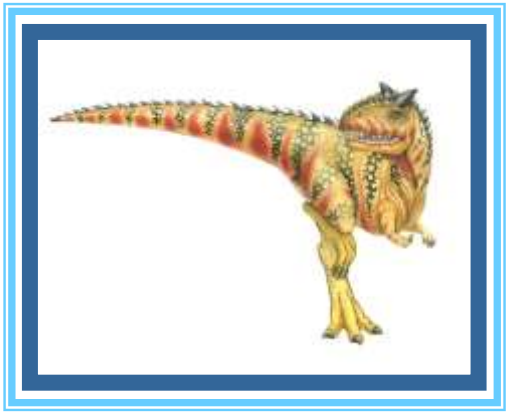
# Chapter 2: Operating-System Structures



# Outline

**Outline**

- Operating System Services
- User Operating System Interface
- System Calls
- System Programs
- Operating System Structure

**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

# Operating System Services

- **For user**
  - User interface
  - Program execution
  - I/O operations
  - File-system manipulation
  - Communication process communication
  - Error detection and handling
- **For System**: efficiency ad sharing
  - Resource allocation
  - Accounting
  - Protection and security

# 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

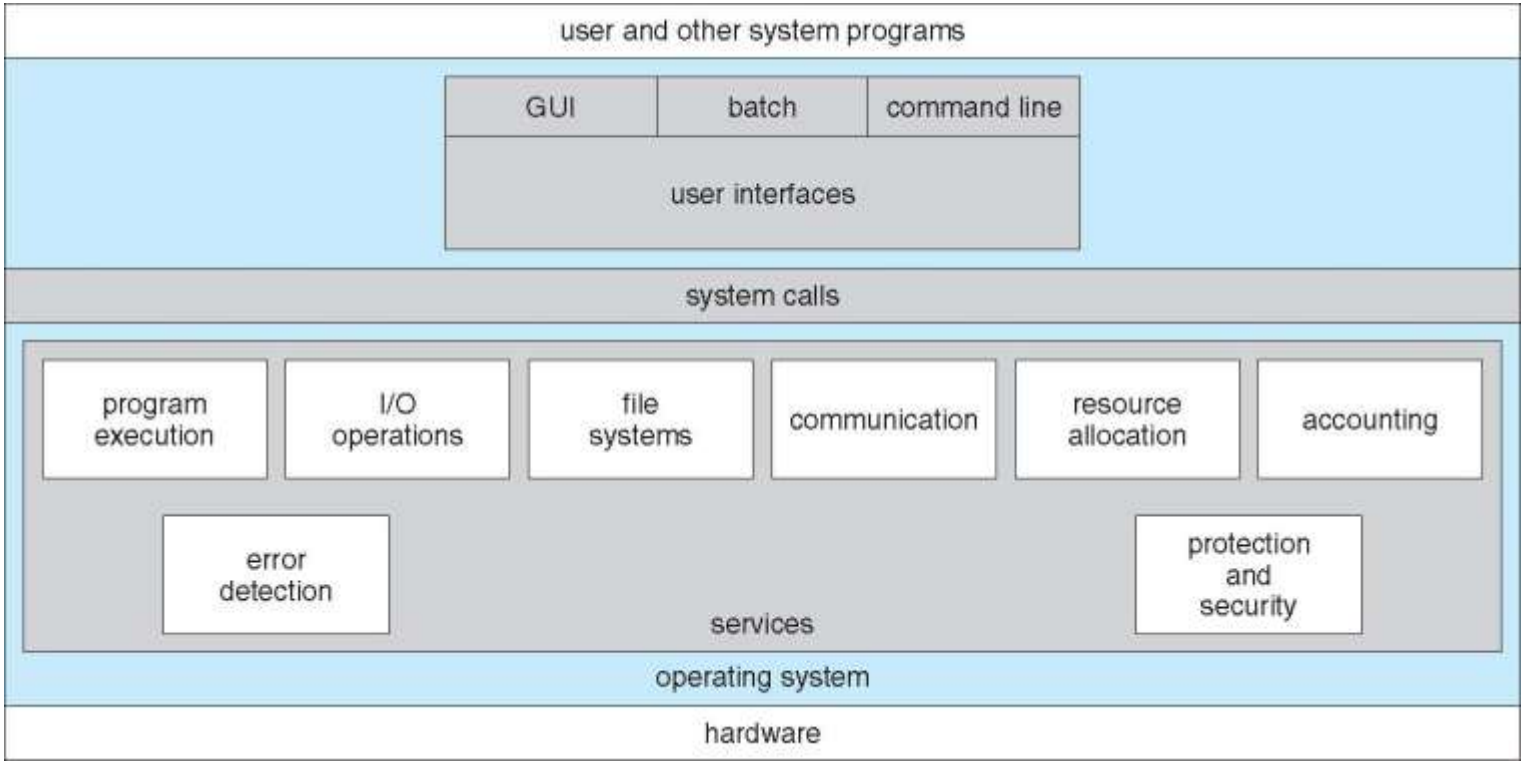
# Operating System Services (Cont.)

