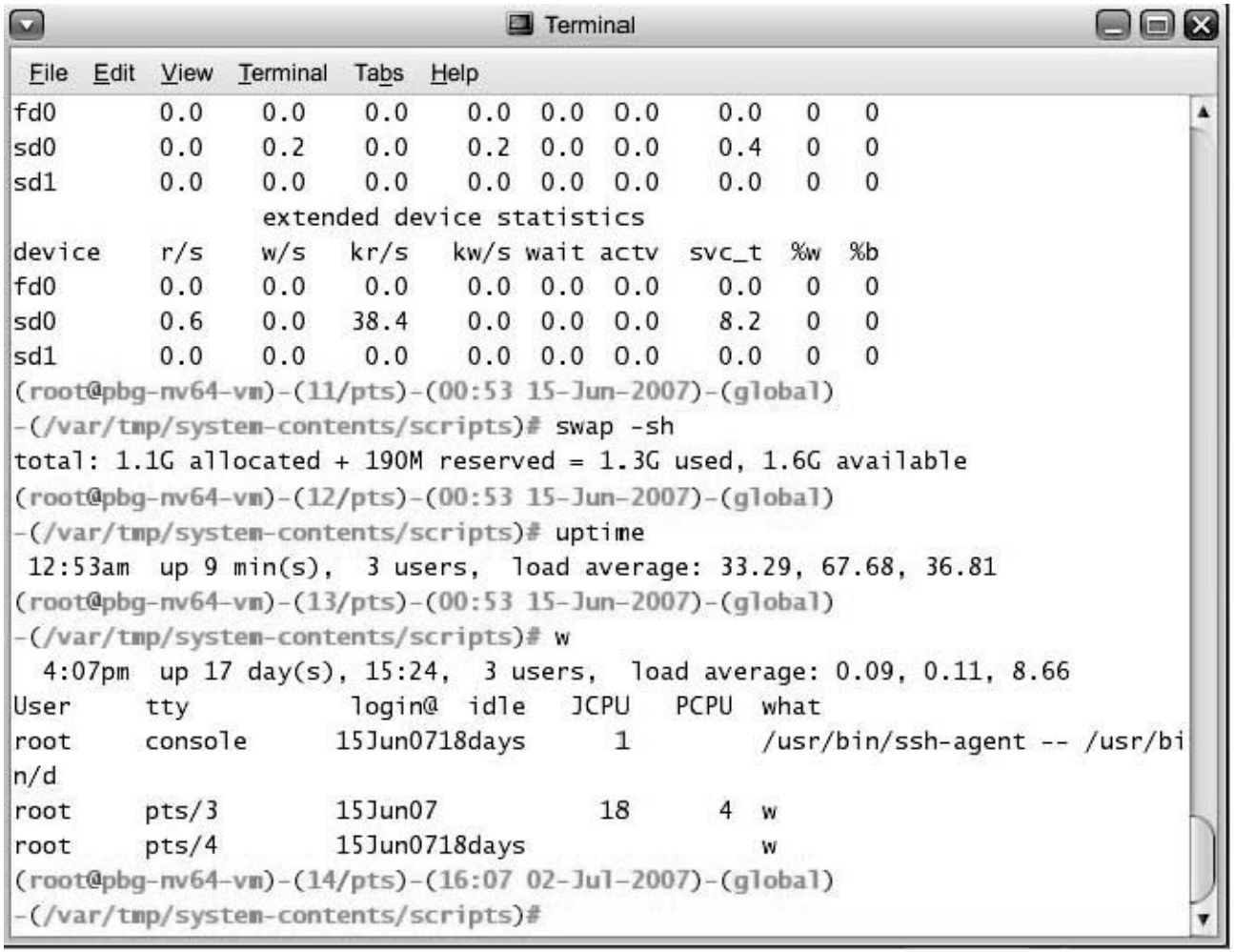
User Operating System Interface - CLI

- 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

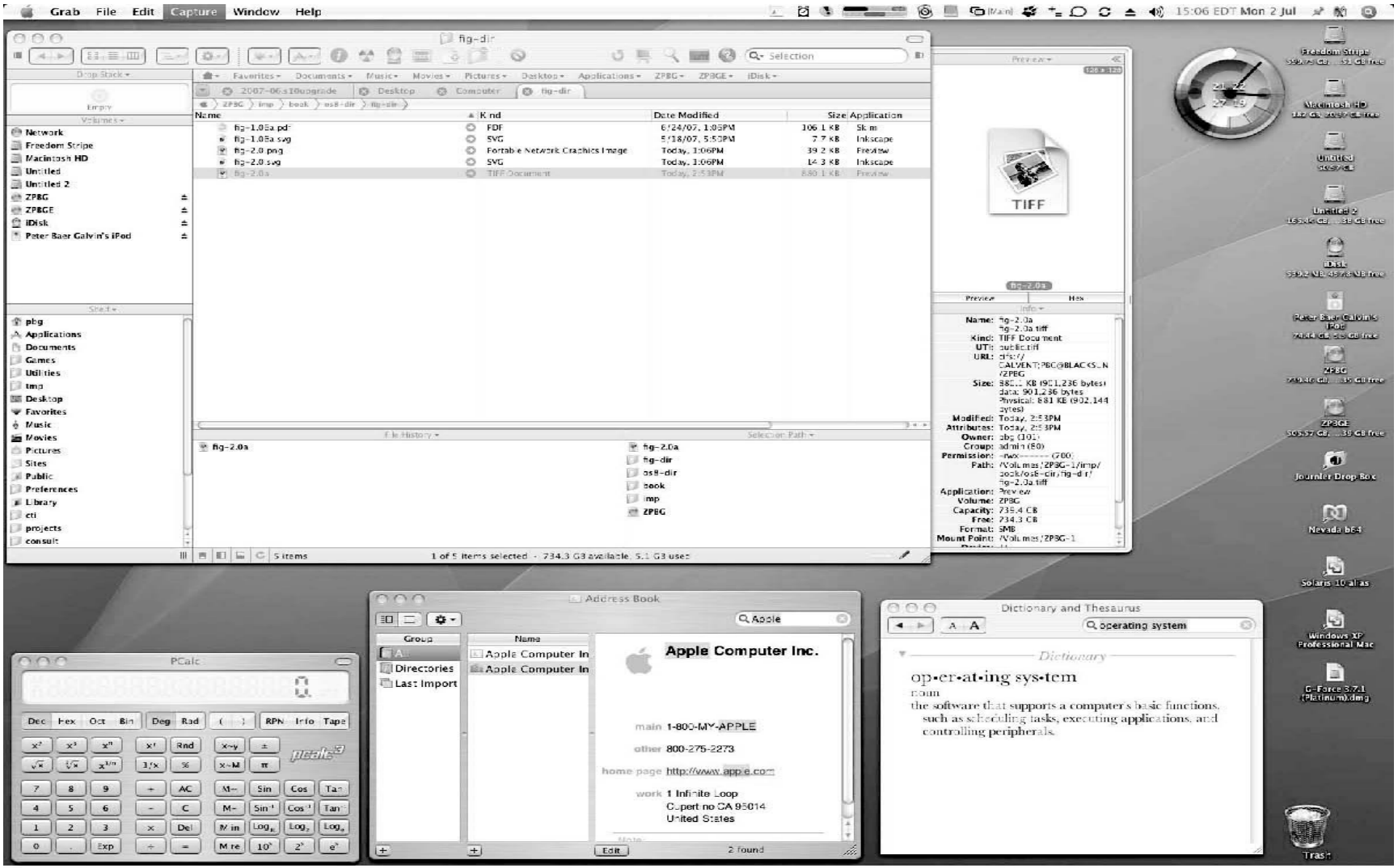
User Operating System Interface - GUI

- 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)

Bourne Shell Command Interpreter



The Mac OS X GUI



Outline

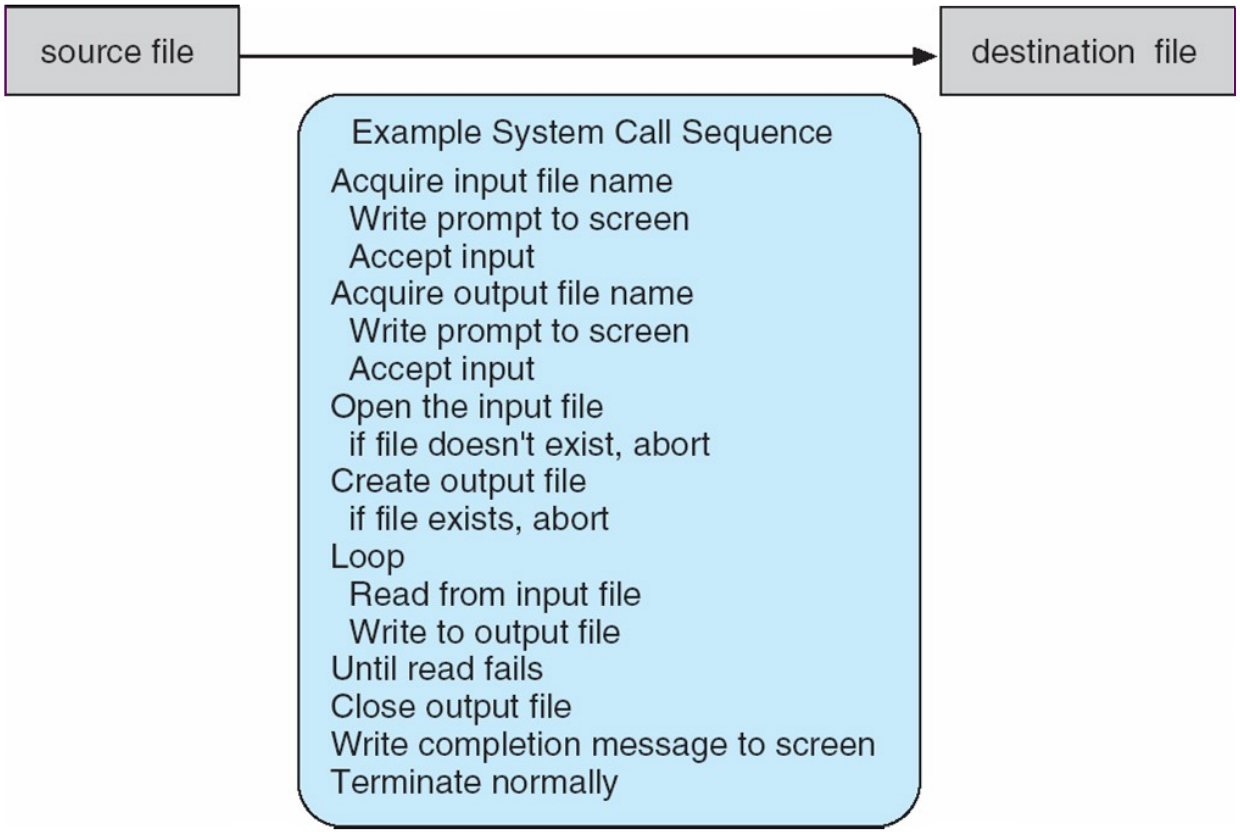
- Operating System Services
- User Operating System Interface
- **System Calls**
- **Types of System Calls**
- System Programs
- Operating System Design and Implementation
- Operating System Structure
- Virtual Machines
- Operating System Debugging
- Operating System Generation
- System Boot

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)

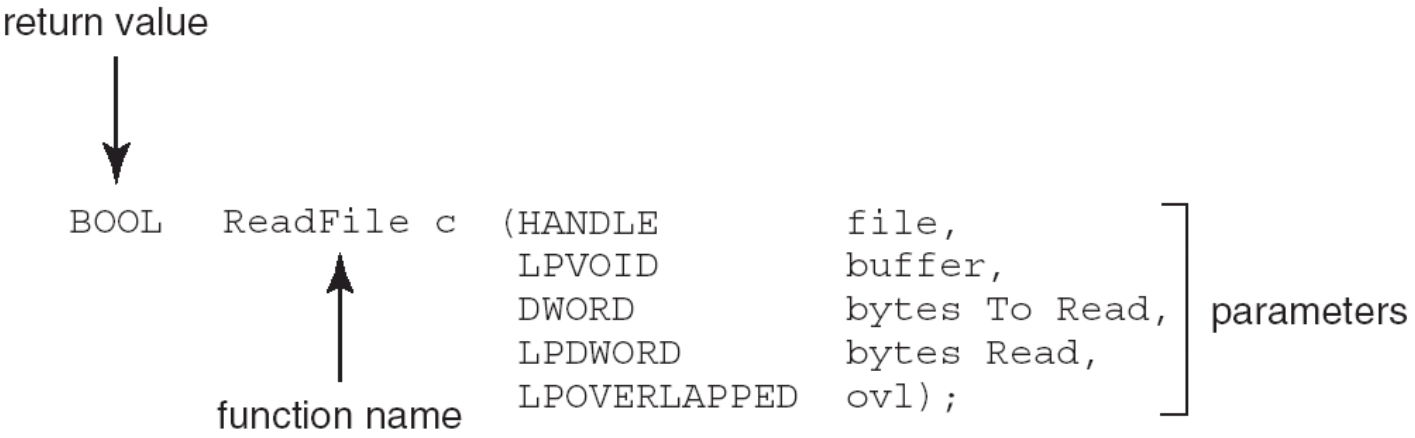
Example of System Calls

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



Example of Standard API

- Consider the ReadFile() function in the
- Win32 API—a function for reading from a file

