Operating Systems (CSE531)
Lecture # 05

Manish Shrivastava

LTRC, IIIT Hyderabad

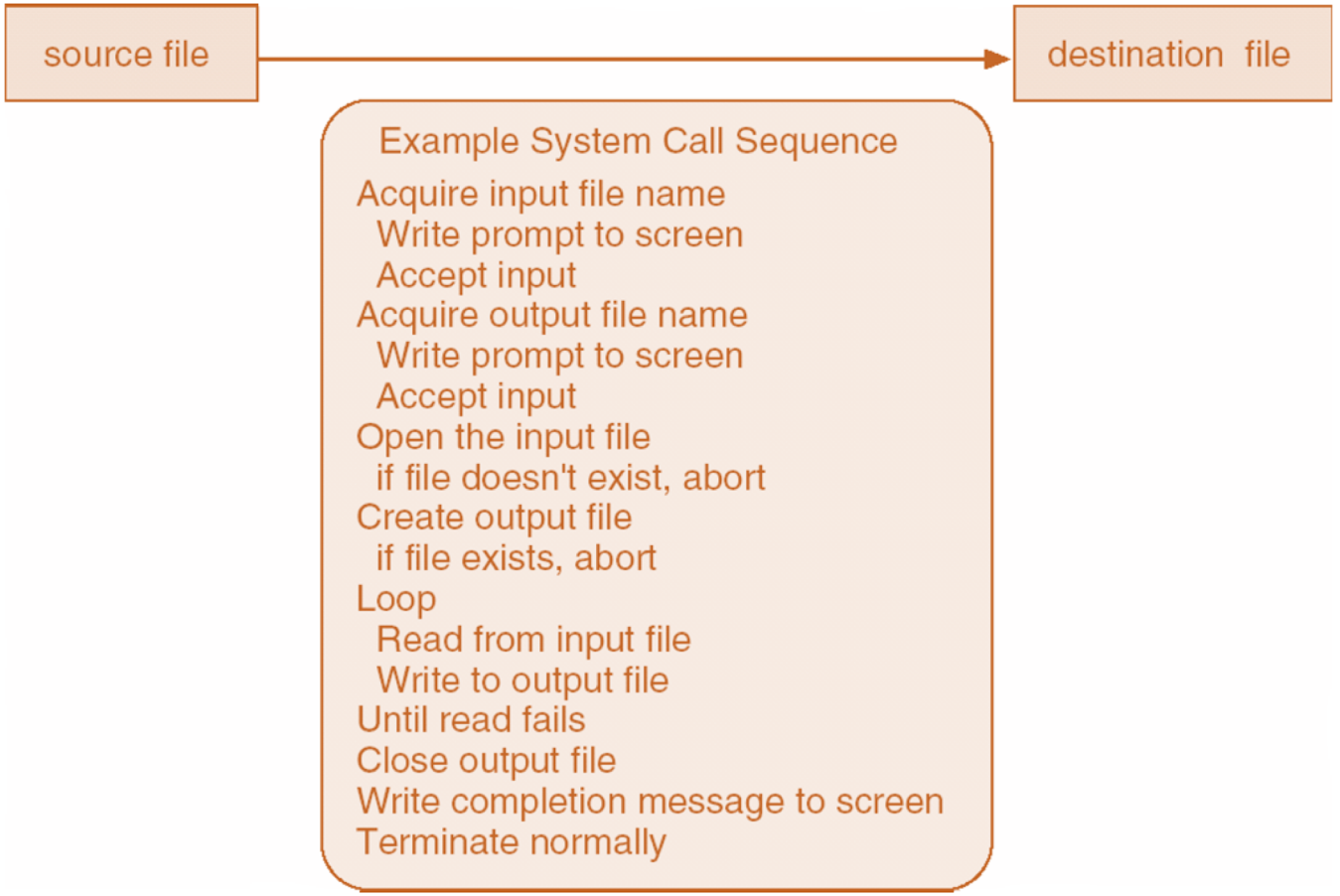
System Calls

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

*Portable Operating System Interface (POSIX)