- One set of operating-system services provides functions that are helpful to the user (Cont.):
  - **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.
  - **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)
  - **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.)

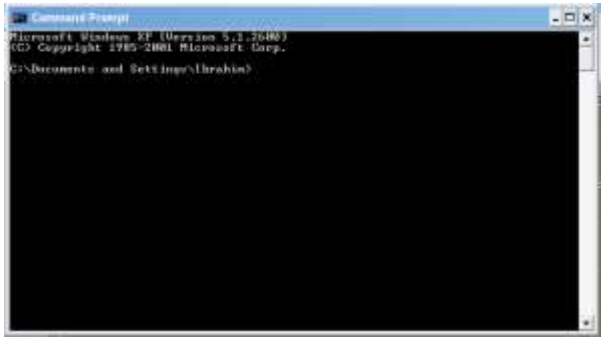
- Another set of OS functions exists for ensuring the efficient operation of the system itself via resource sharing
  - **Resource allocation -** When multiple users or multiple jobs running concurrently, resources must be allocated to each of them
    - Many types of resources - CPU cycles, main memory, file storage, I/O devices.
  - **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

# A View of Operating System Services



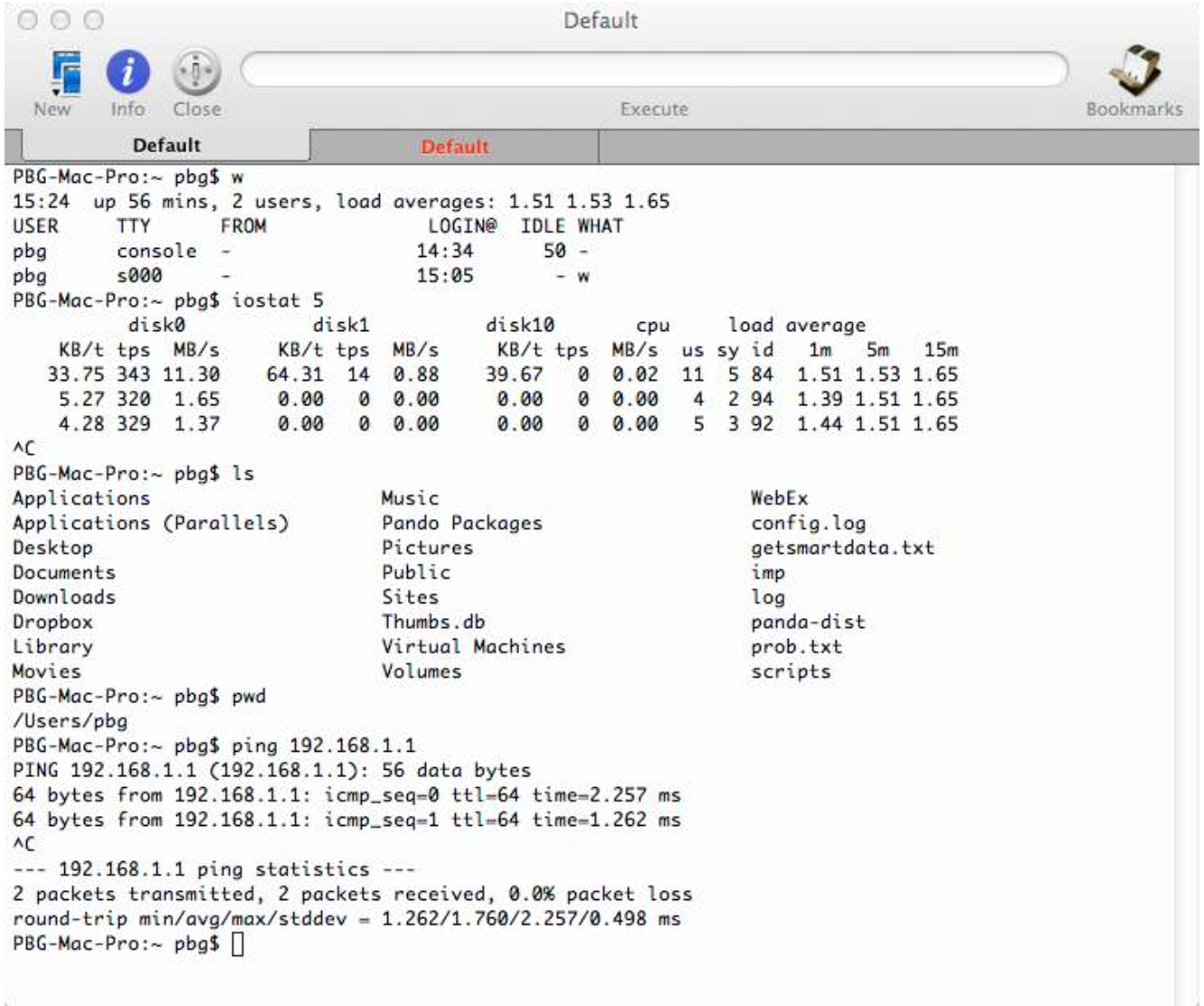
# [User - Operating System] Interface - CLI