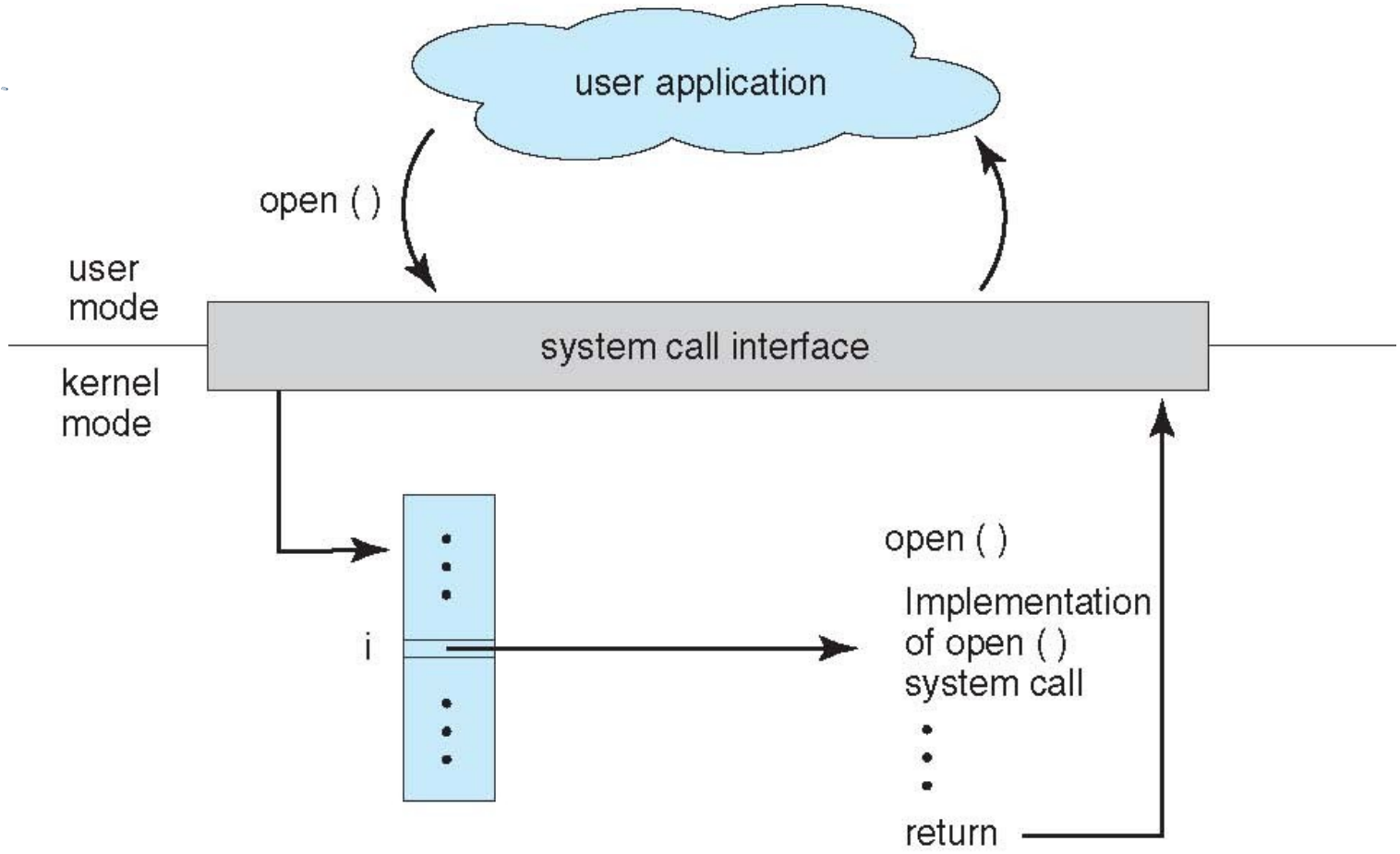


- A description of the parameters passed to ReadFile()
  - HANDLE file—the file to be read
  - LPVOID buffer—a buffer where the data will be read into and written from
  - DWORD bytesToRead—the number of bytes to be read into the buffer
  - LPDWORD bytesRead—the number of bytes read during the last read
  - LPOVERLAPPED ovl—indicates if overlapped I/O is being used

System Call Implementation

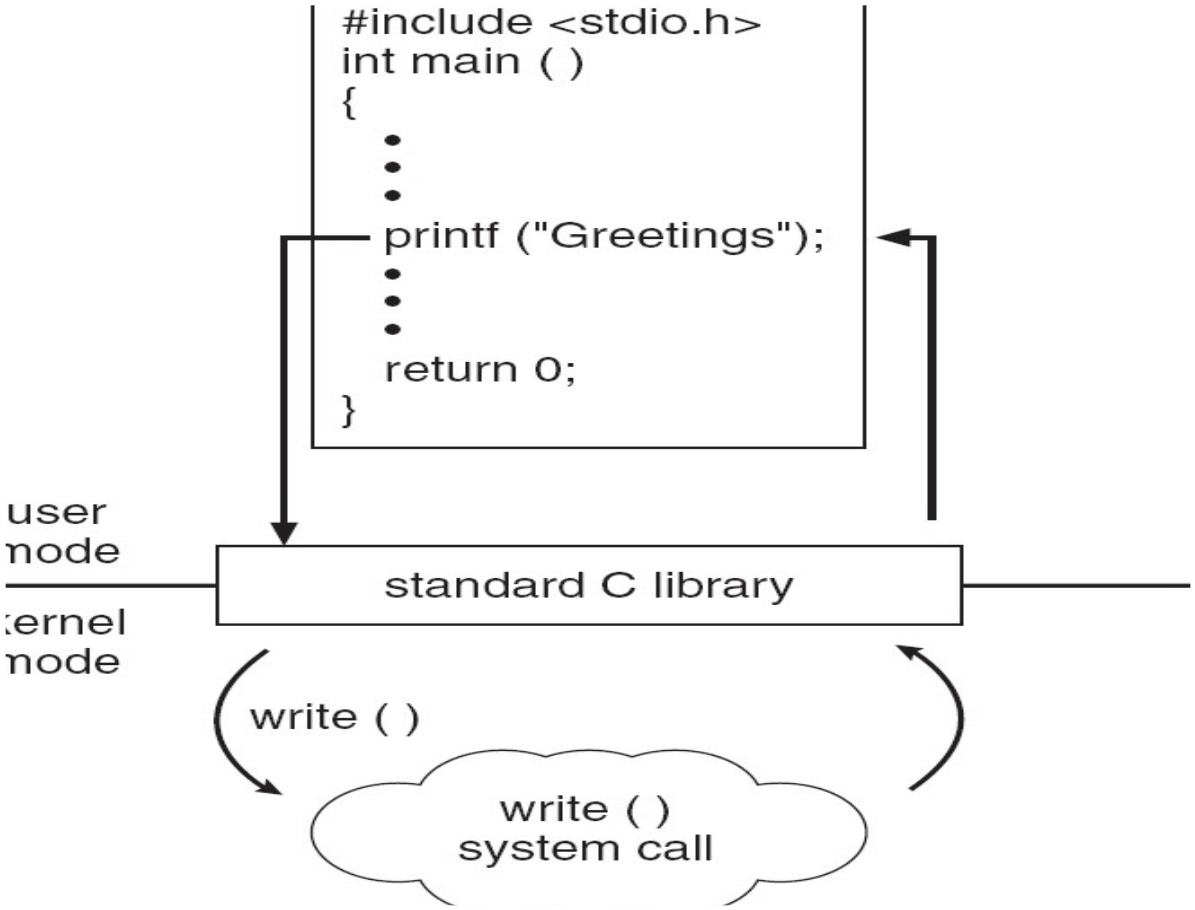
- Typically, a number associated with each system call
  - System-call interface maintains a table indexed according to these numbers
- The system call interface invokes intended system call in OS kernel and returns status of the system call and any return values
- The caller need know nothing about how the system call is implemented
  - Just needs to obey API and understand what OS will do as a result call
  - Most details of OS interface hidden from programmer by API
    - ✔ Managed by run-time support library (set of functions built into libraries included with compiler)

API – System Call – OS Relationship



Standard C Library Example

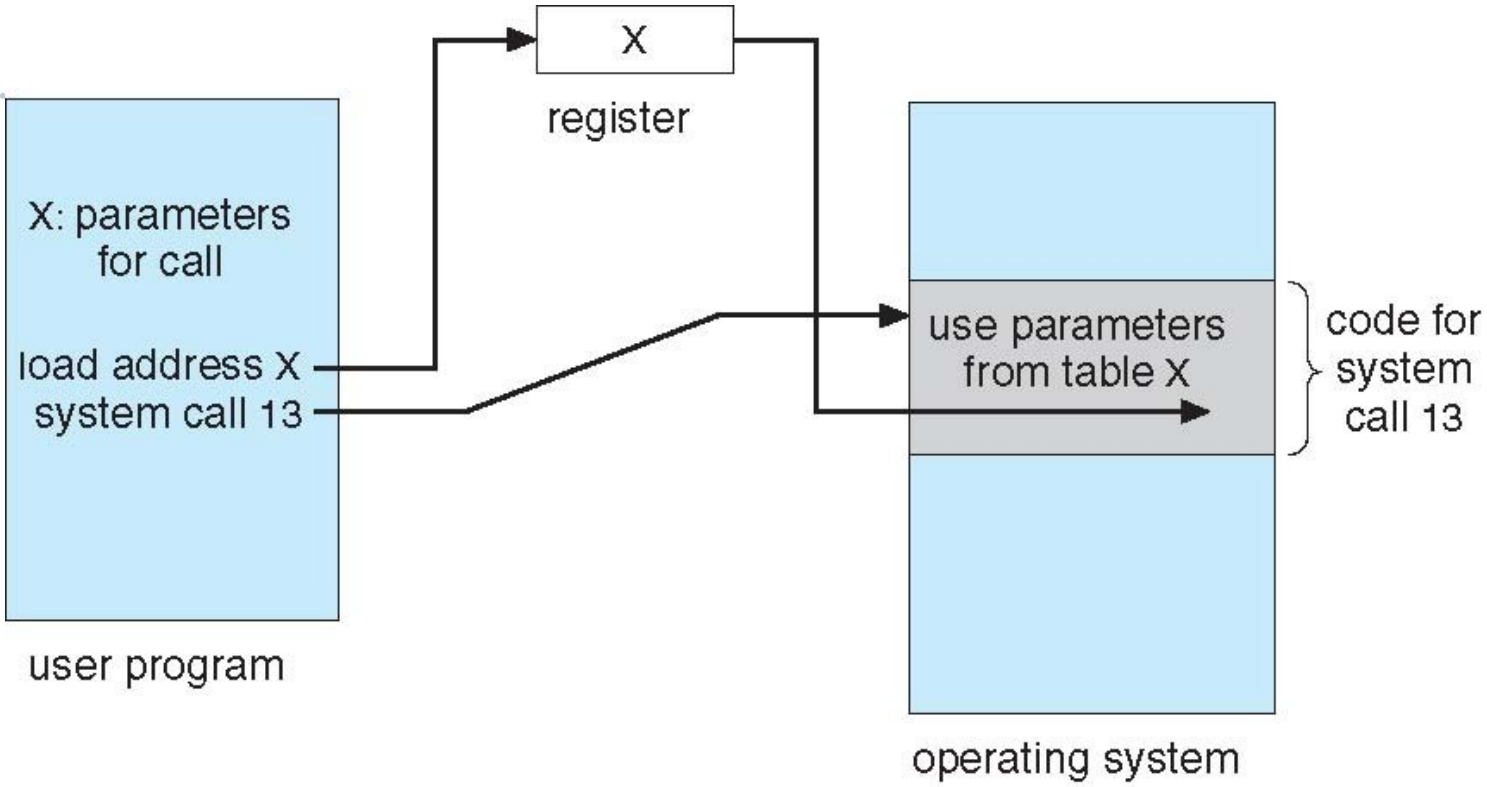
- C program invoking printf() library call, which calls write() system call



System Call Parameter Passing

- Often, more information is required than simply identity of desired system call
  - Exact type and amount of information vary according to OS and call
- Three general methods used to pass parameters to the OS
  - Simplest: pass the parameters in *registers*
    - ✔ In some cases, may be more parameters than registers
  - Parameters stored in a *block,* or table, in memory, and address of block passed as a parameter in a register
    - ✔ This approach taken by Linux and Solaris
  - Parameters placed, or *pushed,* onto the *stack* by the program and *popped* off the stack by the operating system
  - Block and stack methods do not limit the number or length of parameters being passed

Parameter Passing via Table



Types of System Calls

- Process control
  - end, abort
  - load, execute
  - create process, terminate process
  - get process attributes, set process attributes
  - wait for time
  - wait event, signal event
  - allocate and free memory
- File management
  - create file, delete file
  - open, close file
  - read, write, reposition
  - get and set file attributes

Types of System Calls (Cont.)