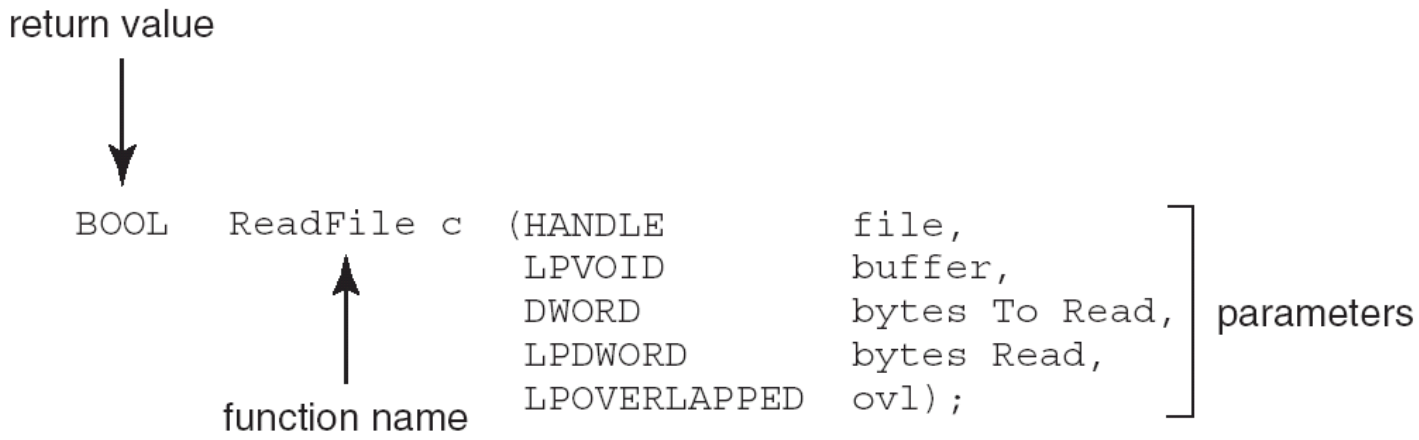
Example of System Calls

- 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

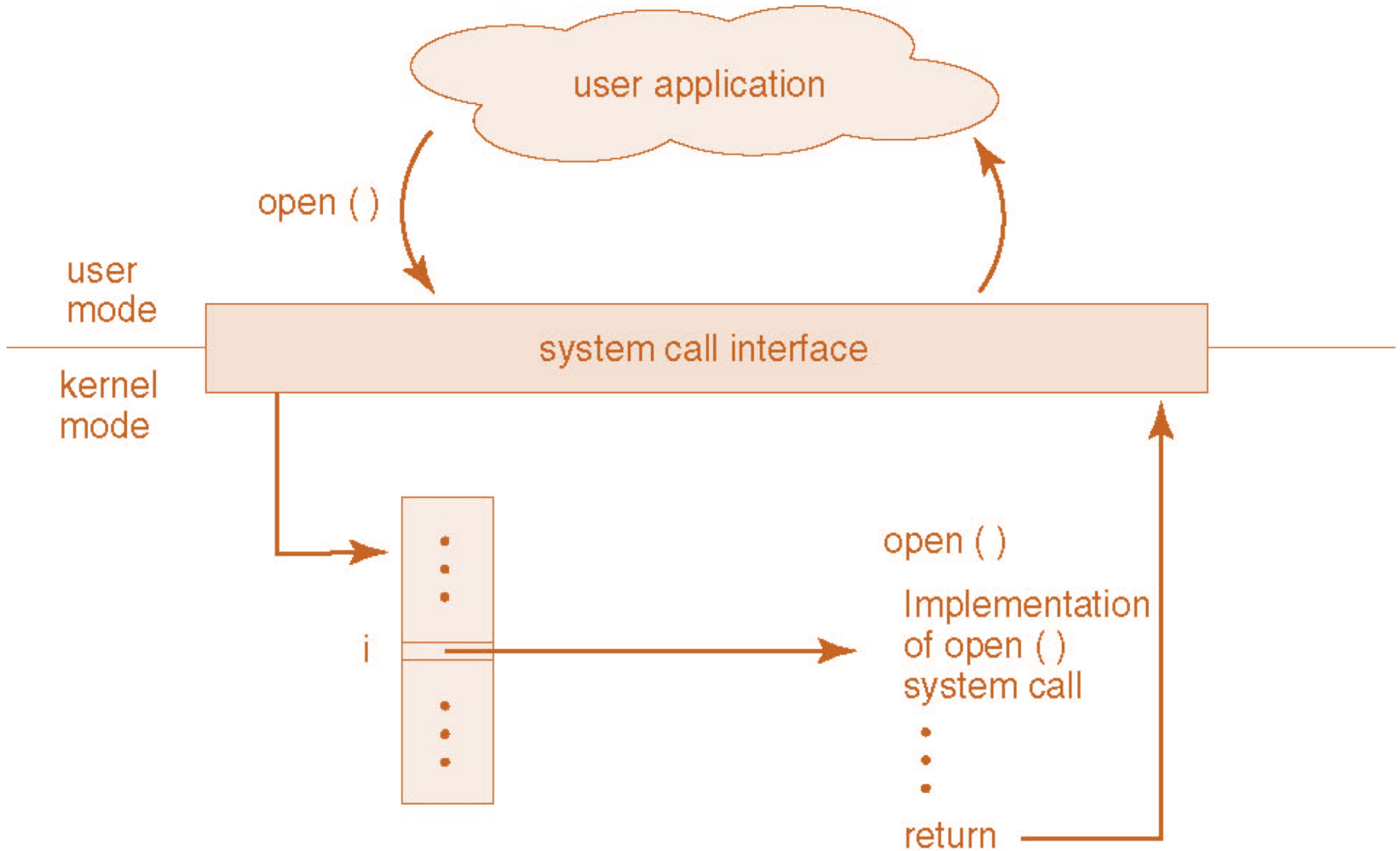
System Call Implementation

- 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

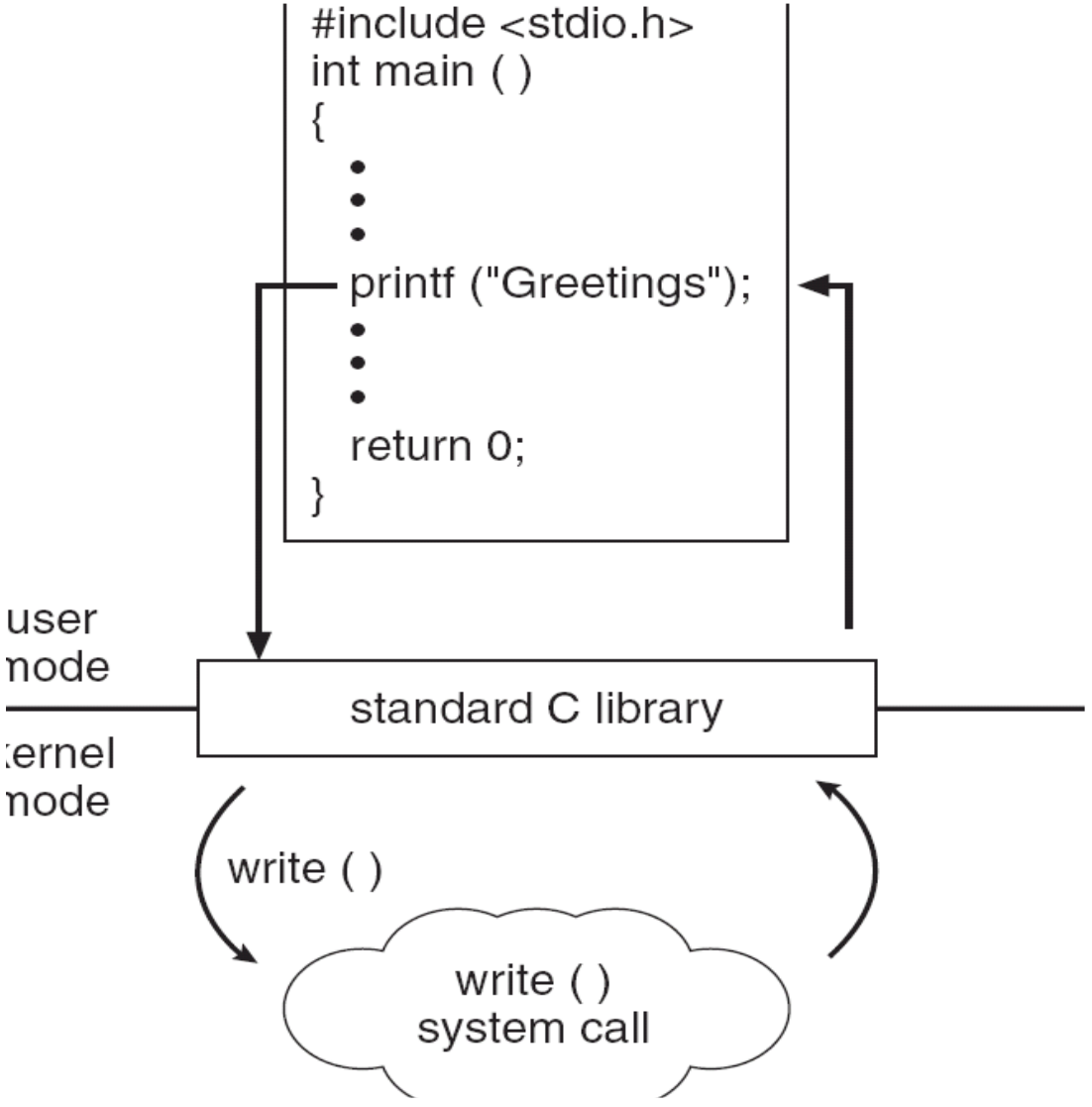
- The caller need know nothing about how the system call is implemented
  - Just needs to obey API and understand what OS will do as a result call
  - Most details of OS interface hidden from programmer by API
    - Managed by run-time support library (set of functions built into libraries included with compiler)