- CLI: Command Line Interface (CLI) or command interpreter (shell)
  - in kernel or as a system program,
  - Many flavors
  - fetches a command from user and executes it
    - Command may be **built-in**,
    - Command may be **another program**
- GUI: User-friendly desktop interface
  - Icons represent files, programs, actions, etc.




Many operating systems now include both CLI and GUI interfaces
Linux: command shells available (CLI); KDE as GUI

# Bourne Shell Command Interpreter



# 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 is "Aqua" GUI interface with UNIX kernel underneath and shells available
  - Unix and Linux have CLI with optional GUI interfaces (CDE, KDE, GNOME)

# The Mac OS X GUI

# Touchscreen Interfaces

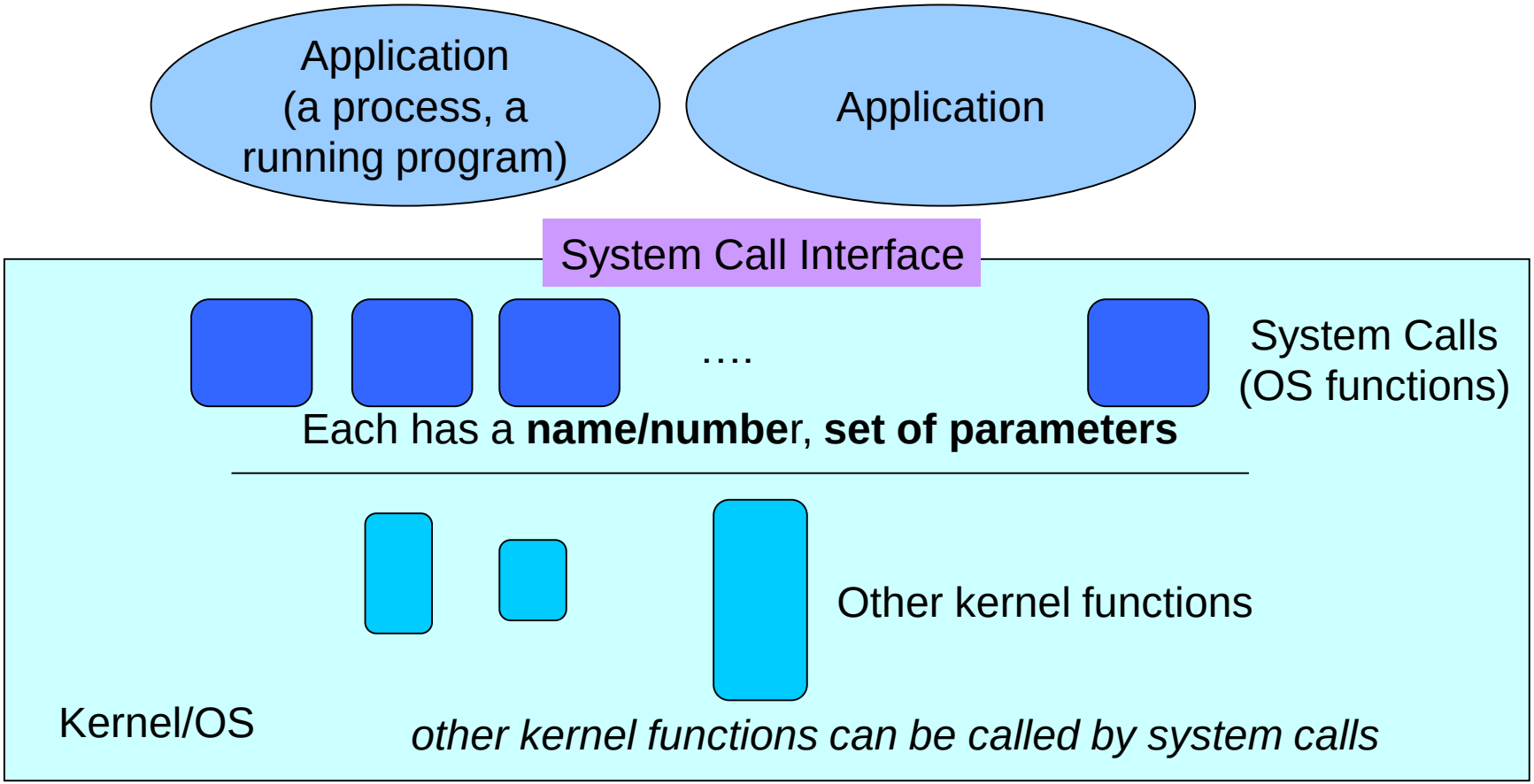
- Touchscreen devices require new interfaces
  - Mouse not possible or not desired
  - Actions and selection based on gestures
  - Virtual keyboard for text entry
- Voice commands.

# System Calls

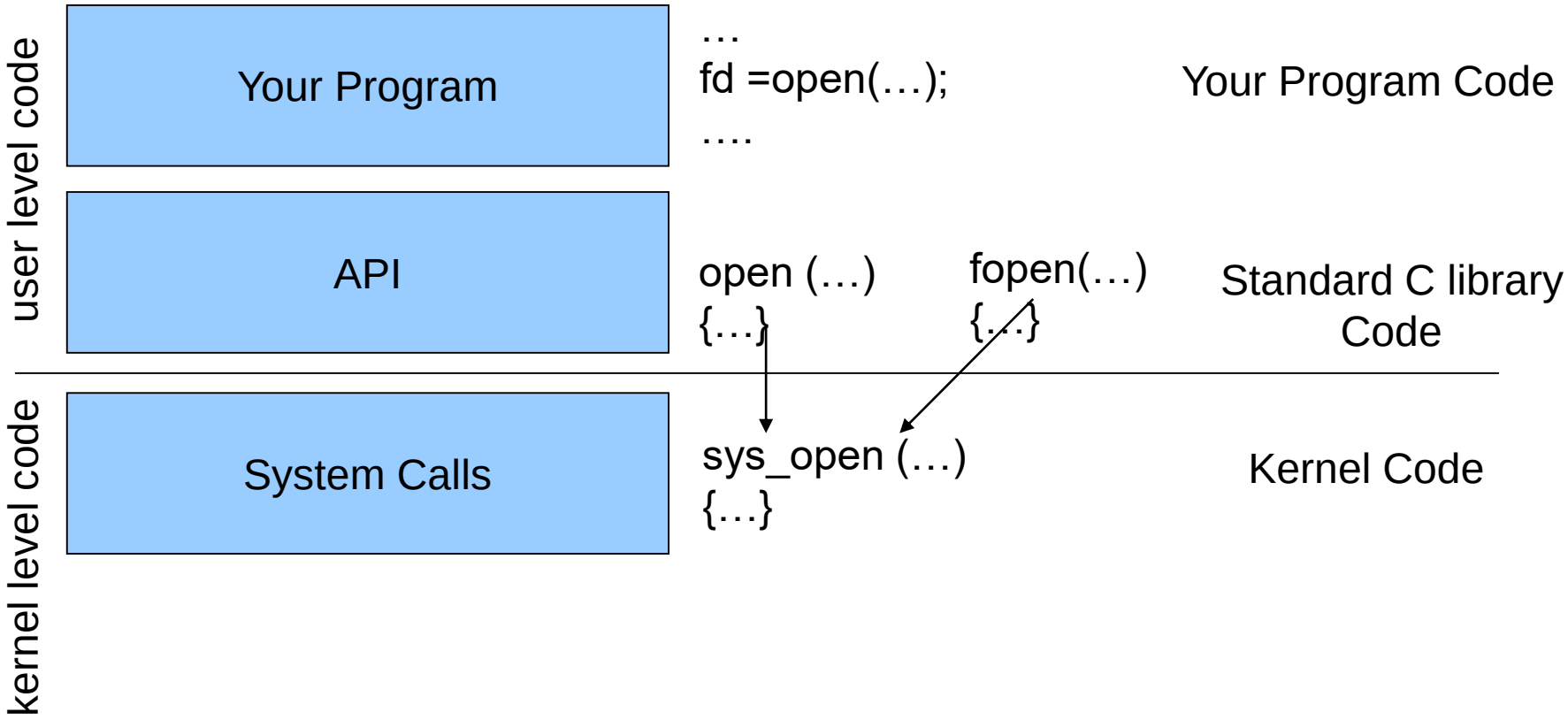
- Programming interface to the services provided by the OS
- Typically written in a high-level language (C or C++)
- Mostly accessed by programs via a high-level **Application**
- **Programming Interface (API)** rather than direct system call use
- Three most common APIs are
- Win32 API for Windows,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