- Device management
  - request device, release device
  - read, write, reposition
  - get device attributes, set device attributes
  - logically attach or detach devices
- Information maintenance
  - get time or date, set time or date
  - get system data, set system data
  - get and set process, file, or device attributes
- Communications
  - create, delete communication connection
  - send, receive messages
  - transfer status information
  - attach and detach remote devices

Examples of Windows and Unix System Calls

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

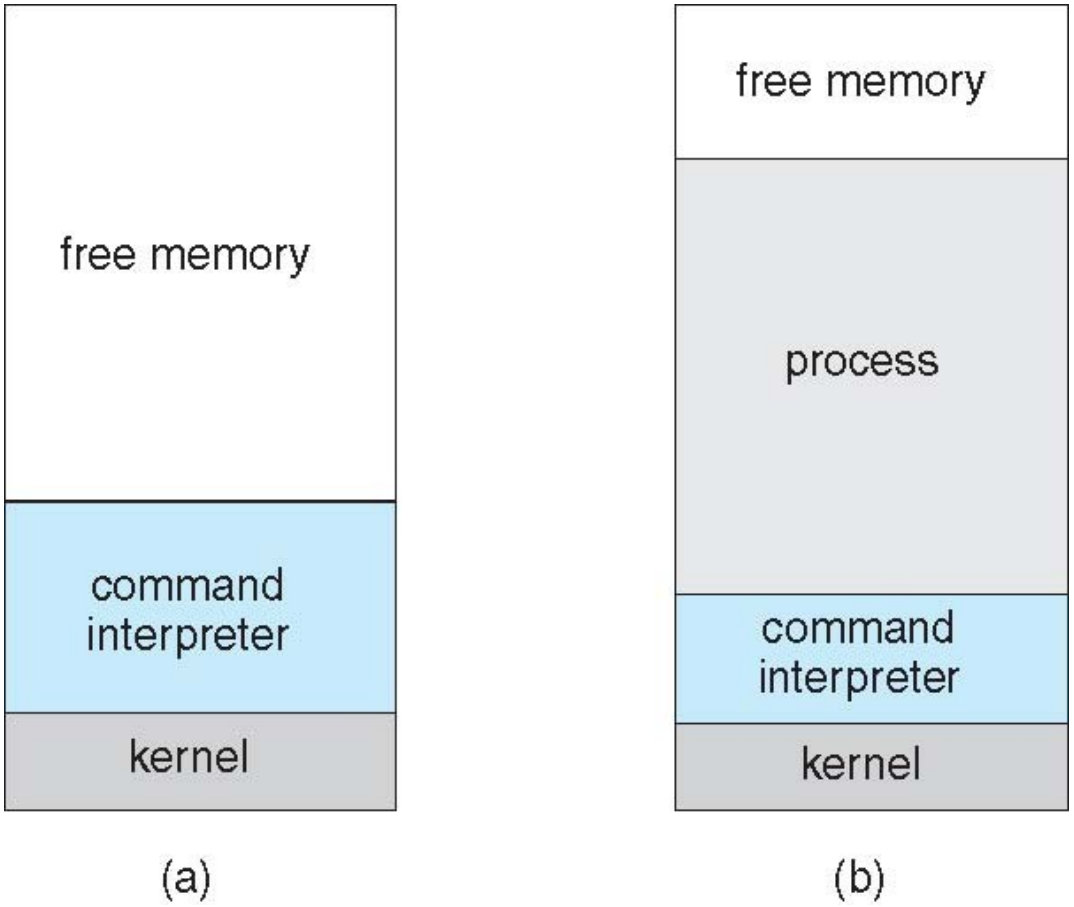
About System calls

- **The system calls are the instruction set of the OS virtual processor.**

Types of System Calls: Process Control Example: MS-DOS

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

MS-DOS execution

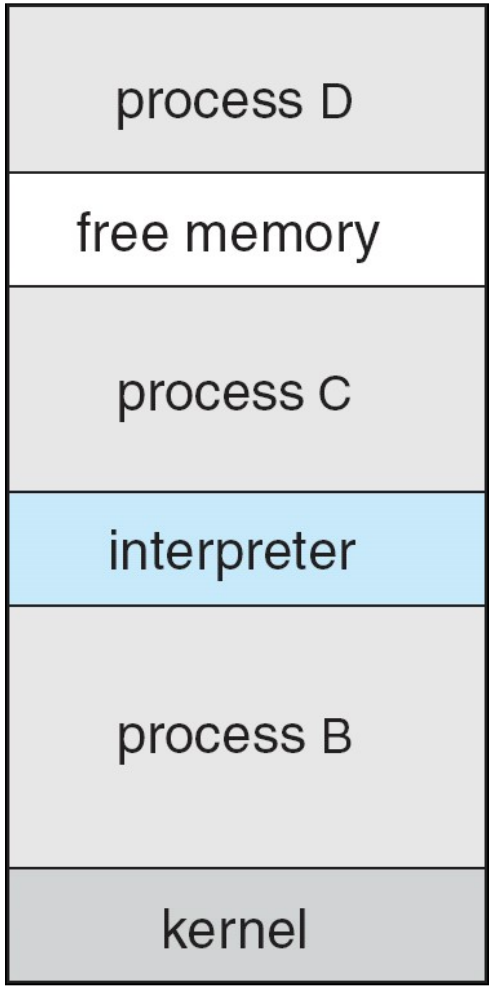


(a) At system startup (b) running a program

Process Control
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



Outline

- Operating System Services
- User Operating System Interface
- System Calls
- Types of System Calls
- **System Programs**
- Operating System Design and Implementation
- Operating System Structure
- Virtual Machines
- Operating System Debugging
- Operating System Generation
- System Boot

System Programs

- System programs provide convenient environment for program execution and development.
- Some are user interfaces to system calls; some are more complex.
- Each system call is usually supported by a system program.
  - File/directory manipulation: create, delete, copy name, print, dump, list
  - Status information: Programs that ask system information and display to users.
  - File modification: Text editor programs
  - Command interpreter
  - Programming language support
    - ✔ Compilers, assemblers, debuggers
  - Loaders and linkers
    - ✔ For enabling the execution of programs that start new processes and connect different ones.
  - Application programs
    - ✔ Text processing software, graphics support, spread sheets, games

System Programs

- System programs provide a convenient environment for program development and execution. They 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

Outline

- Operating System Services
- User Operating System Interface
- System Calls
- Types of System Calls
- System Programs
- **Operating System Design and Implementation**
- Operating System Structure
- Virtual Machines
- Operating System Debugging
- Operating System Generation
- System Boot

System Design and Implementation :design goals

- We discuss problems we face while designing and implementing a system
- First problem is defining the goals
  - Batch, time-shared, single-user, multi-user, distributed, real-time, or general purpose.
- The goals can be divided into two parts: user goals and system goals
  - Users: Convenient to use, easy to learn, easy to use, reliable, safe, and fast.
  - System goals by designers: easy to design, create, maintain, and operate. It should be flexible, reliable, and error-free.
  - Ex: MSDOS is single user OS; MVS (multiple virtual storage) is a multiuser, multi access OS for IBM mainframes.
- Software engineering principles can be applied for deciding goals.

Mechanisms and Policies

- The specification and design of OS is highly creative task.
- One important principle is separation of policy and mechanism.
- Mechanisms do not change often; Policies may change
- Mechanisms describe **how** things are done
  - For CPU protection, timer will be used.
  - Does not include decisions as to which process gets priority
- Policies describe **what** will be done
  - Deciding whether processes with more I/O get priority over CPU bound processes is an example
  - This policy can be a input parameter and can change from scheduler to scheduler
  - For CPU protection, how long the timer is set is a policy decision.
- A mechanism which is insensitive to change in policy is preferred.
- **Micro-kernel OSs** only implement mechanisms. Policies are provided at user level

Implementation

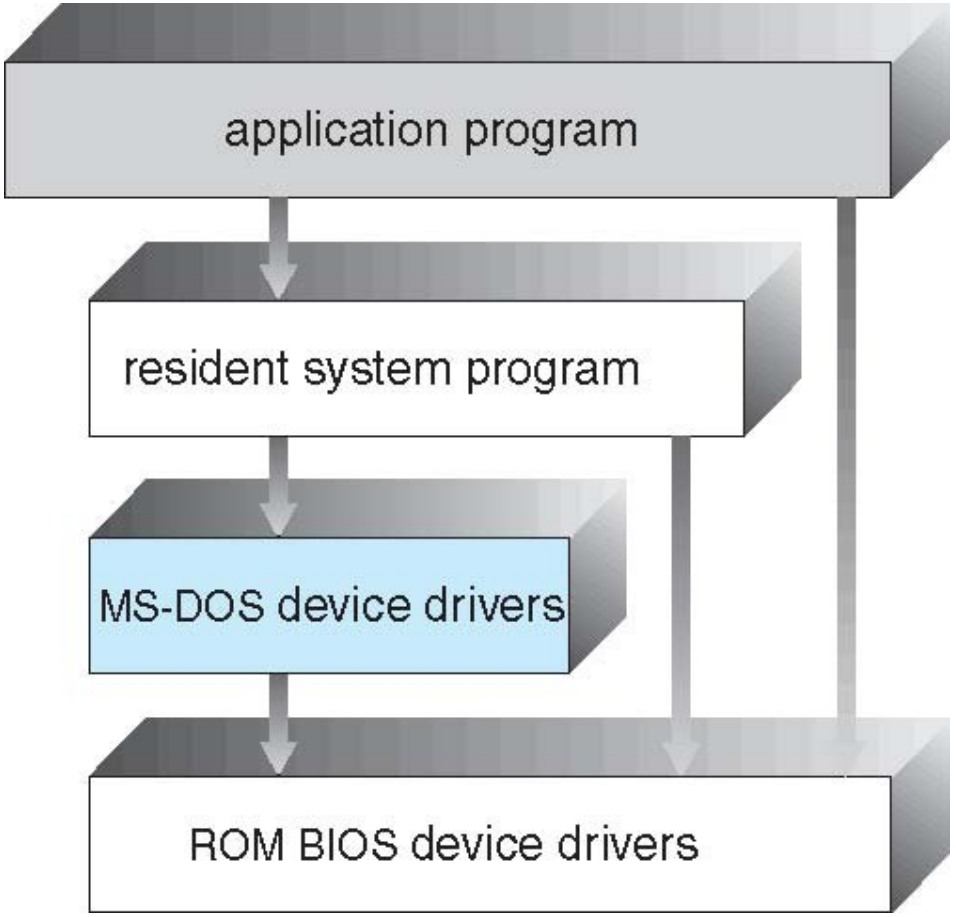
- Assembly language can be used.
  - High speed
- High level language
  - Easy to develop, easy to understand and debug, easy to port.
  - UNIX was written in C.
  - Dis adv: reduced speed and increased storage requirements.
  - We have sophisticated compilers.
- After writing in high-level language, bottleneck portion can be replaced by assembly language.

Outline

- Operating System Services
- User Operating System Interface
- System Calls
- Types of System Calls
- System Programs
- Operating System Design and Implementation
- **Operating System Structure**
- Virtual Machines
- Operating System Debugging
- Operating System Generation
- System Boot

Simple Structure

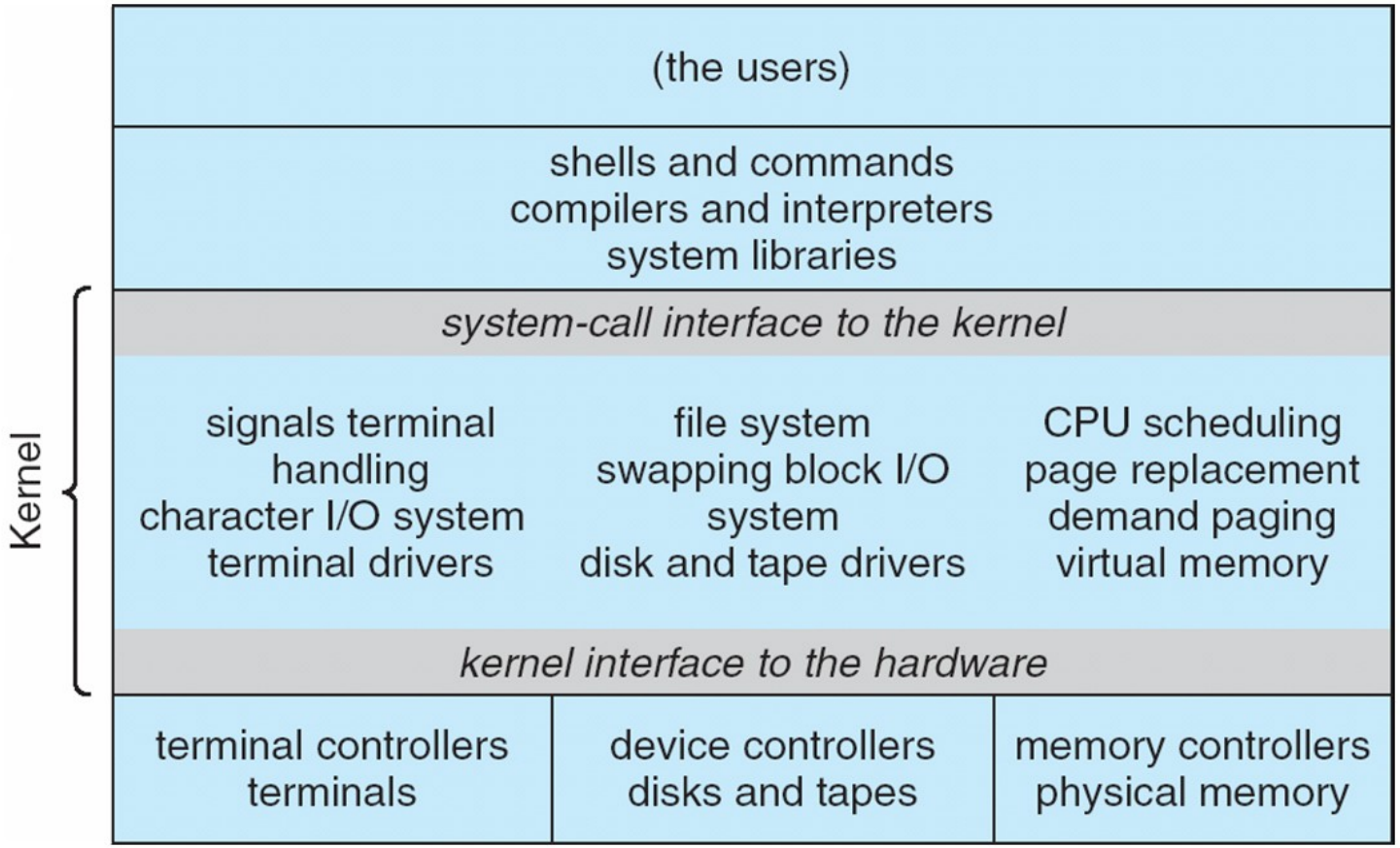
- MS-DOS – written to provide the most functionality in the least space
  - Not divided into modules
  - Although MS-DOS has some structure, its interfaces and levels of functionality are not well separated