API – System Call – OS Relationship



Standard C Library Example

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

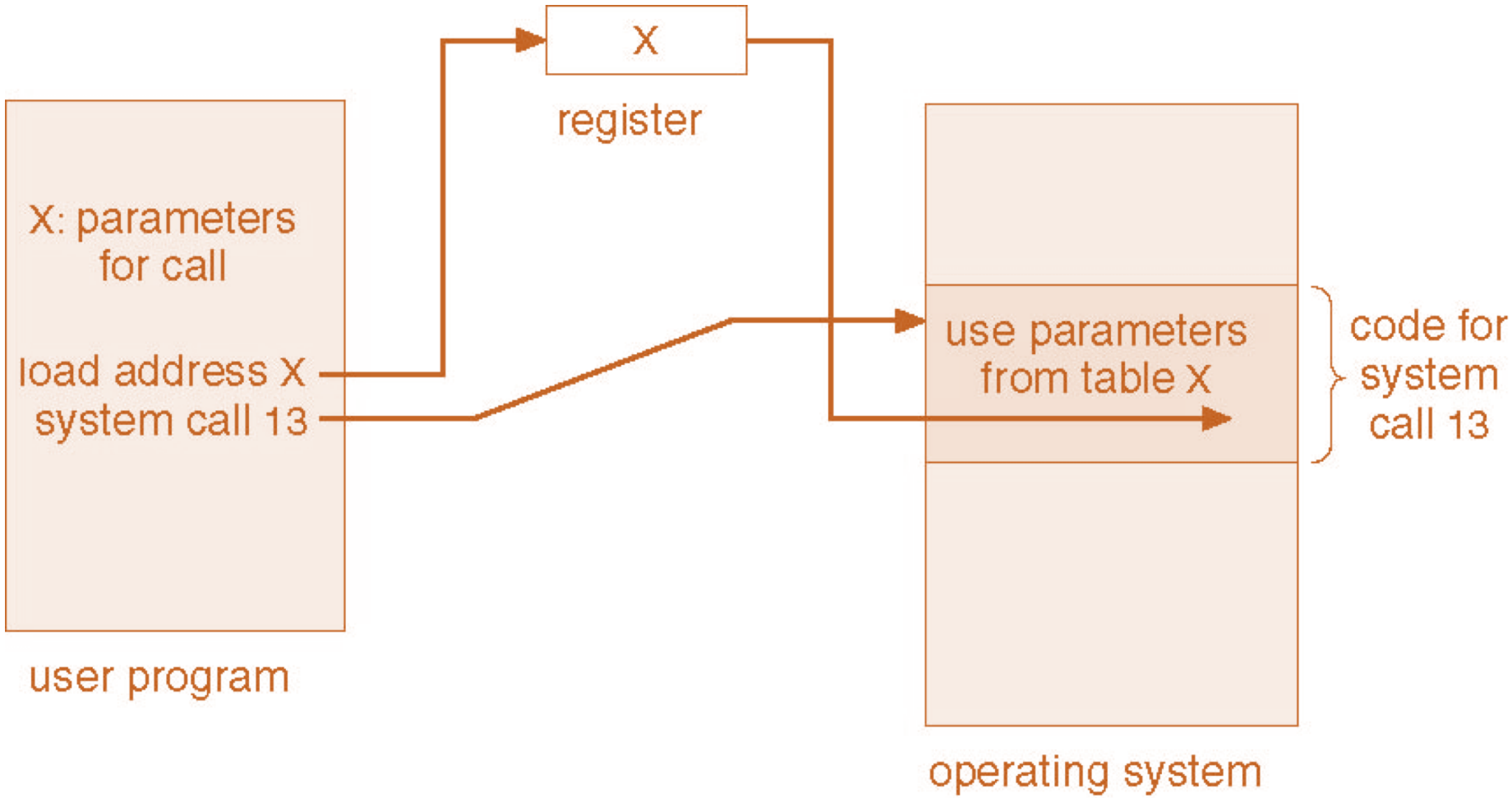


System Call Parameter Passing

- 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

- 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

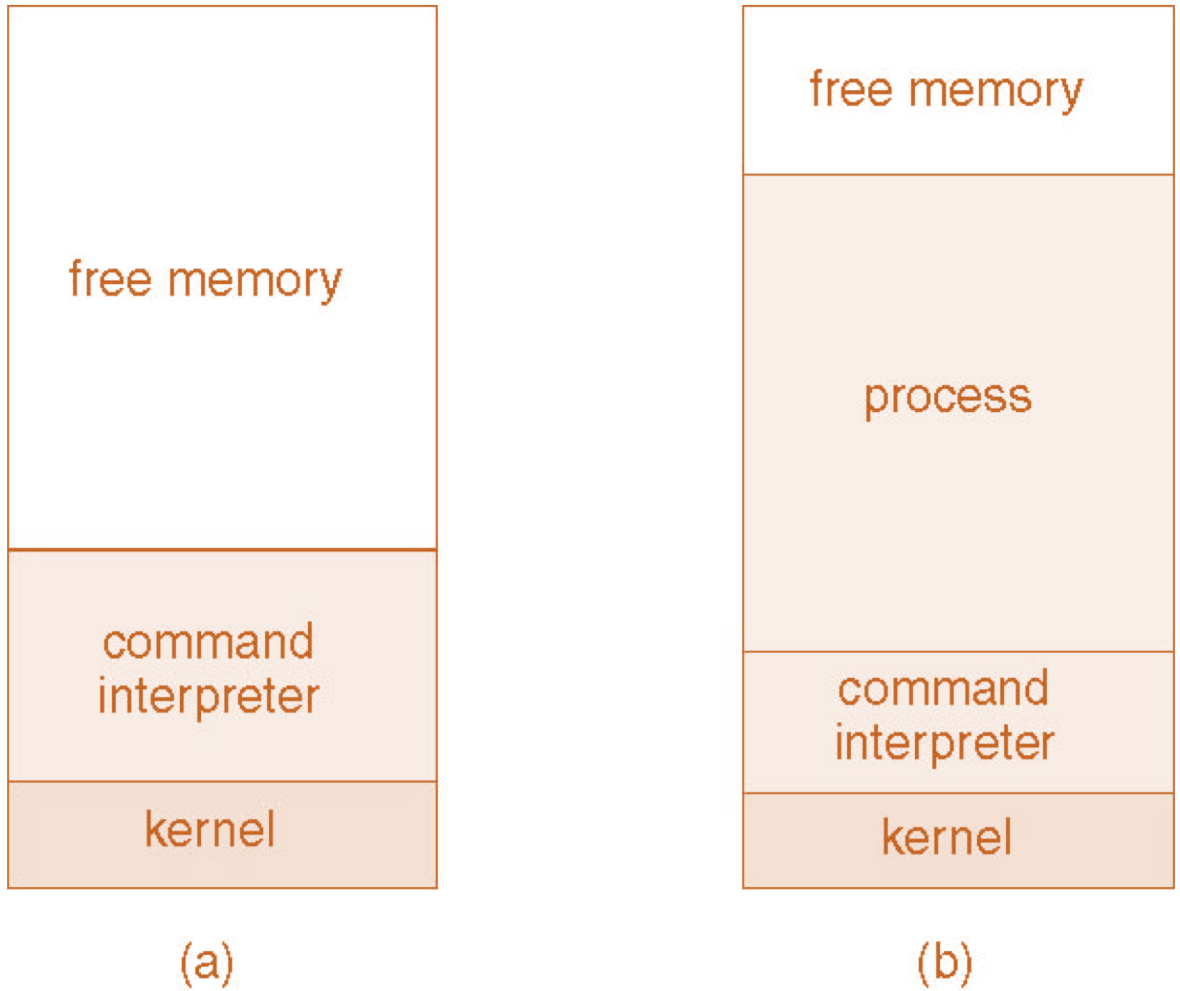
Examples of Windows and Unix System Calls

	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()

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

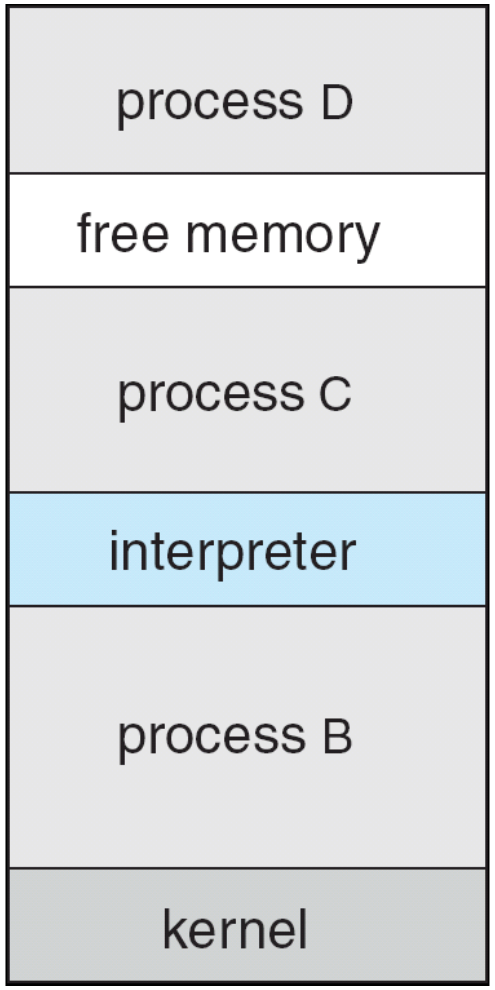
MS-DOS execution



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

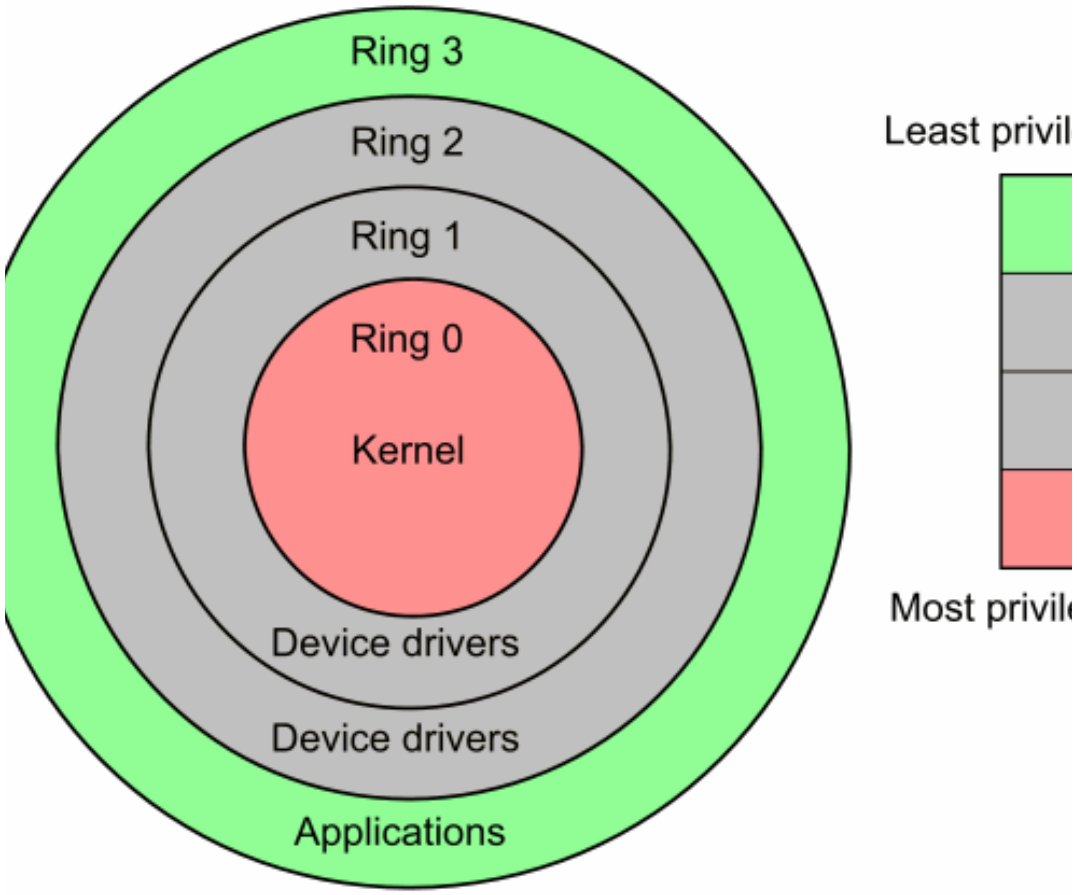
FreeBSD Running Multiple Programs



Linux system calls

- Operating systems offer processes running in User Mode ***a set of interfaces*** to interact with ***hardware devices*** such as
  - the **CPU**
  - disks
  - printers.

- **Unix** systems implement most interfaces between ***User Mode processes*** and ***hardware devices*** by means of ***system calls*** issued to the kernel.



POSIX APIs vs. System Calls

- An *application programmer interface* (API) is a function definition that specifies how to obtain a given service.

- A *system call* is an explicit request to the kernel made via a software interrupt.

From a Wrapper Routine to a System Call

- Unix systems include several libraries of functions that provide APIs to programmers.
- Some of the APIs defined by the ***libc*** standard C library refer to wrapper routines (routines whose only purpose is to issue a system call).
- Usually, each system call has a corresponding wrapper routine, which defines the API that application programs should employ.

APIs and System Calls

- An **API** does not necessarily correspond to a specific system call.
  - First of all, the **API** could offer its services directly in User Mode. (For something abstract such as math functions, there may be no reason to make system calls.)
  - Second, a single **API** function could make several system calls.
  - Moreover, several **API** functions could make the same system call, but wrap extra functionality around it.

Example of Different APIs Issuing the Same System Call

- In Linux, the *malloc( )*, *calloc( )*, and *free( )* APIs are implemented in the *libc* library.

- The code in this library keeps track of the allocation and deallocation requests and uses the *brk( )* system call to enlarge or shrink the process heap.

- Linux defines a set of seven macros called _syscall0 through _syscall6.

- Example: _syscall3(int,write,int,fd,const char *,buf,unsigned int,count)

The Return Value of a Wrapper Routine

- Most wrapper routines return an integer value, whose meaning depends on the corresponding system call.
- A return value of -1 usually indicates that the kernel was unable to satisfy the process request.
- A failure in the system call handler may be caused by
  - invalid parameters
  - a lack of available resources
  - hardware problems, and so on.
- The specific error code is contained in the ***errno*** variable, which is defined in the *libc* library.

Execution Flow of a System Call

- When a User Mode process invokes a system call, the CPU switches to Kernel Mode and starts the execution of a kernel function.

- As we will see in the next section, in the 80x86 architecture a Linux system call can be invoked in two different ways.

- The net result of both methods, however, is a jump to an assembly language function called the system call handler.

System Call Number

- Because the kernel implements many different system calls, the User Mode process must pass a parameter called the system call number to identify the required system call.

- The ***eax*** register is used by Linux for this purpose.

- Additional parameters are usually passed when invoking a system call.

The Return Value of a System Call

- All system calls return an integer value.
- The conventions for these return values are different from those for wrapper routines. In the kernel:
    - positive or 0 values denote a successful termination of the system call
    - negative values denote an error condition
        - In the latter case, the value is the negation of the error code that must be returned to the application program in the **`errno`** variable.
    - The **`errno`** variable is not set or used by the kernel. Instead, the wrapper routines handle the task of setting this variable after a return from a system call.

Operations Performed by a System Call

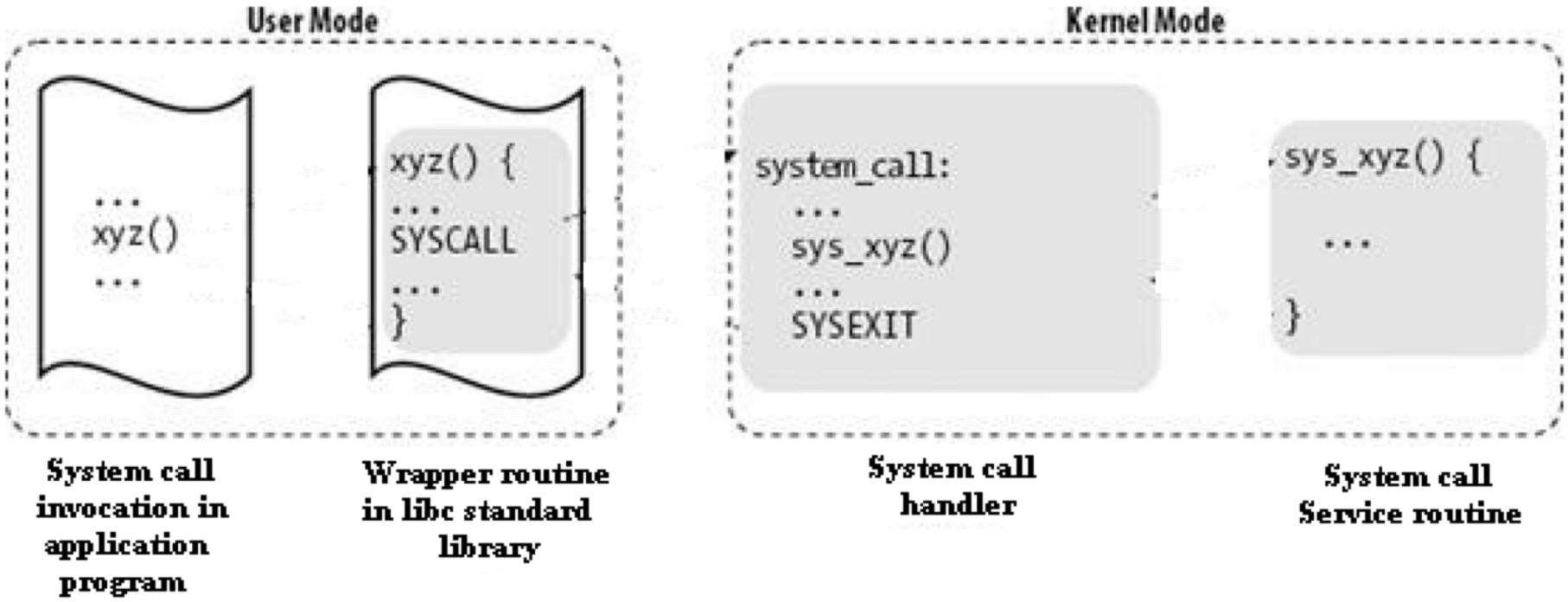
- The ***system call handler***, which has a structure similar to that of the other ***exception handlers***, performs the following operations:
  - *Saves the contents of most registers in the Kernel Mode stack.
  - Handles the system call by invoking a corresponding **C** function called the ***system call service routine***.
  - *Exits from the handler:
    - the registers are loaded with the values saved in the Kernel Mode stack
    - the **CPU** is switched back from Kernel Mode to User Mode.

*This operation is common to all system calls and is coded in assembly language.

Naming Rules of System Call Service Routines

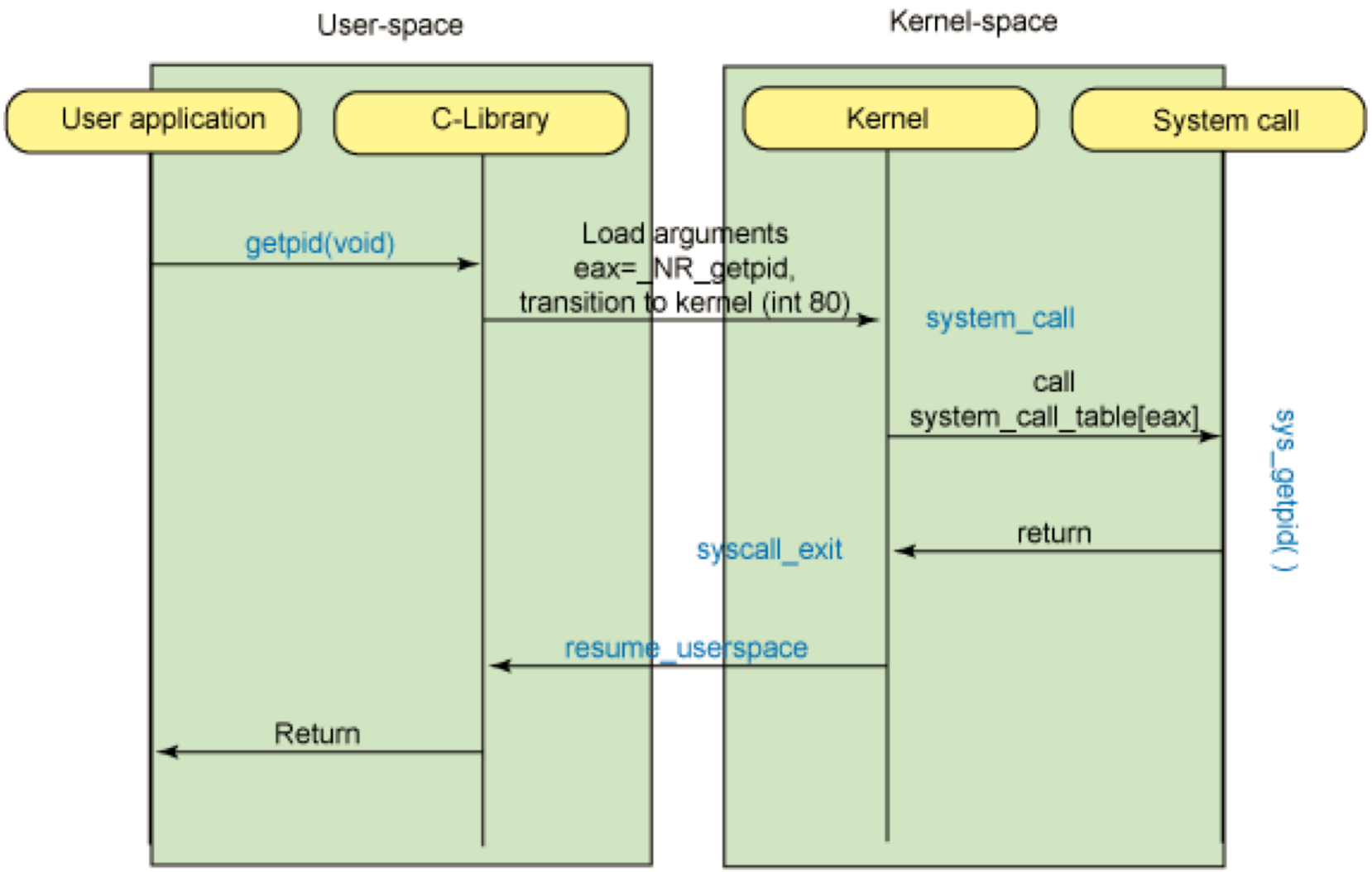
- The name of the service routine associated with the ***xyz( )*** system call is usually **sys_*xyz*( )**; there are, however, a few exceptions to this rule.

Control Flow Diagram of a System Call



- The **arrows** denote the execution flow between the functions.
- The terms "**SYSCALL**" and "**SYSEXIT**" are placeholders for the actual assembly language instructions that switch the **CPU**, respectively, from User Mode to Kernel Mode and from Kernel Mode to User Mode.