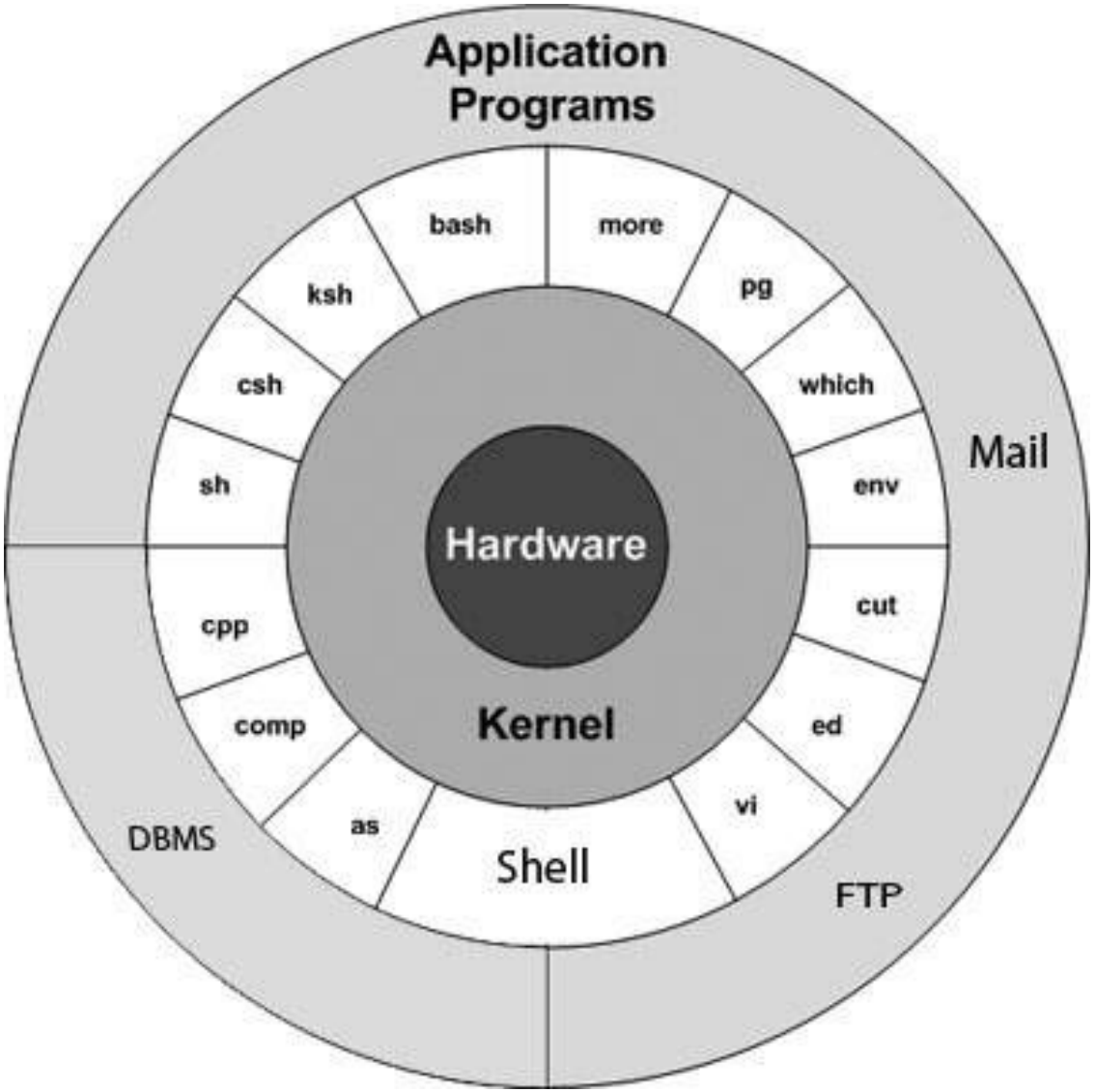
UNIX

- UNIX – limited by hardware functionality, the original UNIX operating system had limited structuring. The UNIX OS consists of two separable parts
  - Systems programs
  - The kernel
    - Consists of everything below the system-call interface and above the physical hardware
    - Provides the file system, CPU scheduling, memory management, and other operating-system functions; a large number of functions for one level

Traditional UNIX System Structure



Traditional UNIX System Structure



Structure view: Structure of OS designs

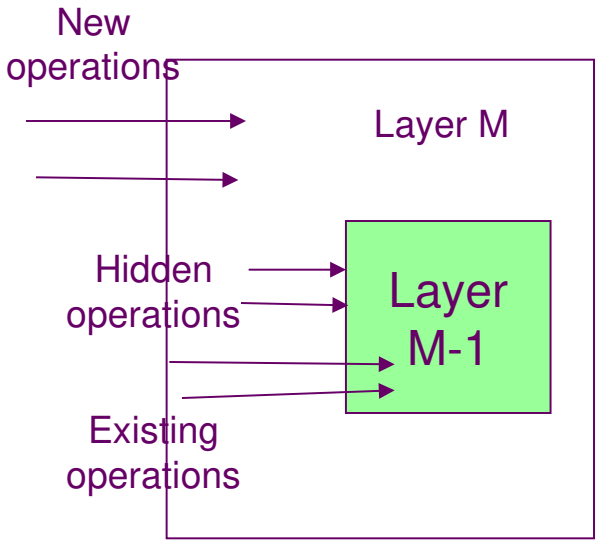
- Traditionally adhoc and unstructured approaches.
- Structured designs
  - Layered approach is followed to realize modularity
  - Encapsulate the modules separately
  - Have a hierarchy of layers where each layer calls procedures from the layers below.

- The THE OS (Dijkstra)
  - Layer 0 : Hardware
  - Layer 1: CPU scheduler
  - Layer 2: Memory manager
  - Layer 3: Operator-console device driver
  - Layer 4: I/O buffering
  - Layer 5: user program layer

- The VENUS OS (Liskov)
  - Layer 0 : Hardware
  - Layer 1: Instruction interpreter
  - Layer 2: CPU scheduler
  - Layer 3: I/O channel
  - Layer 4: Virtual memory
  - Layer 5: device drivers and schedulers
  - Layer 6: user program layer

Structure view: layered system design

- A general philosophy that builds on the above approach
  - Decompose functionality into layers such that
    - ✔ Hardware is level 0, and layer t accesses functionality at layer (t-1) or less
    - ✔ Access via appropriately defined system calls.
- Advantages
  - Modular design: well defined interfaces between layers
  - Prototyping/development
    - ✔ Association between function and layer eases overall OS design.
    - ✔ OS development and debugging is layer by layer.
    - ✔ Simplifies debugging and system verification

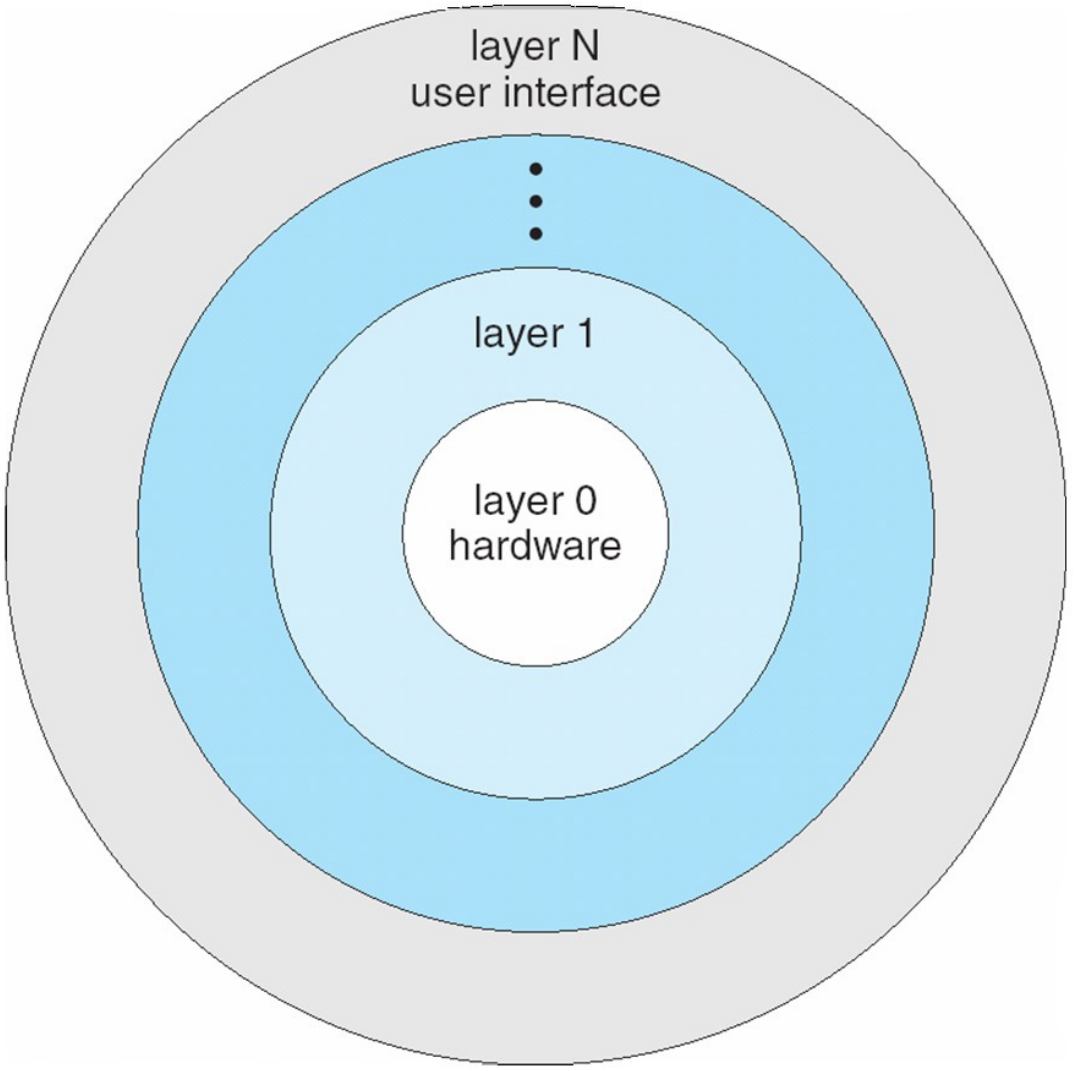


Layered Approach

- The operating system is divided into a number of layers (levels), each built on top of lower layers. The bottom layer (layer 0), is the hardware; the highest (layer N) is the user interface.

- With modularity, layers are selected such that each uses functions (operations) and services of only lower-level layers

Layered Operating System



OS design hierarchy: hypothetical model

Level	Name	Objects	Example operations
15	Shell	User programming environment	Statements in shell language
14	Directories (maintains association between external and internal identifiers of system resources and objects)	Directories	Create, destroy, search, list, access rights
13	User Processes	User process	Fork, quit, kill, suspend, resume
12	Stream I/O	streams	open., close
11	Devices (access to external devices)	Printers, displays, and key boards	Create, destroy, open, close, read, write
10	File system (long storage of named files)	files	Create, destroy, open, close
9	Communications	pipes	Create, destroy, open, close, read, write
8	Capabilities	Capabilities	Create, Validate, Attenuate
7	Virtual memory (creating logical address space for programs)	Segments, pages	Read, write, fetch
6	Local secondary storage (position of read/write heads)	Blocks of data, device channels	Read, write, allocate, free
5	Primitive processes	Primitive process, semaphores, synchronization premitives	Suspend, resume, wait, signal
4	Interrupts	Interrupts handling programs	Invoke, mask, unmask, retry
3	Procedures	Procedures, call stack	Mark stack, call, return
2	Instruction set	Evaluation, stack, micro-program, interpreter	Load, store, add, subtract, branch

Design hierarchy: hypothetical model

- Part of hardware
  - Layer 1: Operations are actions on logic gates
  - Layer 2: Operations are machine language instructions
  - Layer 3: Call and return operations of procedures
  - Layer 4: Interrupts Operations
- Part of Core Operation system
  - Layer 5: Operations to support multiple processes include ability to suspend and revoke process. Synchronization
  - Layer 6: Operations on secondary storage devices (basic read, write and free blocks of data). Uses operations of layer 5.
  - Layer 7: Create logical address space (virtual memory): Organizes virtual memory into blocks.
  - Layer 8: Capabilities:
- External to OSs
  - Layer 9: Deals with communication of information and messages between processes. Deals with richer sharing of information. Notion of "pipe" is provided which is defined as with its output from one process and its input into another process.

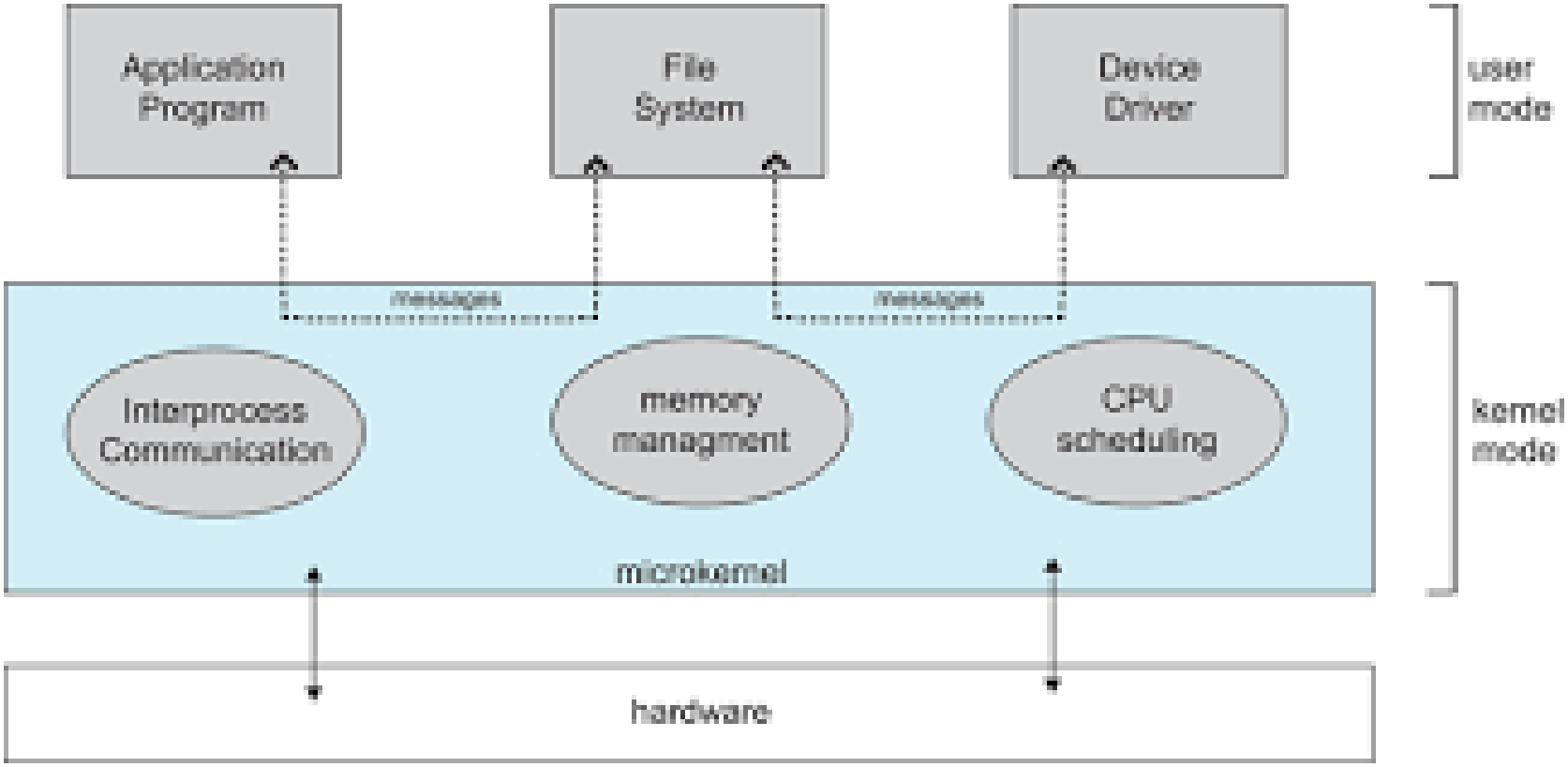
Design hierarchy: hypothetical model

- External to OSs
  - Layer 10: Supports long term storage of named files. Data on secondary storage are viewed in terms of abstract, variable length entities. (Different from layer 6 which views storage as tracks, sectors, and fixed size blocks)
  - Layer 11: Provides access to external devices using standard interfaces
  - Layer 12: Stream I/O.
  - Layer 13: Operations on Directories: Maintains the association between the external and internal identifiers of the system's resources and objects. External identifier is used by user. Internal identifier is used by lower levels of operating system to locate and control the object.
  - Layer 14: Provides full feature facility to support processes. Virtual address space, list of objects processes may interact other characteristics of processes the OS wants to use.
  - Layer 15: Provides Shell: which accepts user commands or job control statements, interprets these and starts processes as needed.