Control Flow Diagram of a System Call



Examples : File Creation

```
/* creat.c */
#include <stdio.h>
#include <sys/types.h> /* defines types used by sys/stat.h */
#include <sys/stat.h> /* defines S_IREAD & S_IWRITE */
int main() {
    int fd;
    fd = creat("datafile.dat", S_IREAD | S_IWRITE);
    if (fd == -1)
            printf("Error in opening datafile.dat\n");
    else {
            printf("datafile.dat opened for read/write access\n");
            printf("datafile.dat is currently empty\n");
    }
    close(fd);
    exit (0);
}
```

Stat.h

```
#define S_IRWXU 0000700 /* -rwx------ */
#define S_IREAD 0000400 /* read permission, owner */
#define S_IRUSR S_IREAD
#define S_IWRITE 0000200 /* write permission, owner */
#define S_IWUSR S_IWRITE
#define S_IEXEC 0000100 /* execute/search permission, owner */
#define S_IXUSR S_IEXEC
#define S_IRWXG 0000070 /* ----rwx--- */
#define S_IRGRP 0000040 /* read permission, group */
#define S_IWGRP 0000020 /* write " " */
#define S_IXGRP 0000010 /* execute/search " " */
#define S_IRWXO 0000007 /* -------rwx */
#define S_IROTH 0000004 /* read permission, other */
#define S_IWOTH 0000002 /* write " " */
#define S_IXOTH 0000001 /* execute/search " " */
```

File Open

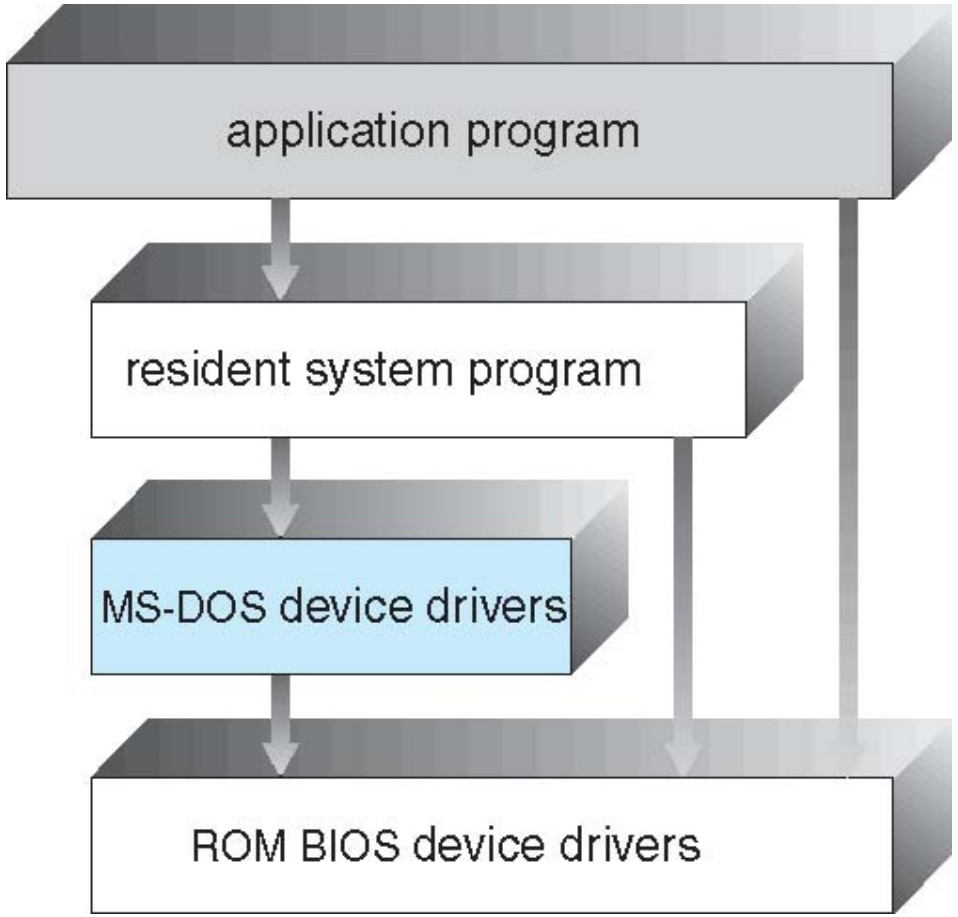
```
/* open.c */
#include <fcntl.h> /* defines options flags */
#include <sys/types.h> /* defines types used by sys/stat.h */
#include <sys/stat.h> /* defines S_IREAD & S_IWRITE */
static char message[] = "Hello, world";
int main() {
        int fd; char buffer[80];
        fd = open("datafile.dat",O_RDWR | O_CREAT | O_EXCL, S_IREAD | S_IWRITE);
        if (fd != -1) {
                        printf("datafile.dat opened for read/write access\n");
                        write(fd, message, sizeof(message));
                        lseek(fd, 0L, 0); /* go back to the beginning of the file */
                        if (read(fd, buffer, sizeof(message)) == sizeof(message))
                                        printf("\"%s\" was written to datafile.dat\n", buffer);
                        else
                                        printf("*** error reading datafile.dat ***\n");
                        close (fd);
                        } else printf("*** datafile.dat already exists ***\n");
        exit (0);
}
```

System Structures and Virtual Machines

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

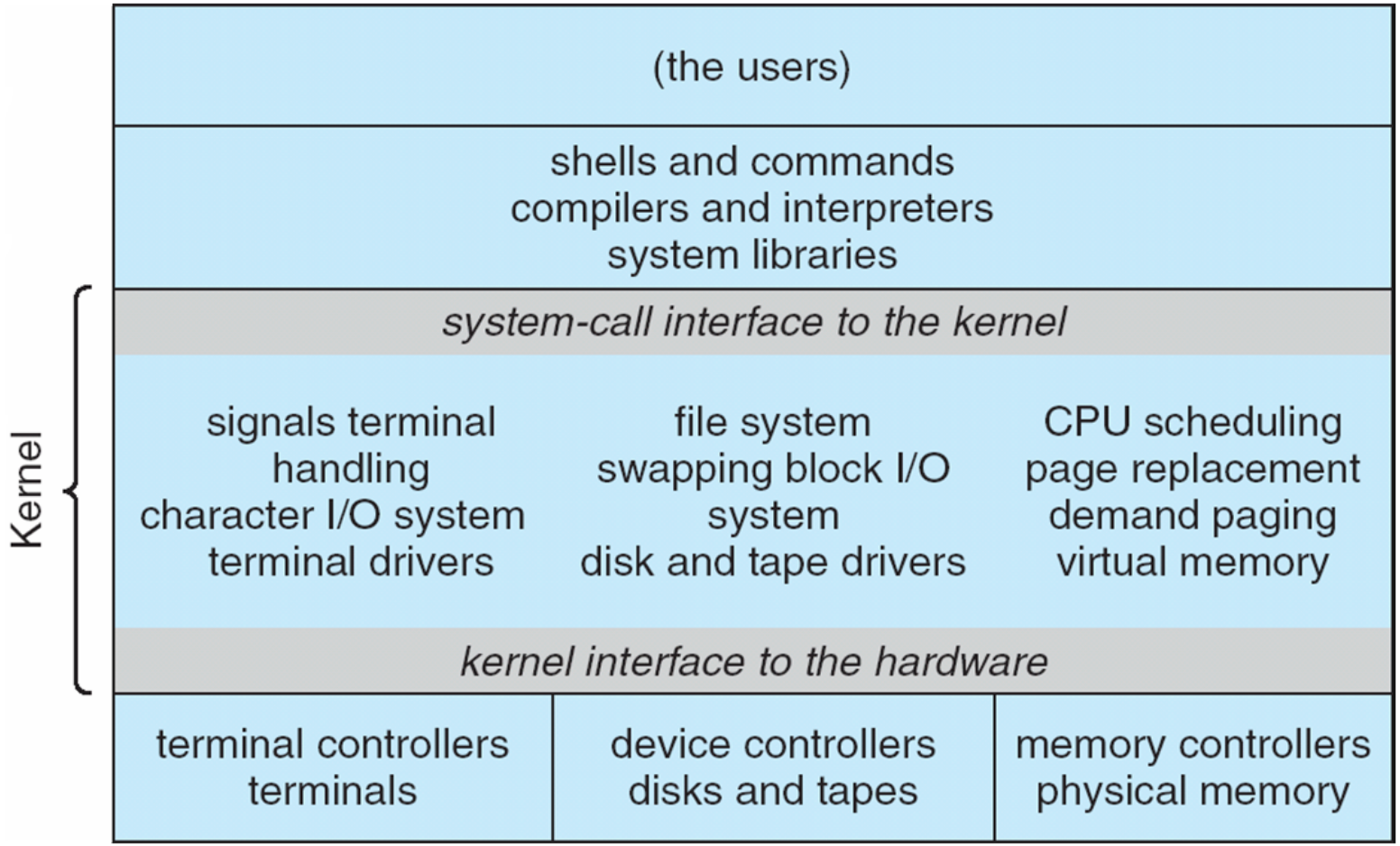
MS-DOS Layer Structure



UNIX

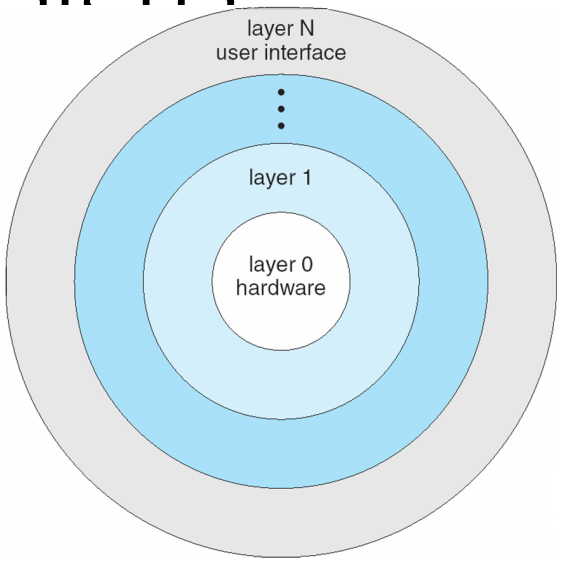
- UNIX – limited by hardware functionality, the original UNIX operating system had limited structuring. The UNIX OS consists of two separable parts
  - Systems programs
  - The kernel
    - Consists of everything below the system-call interface and above the physical hardware
    - Provides the file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

Traditional UNIX System Structure



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

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
1	Electronic circuits	Registers, gates, busses	Clear, transfer, activate, complement

OS 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.

OS 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

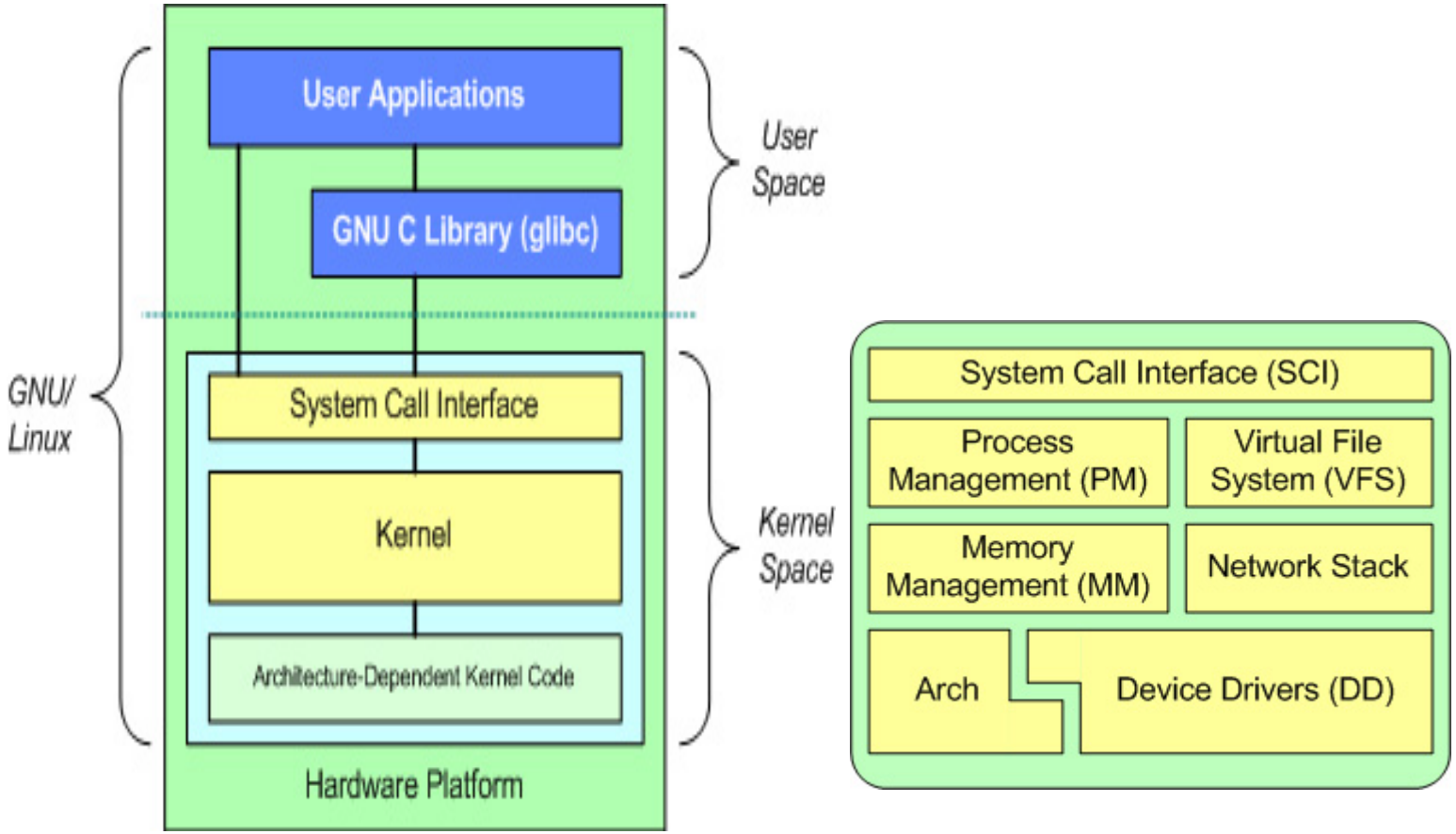
- Moves as much from the kernel into "*user*" space
- Communication takes place between user modules using message passing
- Benefits:
  - Easier to extend a microkernel
  - Easier to port the operating system to new architectures
  - More reliable (less code is running in kernel mode)
  - More secure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

Modules

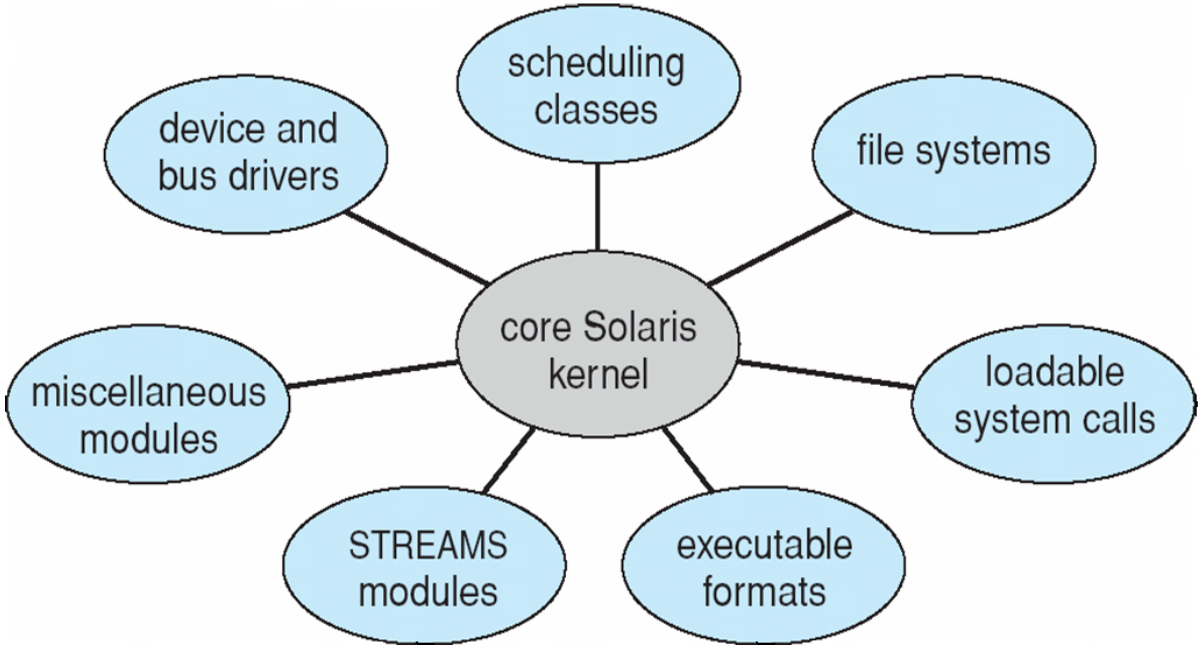
- Most modern operating systems implement kernel modules
  - Uses object-oriented approach
  - Each core component is separate
  - Each talks to the others over known interfaces
  - Each is loadable as needed within the kernel

- Overall, similar to layers but with more flexible

Linux


User Applications
GNU C Library (glibc)
System Call Interface
Kernel
Architecture-Dependent Kernel Code
Hardware Platform
GNU/ Linux
User Space
Kernel Space
System Call Interface (SCI)
Process Management (PM)
Virtual File System (VFS)
Memory Management (MM)
Network Stack
Arch
Device Drivers (DD)

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.

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.

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.