Microkernel System Structure

- Moves as much from the kernel into "*user*" space
- Communication takes place between user modules using message passing
- Benefits:
  - Easier to extend a microkernel
  - Easier to port the operating system to new architectures
  - More reliable (less code is running in kernel mode)
  - More secure
- Detriments:
  - Performance overhead of user space to kernel space communication through message passing.

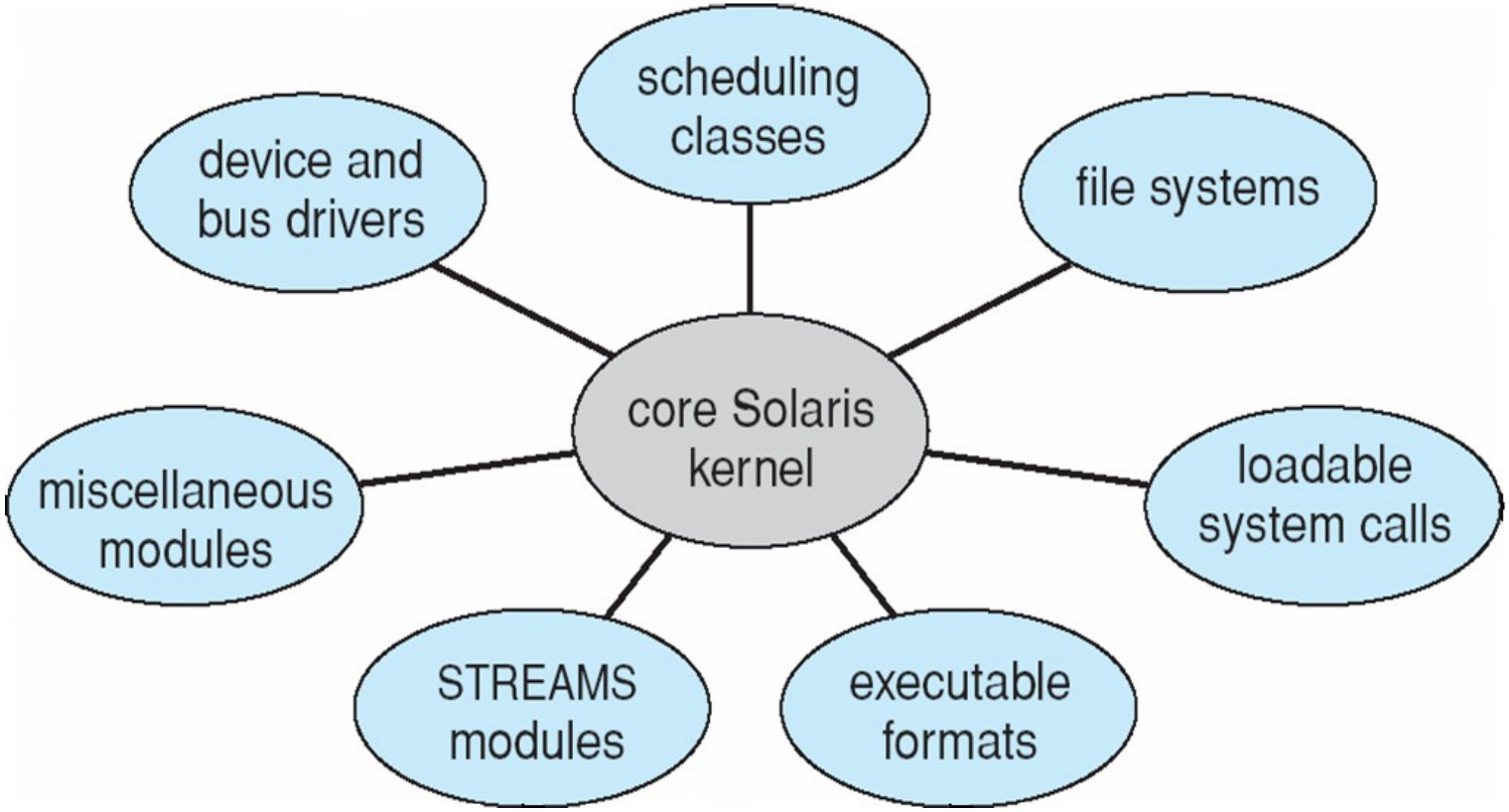
Architecture of a typical microkernel



Modules

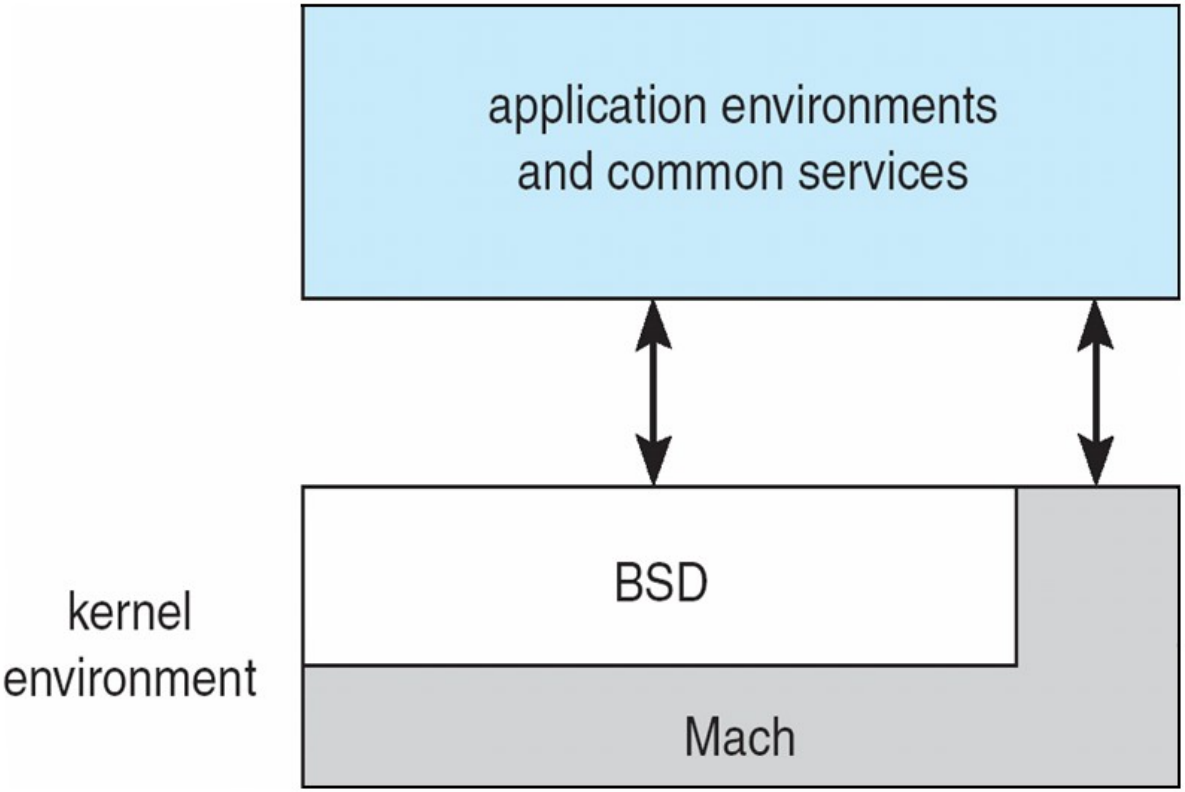
- Most modern operating systems implement kernel modules
  - Uses object-oriented approach
  - Each core component is separate
  - Each talks to the others over known interfaces
  - Loadable kernel modules
    - ✔ Each is loadable as needed within the kernel
- Overall, similar to layers but with more flexible

Solaris Modular Approach



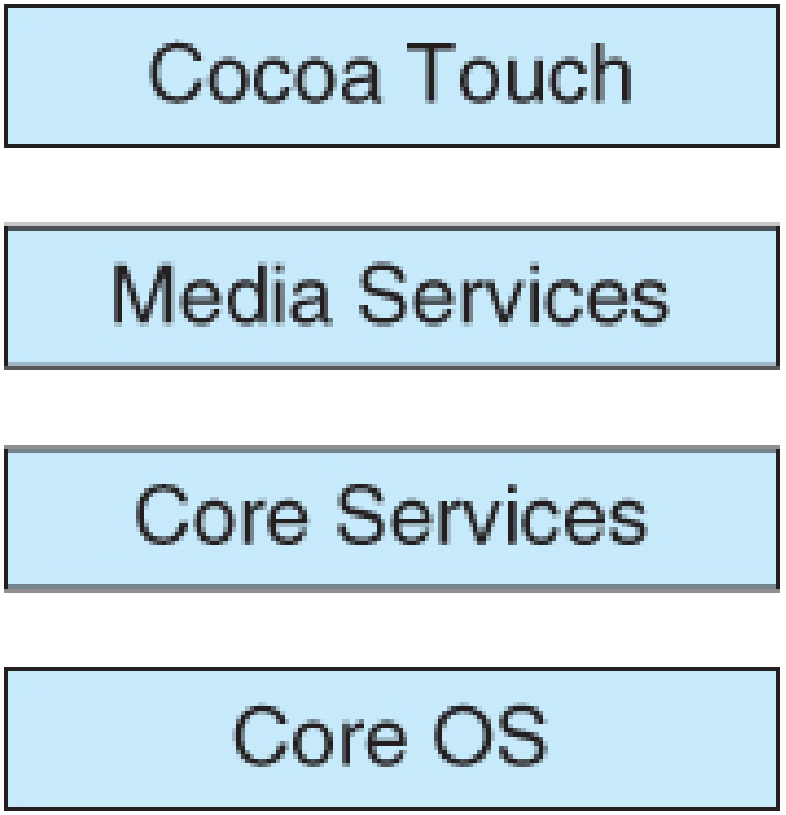
- Primary module is a core module and communicates with other modules.
- Modules need not invoke message passing for communication
- Support different file systems with loadable modules.

Mac OS X Structure: Hybrid Structure



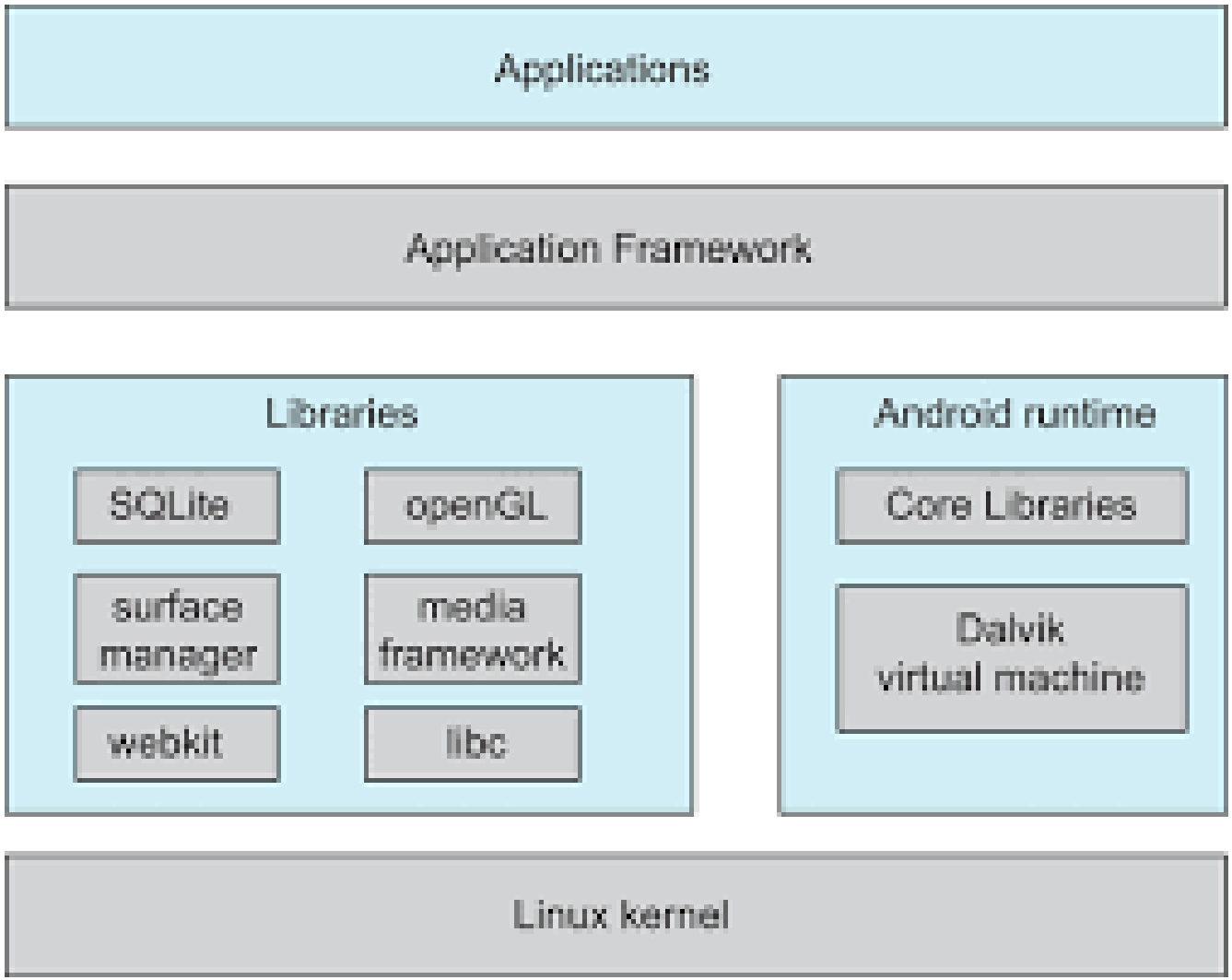
* BSD: Berkeley Software distribution of UNIX OS

Architecture of Apple's IOS



* iOS: iPhone OS

Architecture of Android



Outline

- Operating System Services
- User Operating System Interface
- System Calls
- Types of System Calls
- System Programs
- Operating System Design and Implementation
- Operating System Structure
- **Virtual Machines**
- Operating System Debugging
- Operating System Generation
- System Boot

Structure view: Virtual machines

- System programs above kernel can use either system calls or hardware instructions.
- System programs treat the hardware and the system calls as though they both are at the same level.
- In general application programs may view everything under them in the hierarchy as a part of machine itself, which leads to the concept of virtual machines..
- The virtual machine approach provides an interface that is identical to the underlying bare hardware.
- Each process is provided with a (virtual) copy of the underlying computer.
- The resources of the physical computer are shared to create the virtual machines.
  - CPU scheduling and spooling are used.

Structure view: Virtual machines

- Logical conclusion of layered approach.
  - The OS offers an abstract machine to execute on
    - ✔ Hides the specifics and details of hardware
  - Users are given their own VM, they can run any of the OS or software packages that are available on the underlying machine.
  - Provides a complete copy of underlying machine to the user
    - ✔ Pioneered under the name virtual machine (VM) by IBM
    - ✔ Became a household name with JAVA
- Mechanism of creating this illusion
  - CPU scheduling
    - ✔ Sharing the CPU in a transparent way
  - Memory management
    - ✔ Having large virtual memory space to address
  - Additional features such as file systems and so on are offered through file management subsystems
    - ✔ Problem is with partitioning the disk
      - – VM introduced minidisks such partitioning of tracks on the physical disk.

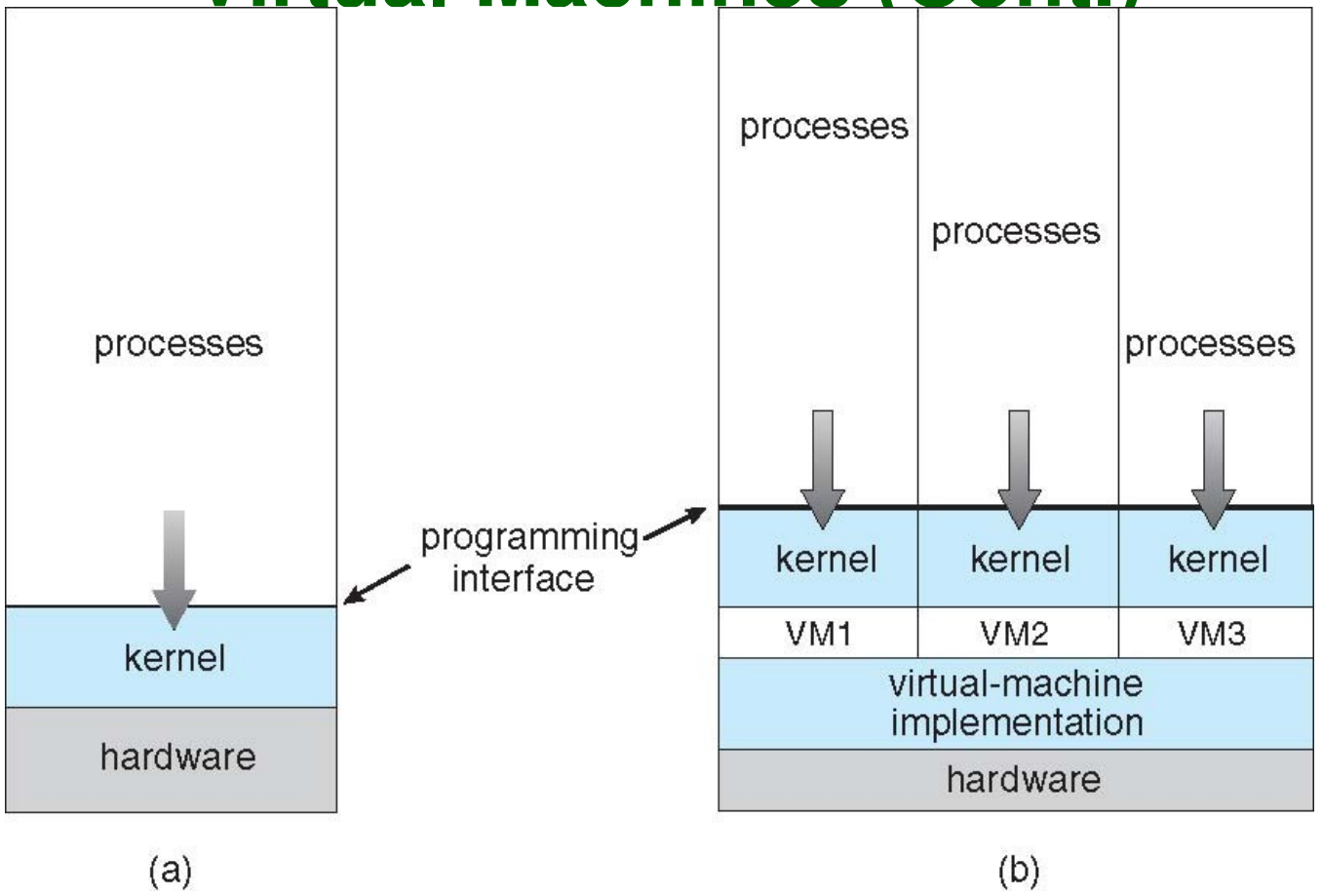
Virtual Machines

- 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) Nonvirtual machine (b) virtual machine

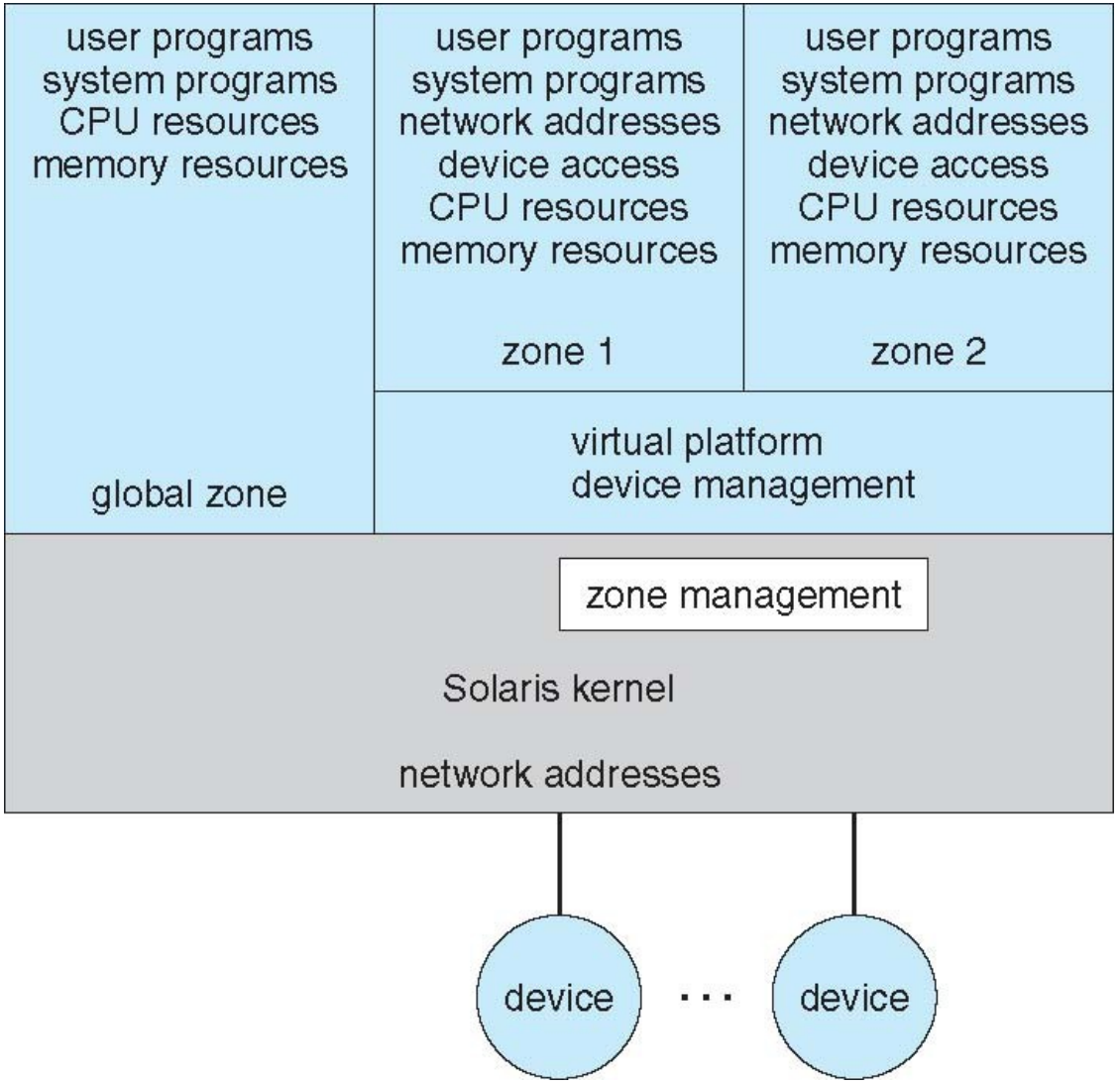
Para-virtualization

- Presents guest with system similar but not identical to hardware
- Guest must be modified to run on paravirtualized hardware
- Guest can be an OS, or in the case of Solaris 10 applications running in **containers**

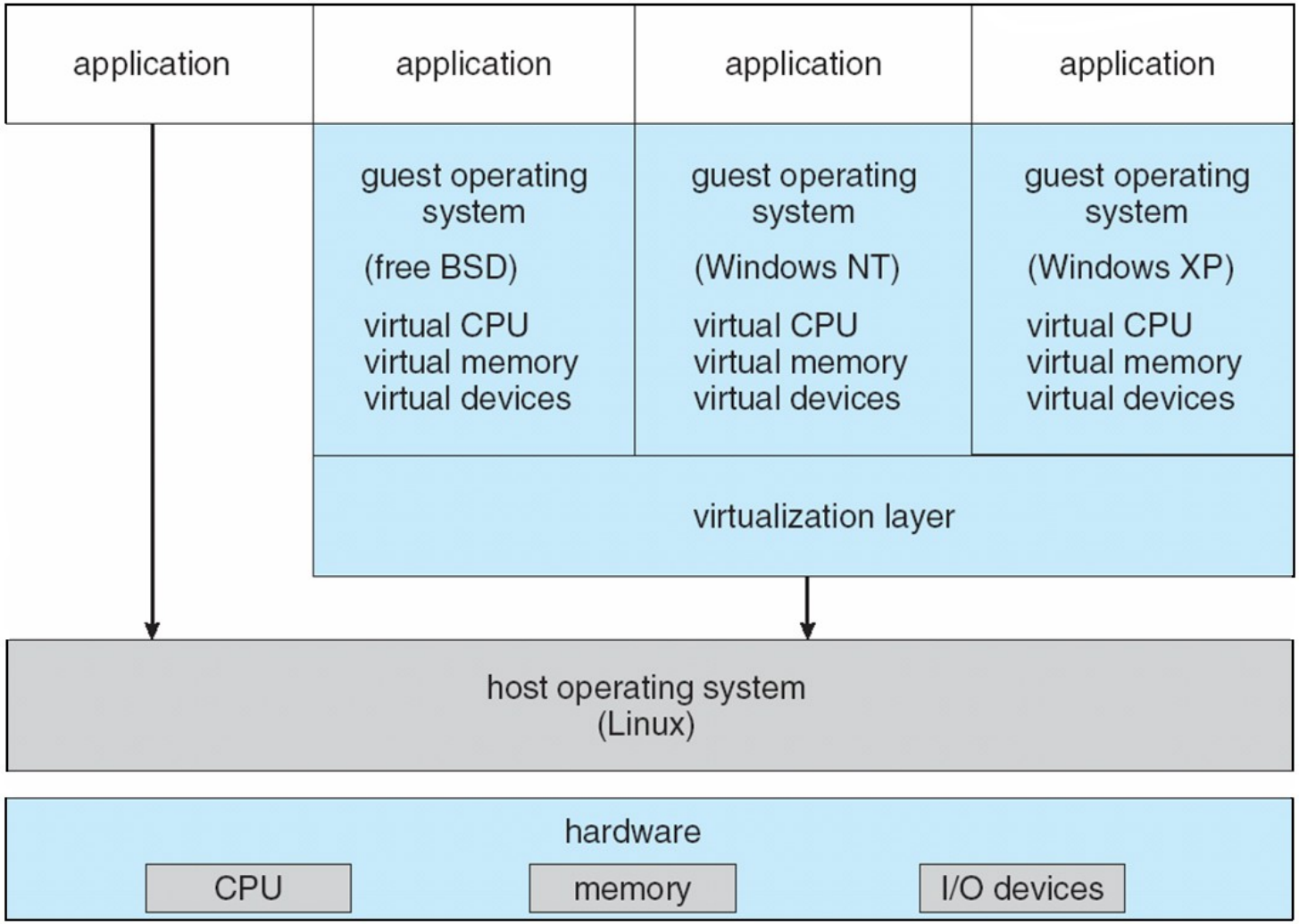
Virtualization Implementation

- Difficult to implement – must provide an *exact* duplicate of underlying machine
  - Typically runs in user mode, creates virtual user mode and virtual kernel mode
- Timing can be an issue – slower than real machine
- Hardware support needed
  - More support-> better virtualization
  - i.e. AMD provides “host” and “guest” modes

Solaris 10 with Two Containers



VMware Architecture



Structure view: Virtual machines (cont..)