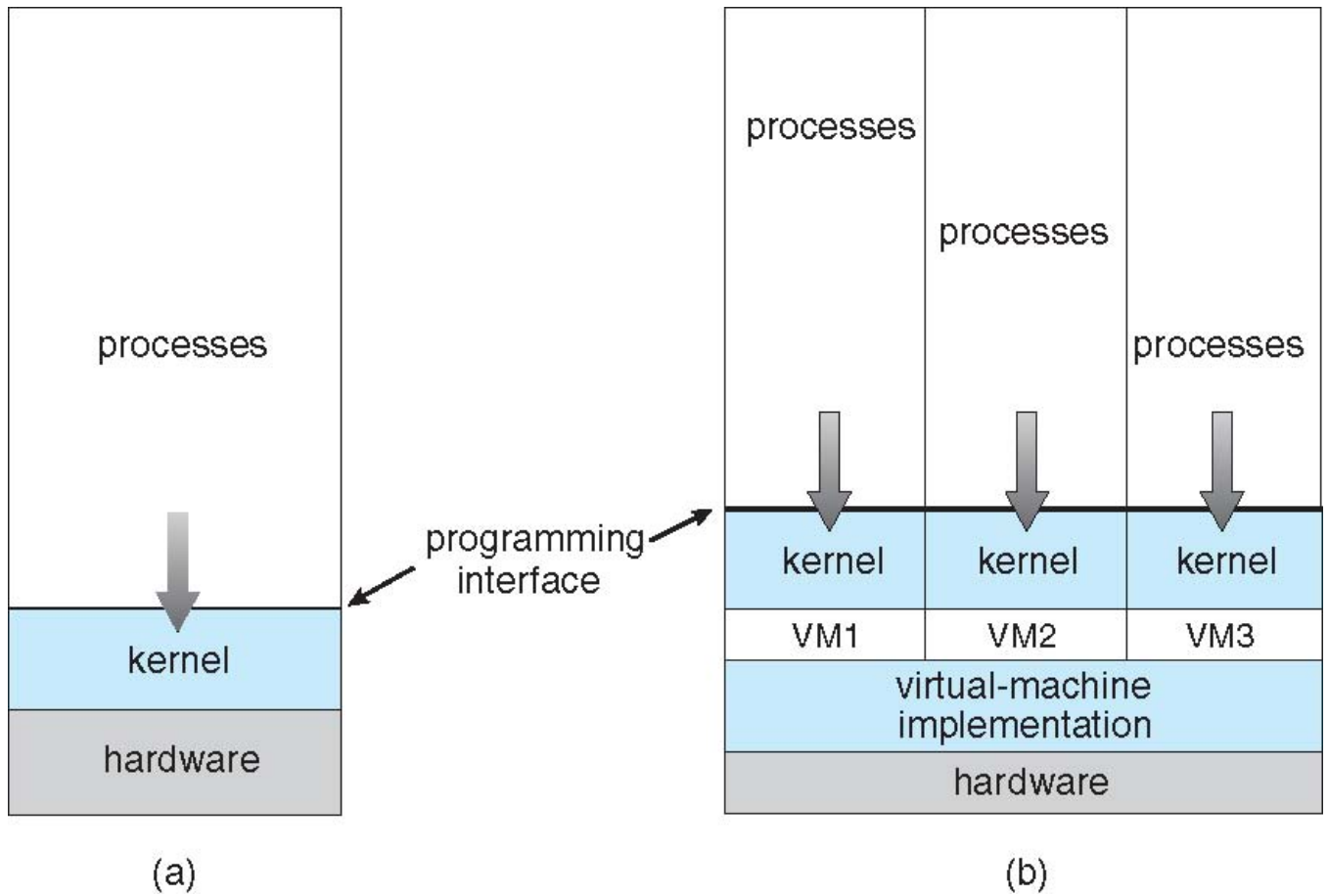
Structure View: Virtual machines

- 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 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.)



Para-virtualization

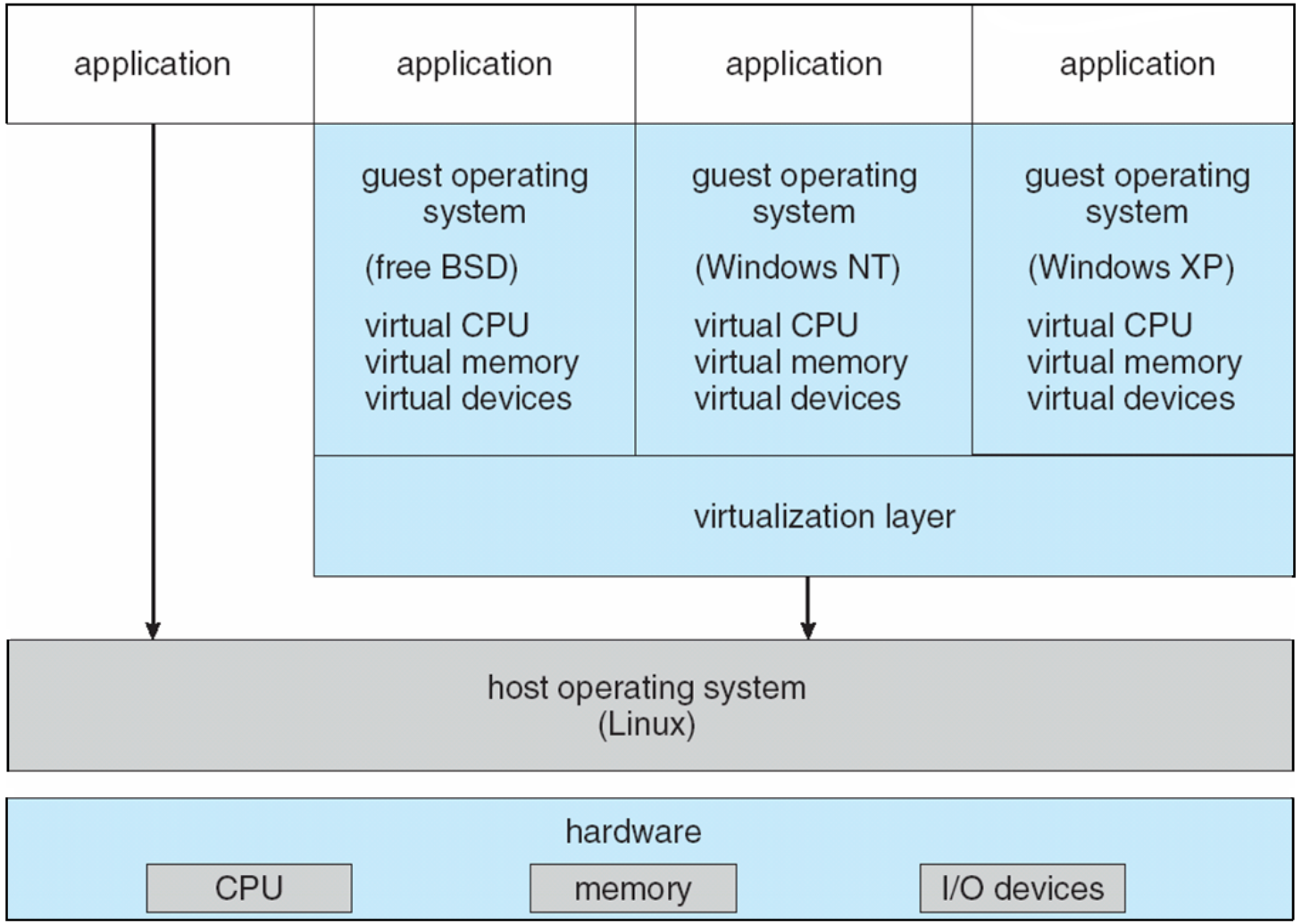
- Presents guest with system similar but not identical to hardware

- The guest operating system (the one being virtualized) is aware that it is a guest and accordingly has drivers that, instead of issuing hardware commands, simply issue commands directly to the host operating system.
  - This also includes memory and thread management as well, which usually require unavailable privileged instructions in the processor.

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

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

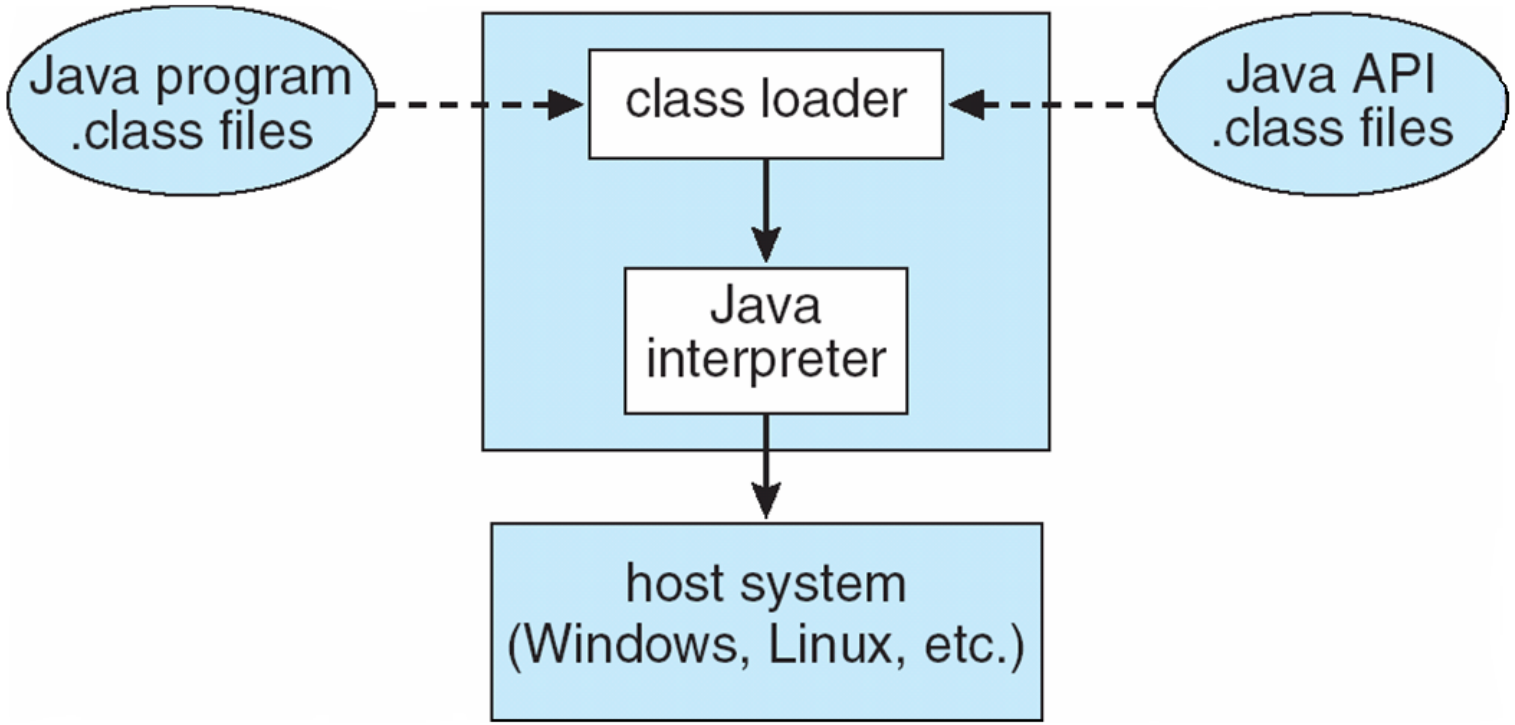
Structure view: Virtual machines (cont..)