- Operation
  - User-level code executes as is
  - Supervisor code executes at user level
    - ✔ Privileged instructions are simulated.
      - – Generate trap to VM emulator
    - ✔ Hidden registers and I/O instructions are simulated.
  - Virtual machine software has two modes
    - ✔ User mode and monitor mode
  - The user program will execute in two modes
    - ✔ Virtual user mode
    - ✔ Virtual monitor mode

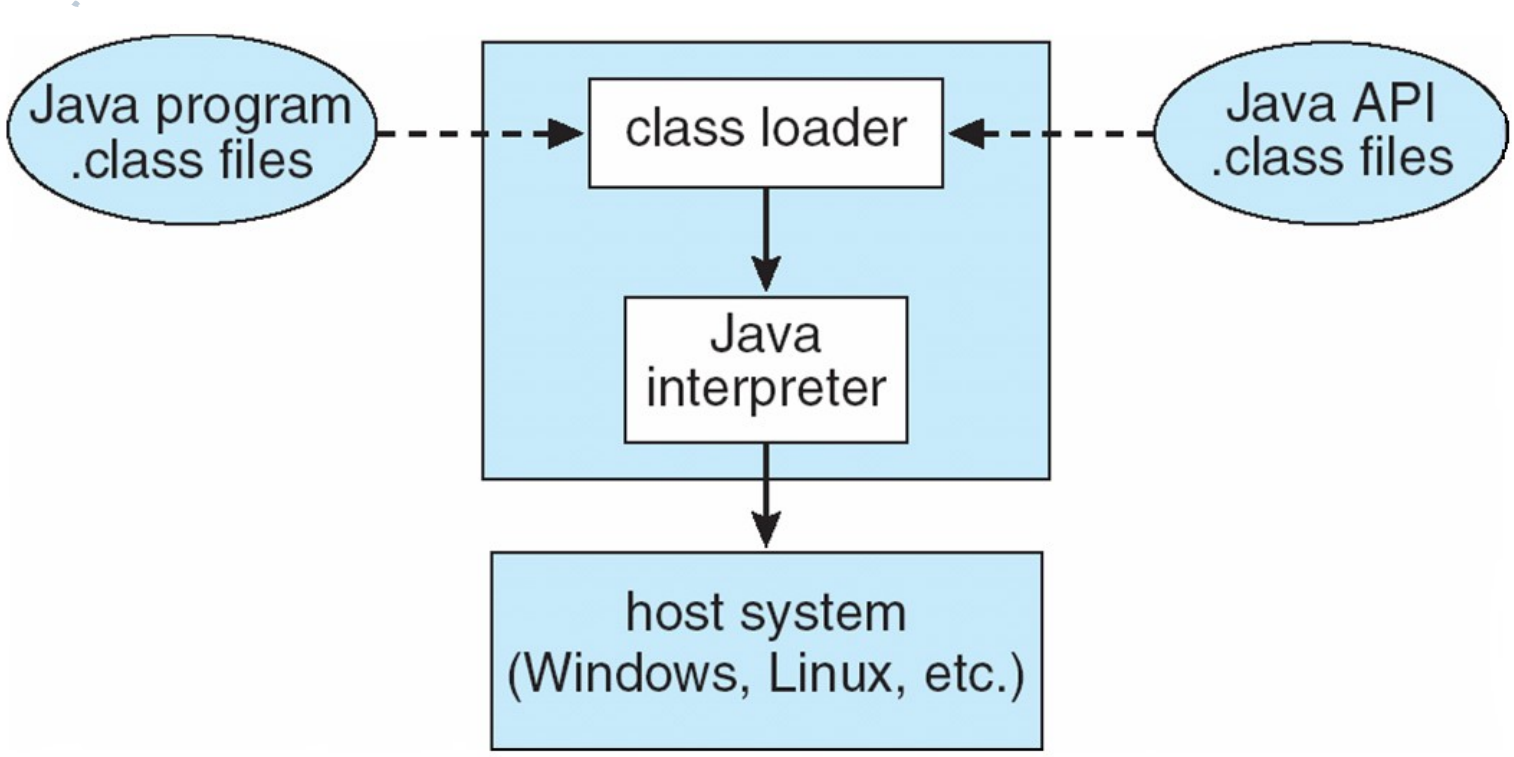
Structure view: Virtual machines (cont..)

- Advantages
  - A software created abstraction that is a replica of the underlying machine
  - Additional functionality can be provided
    - ✔ Can run different OSs on different VMs.
  - Each VM is completely isolated from others, so secure.
  - Protection of various system resources.
  - It can be used for OS research and development.
  - System programs are given their own virtual machine.
  - Thousands of programs are available for MSDOS on Intel CPU-based systems.
  - SUN micro-systems and DEC use faster machines
    - ✔ To test MSDOS programs on the faster machines, the best way is to provide a virtual Intel machine.
  - System compatibility problems can be solved
- Disadvantages
  - Performance degradation is inevitable

JAVA

- JAVA is designed by SUN Micro systems.
- JAVA compiler generates byte code output.
- These instructions run on Java virtual machine.
- JVM runs on many types of computer.
- JVM is implemented in web browsers.
- JVM implements a stack based instruction set.

The Java Virtual Machine



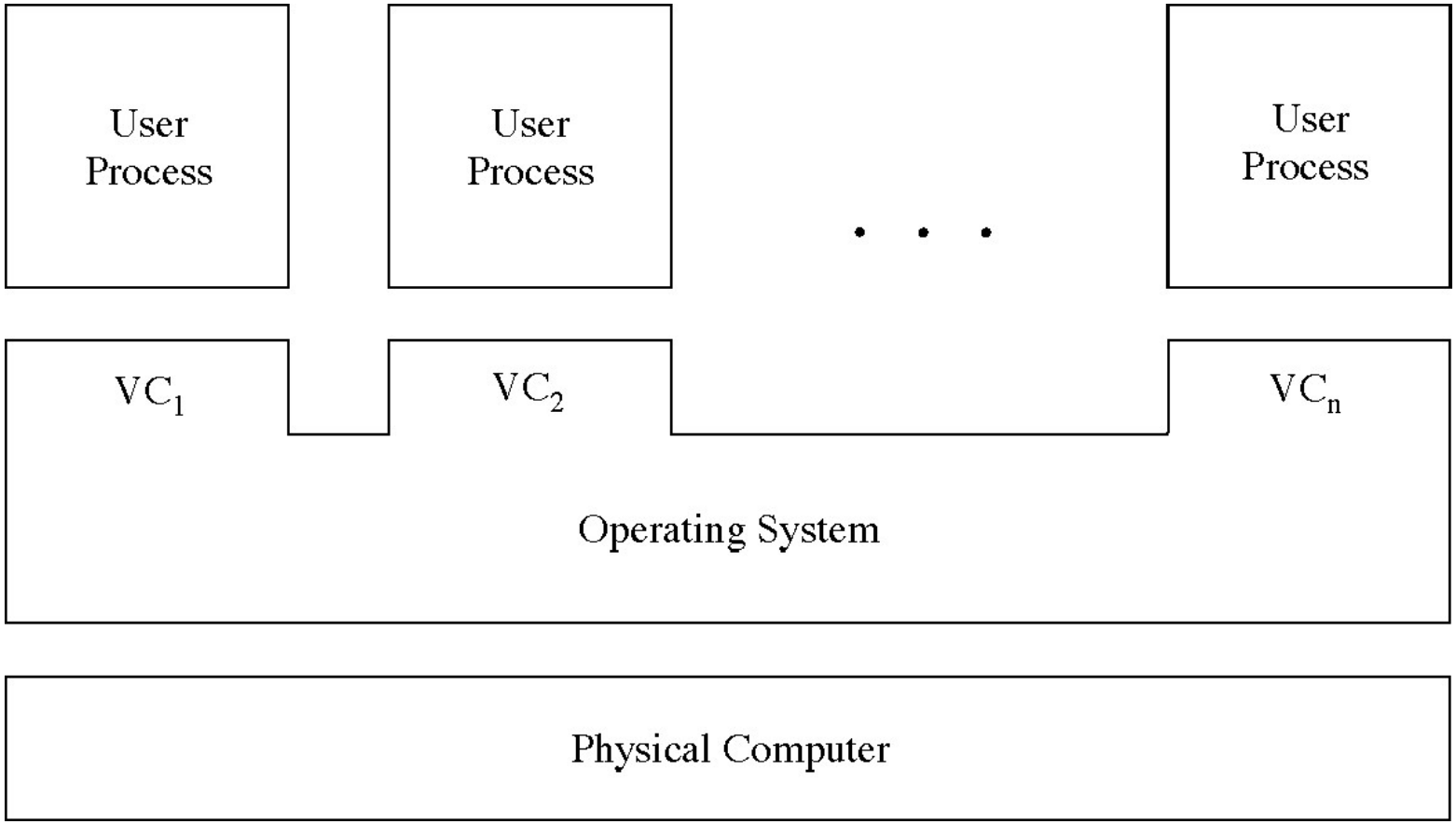
Types of multiplexing

- Time multiplexing
  - time-sharing
  - scheduling a serially-reusable resource among several users
- Space multiplexing
  - space-sharing
  - dividing a multiple-use resource up among several users

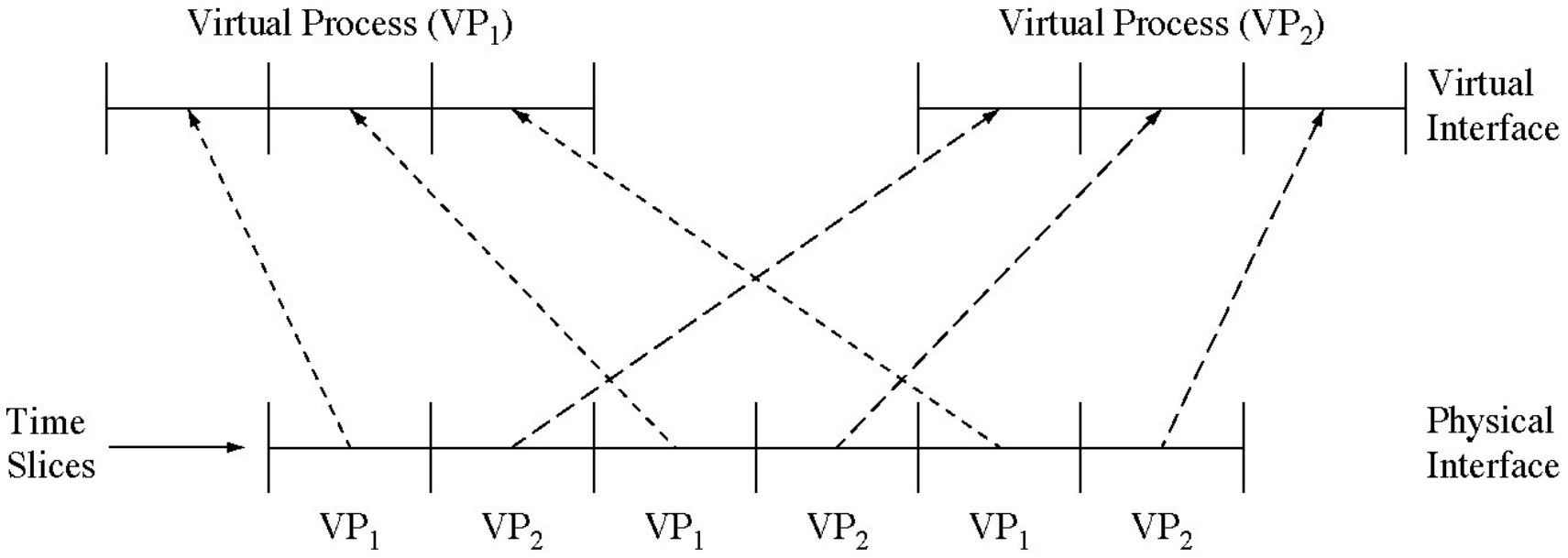
Virtual computers

- Processor virtualized to processes
  - mainly time-multiplexing
- Memory virtualized to address spaces
  - space and time multiplexing
- Disks virtualized to files
  - space-multiplexing
  - transforming

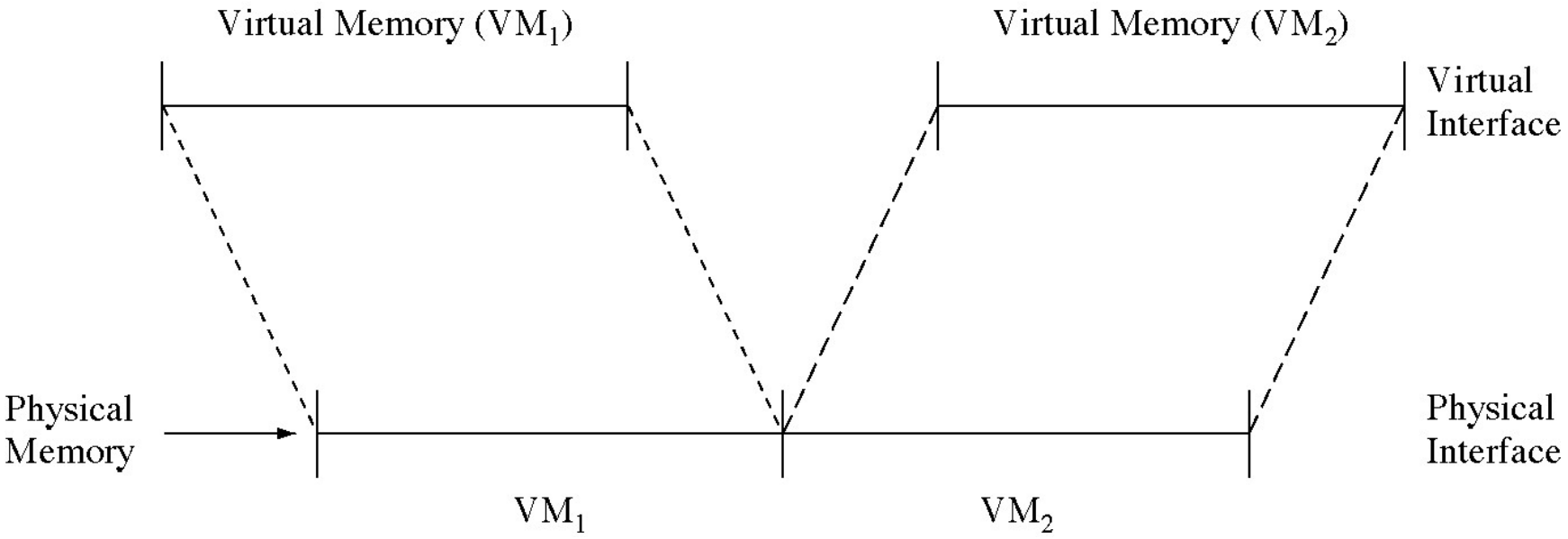
Multiple virtual computers



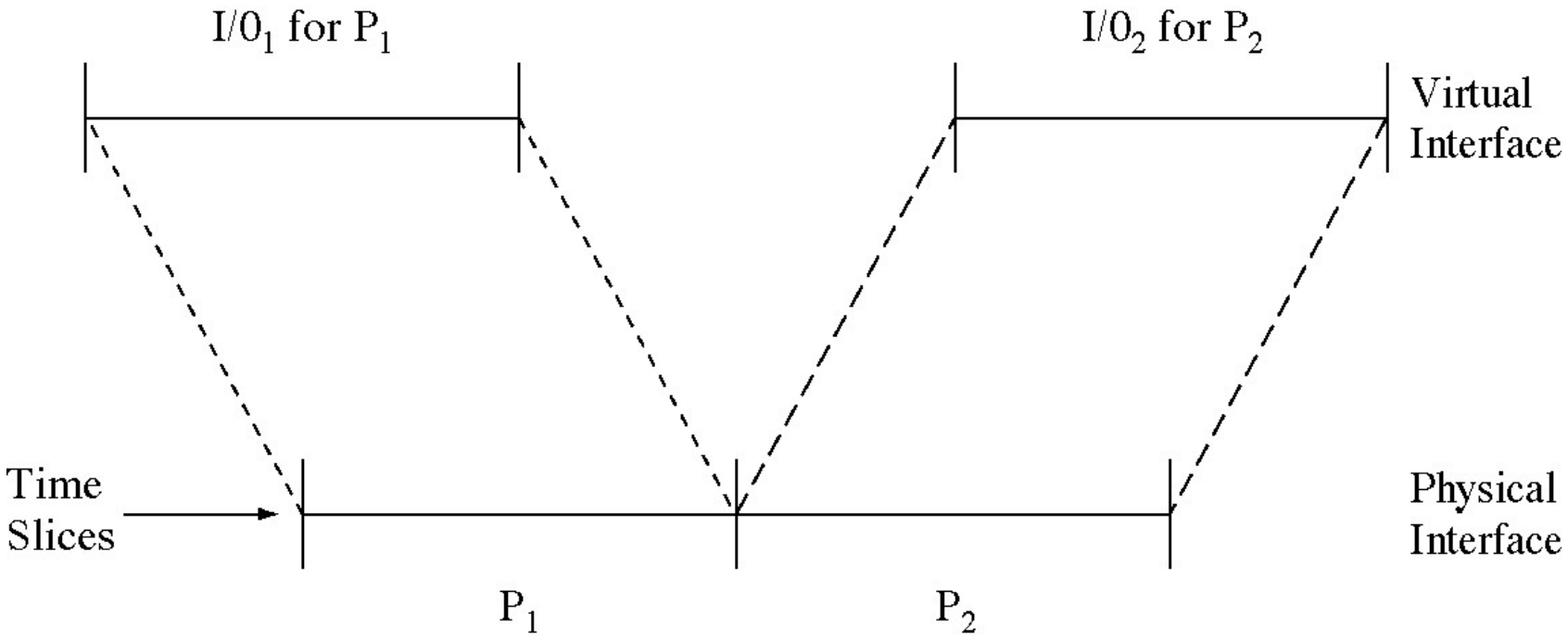
Time-multiplexing the processor



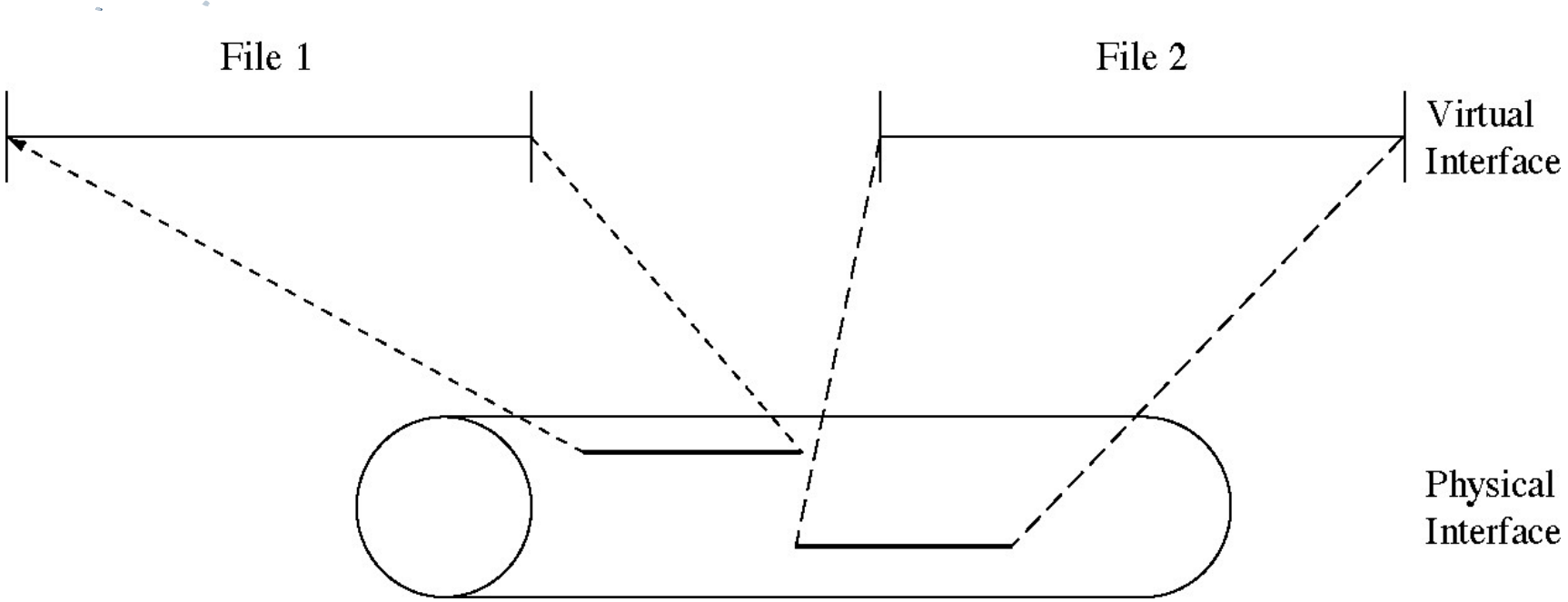
Space-multiplexing memory



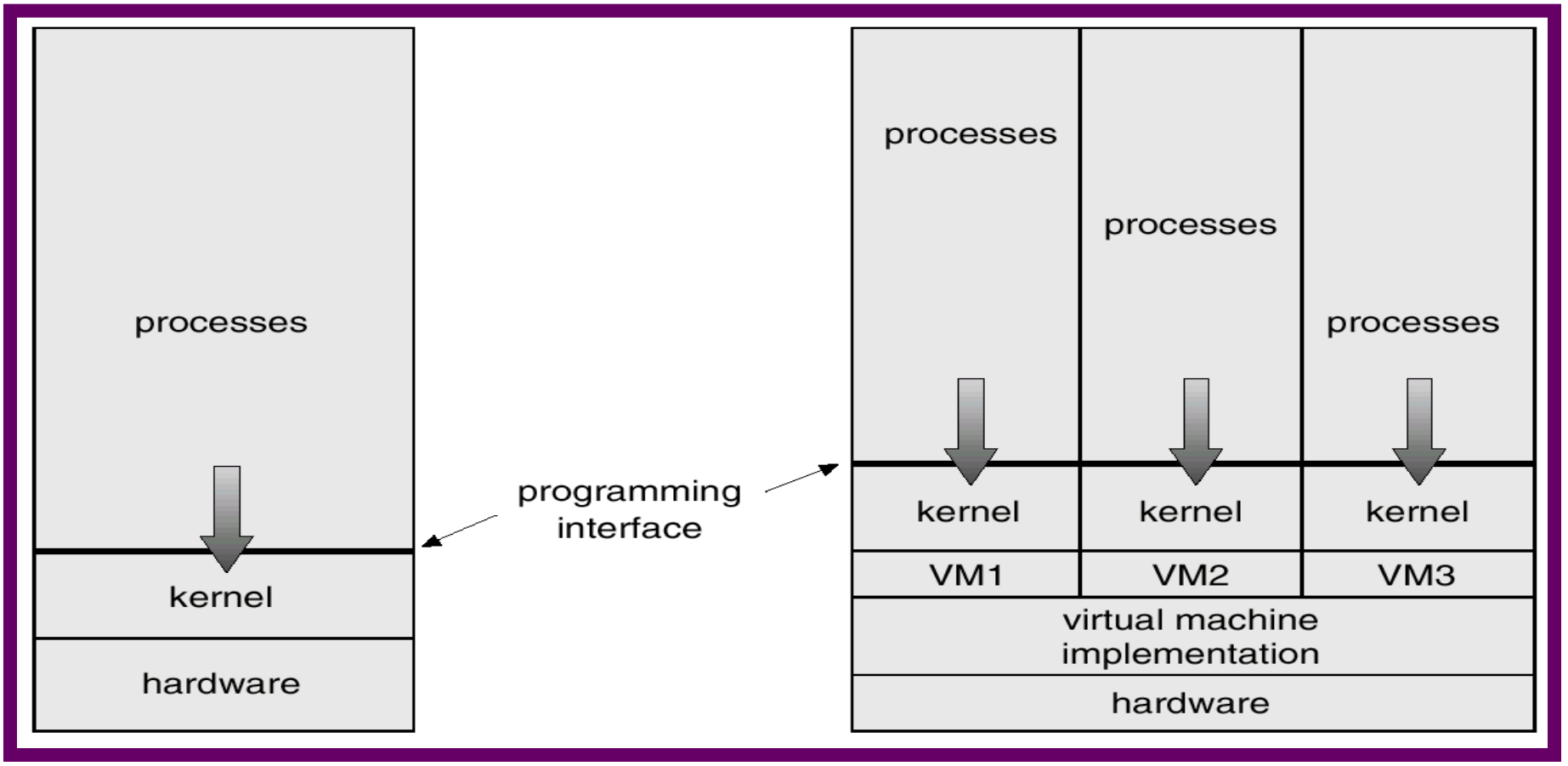
Time-multiplexing I/O devices



Space-multiplexing the disk



System Models



Non-virtual Machine

Virtual Machine

Outline

- Operating System Services
- User Operating System Interface
- System Calls
- Types of System Calls
- System Programs
- Operating System Design and Implementation
- Operating System Structure
- Virtual Machines
- **Operating System Debugging**
- **Operating System Generation**
- **System Boot**

Operating-System Debugging

- **Debugging** is finding and fixing errors, or **bugs**
- OSes generate **log files** containing error information
- Failure of an application can generate **core dump** file capturing memory of the process
- Operating system failure can generate **crash dump** file containing kernel memory
- Beyond crashes, performance tuning can optimize system performance
- Kernighan's Law: "Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it."
- DTrace tool in Solaris, FreeBSD, Mac OS X allows live instrumentation on production systems
  - **Probes** fire when code is executed, capturing state data and sending it to consumers of those probes

Solaris 10 dtrace Following System Call

```
# ./all.d `pgrep xclock` XEventsQueued
dtrace: script './all.d' matched 52377 probes
CPU FUNCTION
  0 -> XEventsQueued                       U
  0   -> _XEventsQueued                    U
  0     -> _X11TransBytesReadable          U
  0     <- _X11TransBytesReadable          U
  0     -> _X11TransSocketBytesReadable    U
  0     <- _X11TransSocketBytesreadable    U
  0     -> ioctl                           U
  0       -> ioctl                         K
  0         -> getf                        K
  0           -> set_active_fd             K
  0           <- set_active_fd             K
  0         <- getf                        K
  0         -> get_udatamodel              K
  0         <- get_udatamodel              K
...
  0         -> releasef                    K
  0           -> clear_active_fd           K
  0           <- clear_active_fd           K
  0           -> cv_broadcast              K
  0           <- cv_broadcast              K
  0         <- releasef                    K
  0       <- ioctl                         K
  0     <- ioctl                           U
  0   <- _XEventsQueued                    U
  0 <- XEventsQueued                       U
```

System generation (SYSGEN)

- The OS program is normally distributed on disk or tape.
- A special program SYSGEN reads from a given file and asks the operator regarding specific configuration of the system.
  - Probes the hardware directly to determine what components are there.
- Regenerate/Configure the OS for new machine parameters
  - CPU type
  - Machine parameters include
    - ✔ Memory size, I/O device parameters, address maps etc.
    - ✔ Mechanisms describe how things are done
  - Devices
  - Options of OS
    - ✔ CPU scheduling algorithm, buffer size.
- Approaches
  - Recompile
  - Have precompiled parameterized libraries: link rather than compiling
  - Use table driven run-time selection

Operating System Generation

- 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

System Boot

- Operating system must be made available to hardware so hardware can start it
  - Small piece of code – **bootstrap loader**, locates the kernel, loads it into memory, and starts it
  - Sometimes two-step process where **boot block** at fixed location loads bootstrap loader
  - When power initialized on system, execution starts at a fixed memory location
    - ✔ Firmware used to hold initial boot code