- 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.
  - System compatibility problems can be solved
- Disadvantages
  - Performance degradation is inevitable

JAVA

- JAVA is designed by SUN Microsystems.
- 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



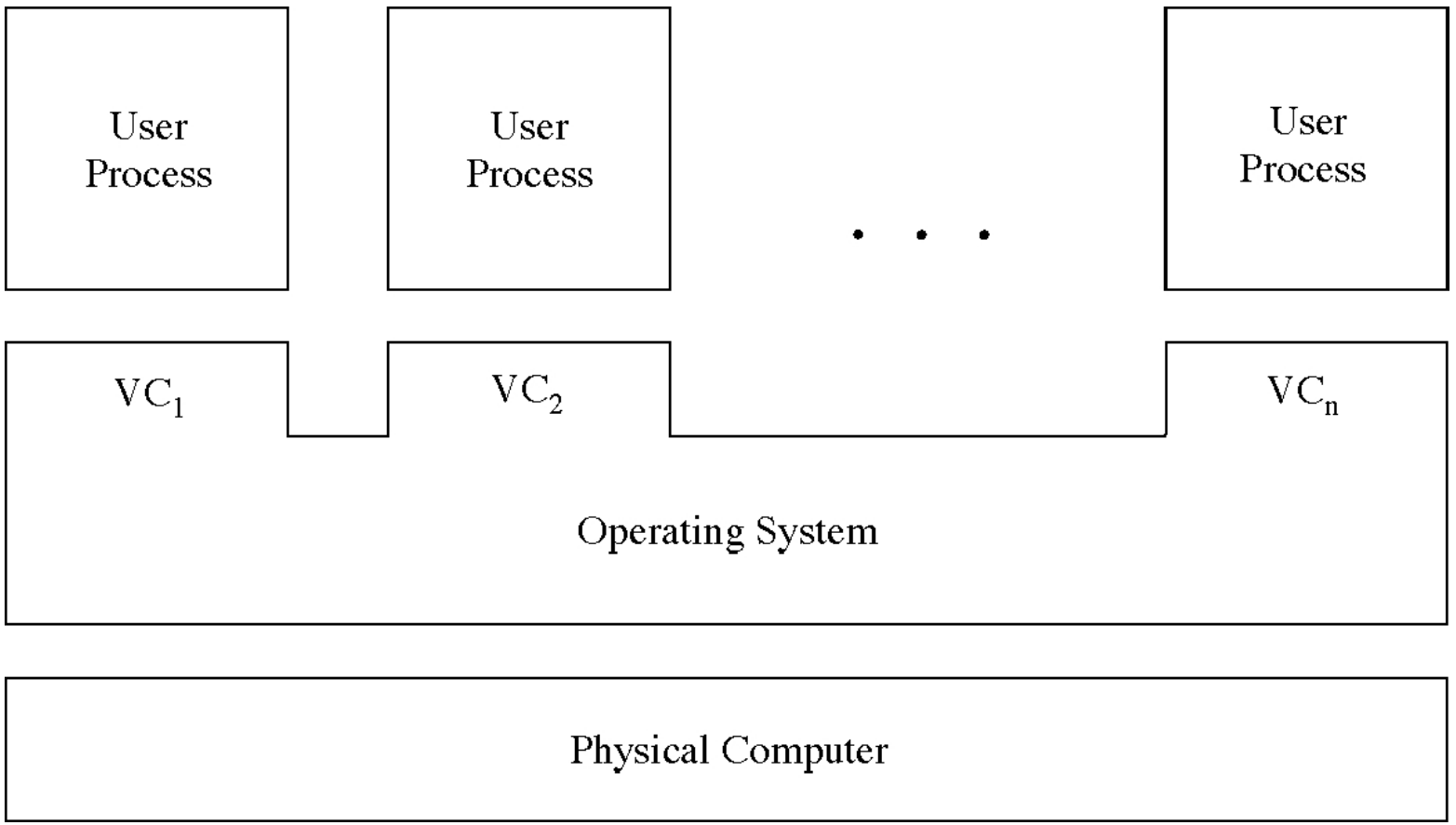
Types of multiplexing

- 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



Boot :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

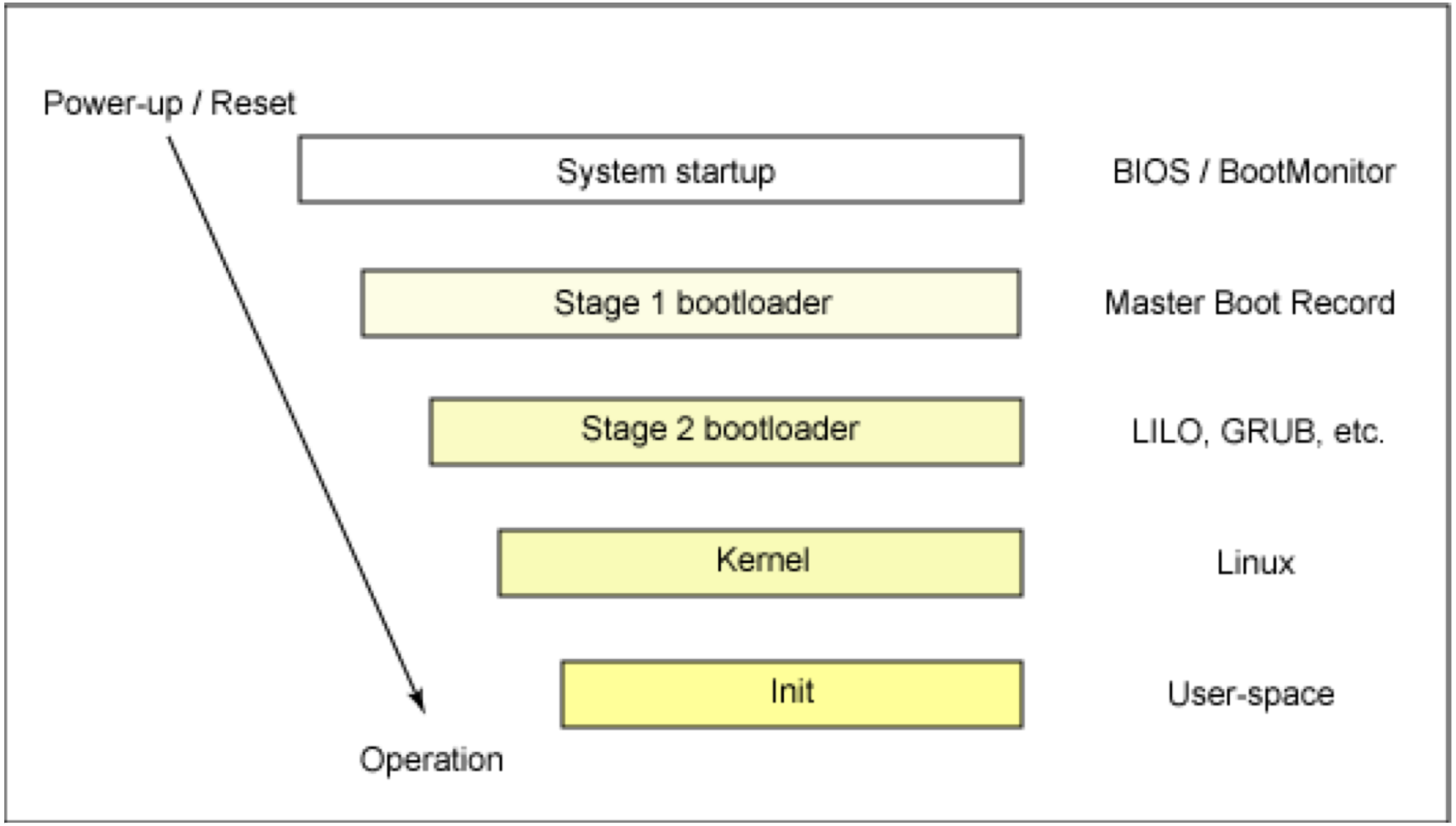
Operating System Generation

- 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

Linux Boot Process



THANK YOU