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

# System Calls

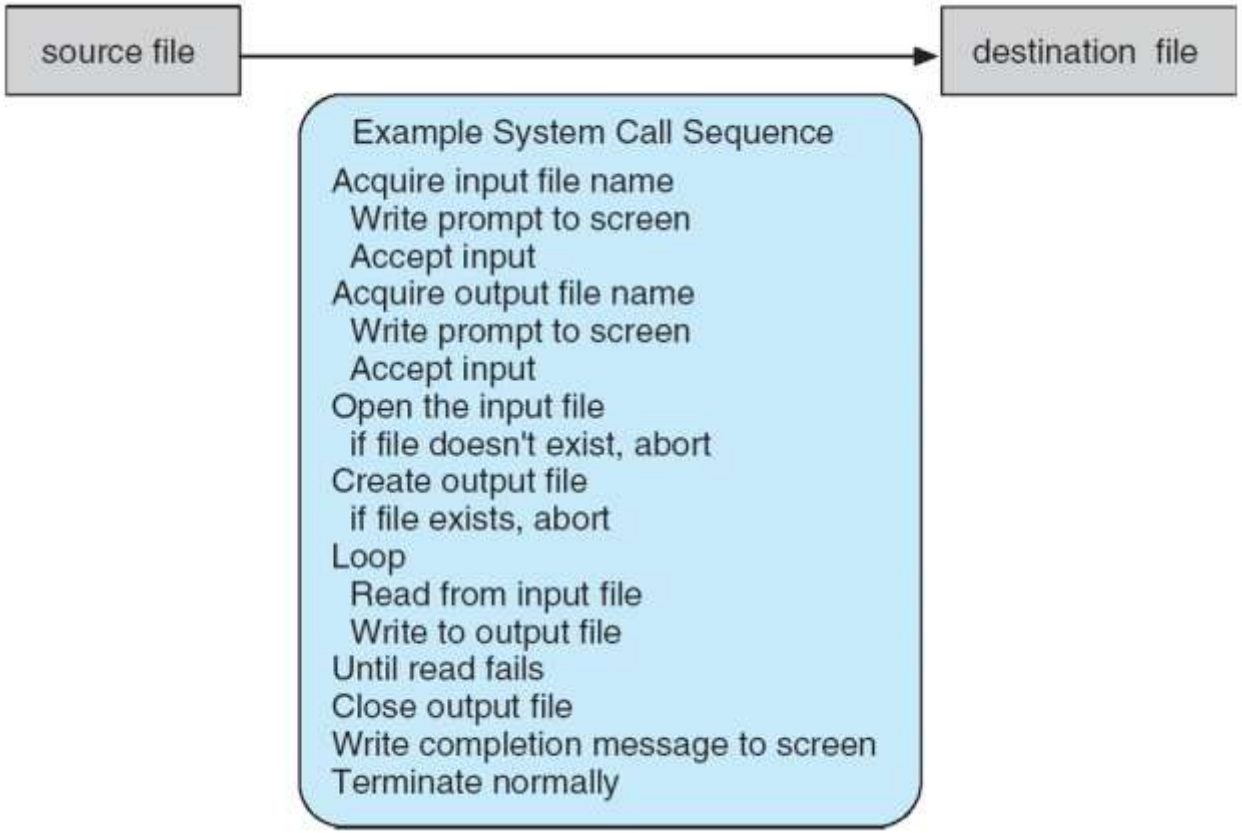


# System Calls



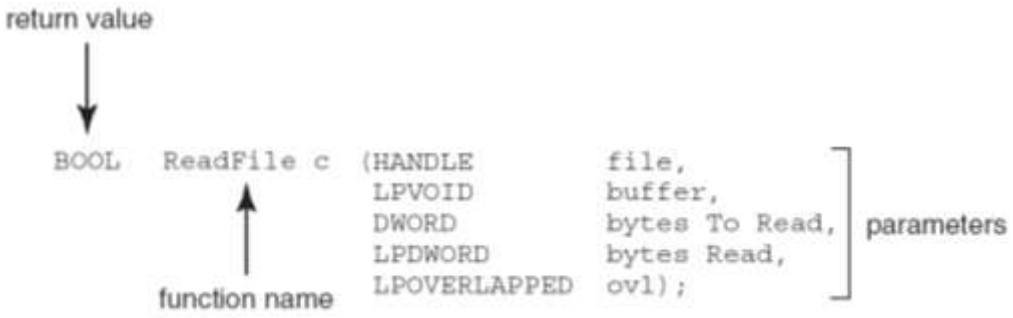
# Example of System Calls

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



# Example of Standard API

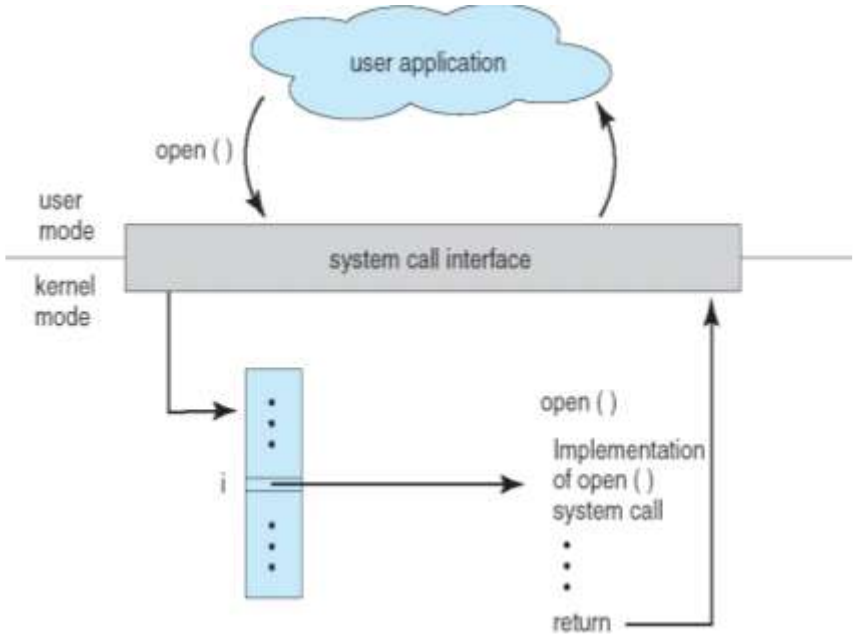
- Consider the ReadFile() function in the Win32 API — a function for reading from a file




- 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 and Calling

- Typically,
  - a **number** associated with each system call
  - Number used as an index to a table: System Call **table**
  - Table keeps **addresses** of system calls (routines)
  - System call runs and returns
- Caller does not know system call implementation
  - Just knows interface



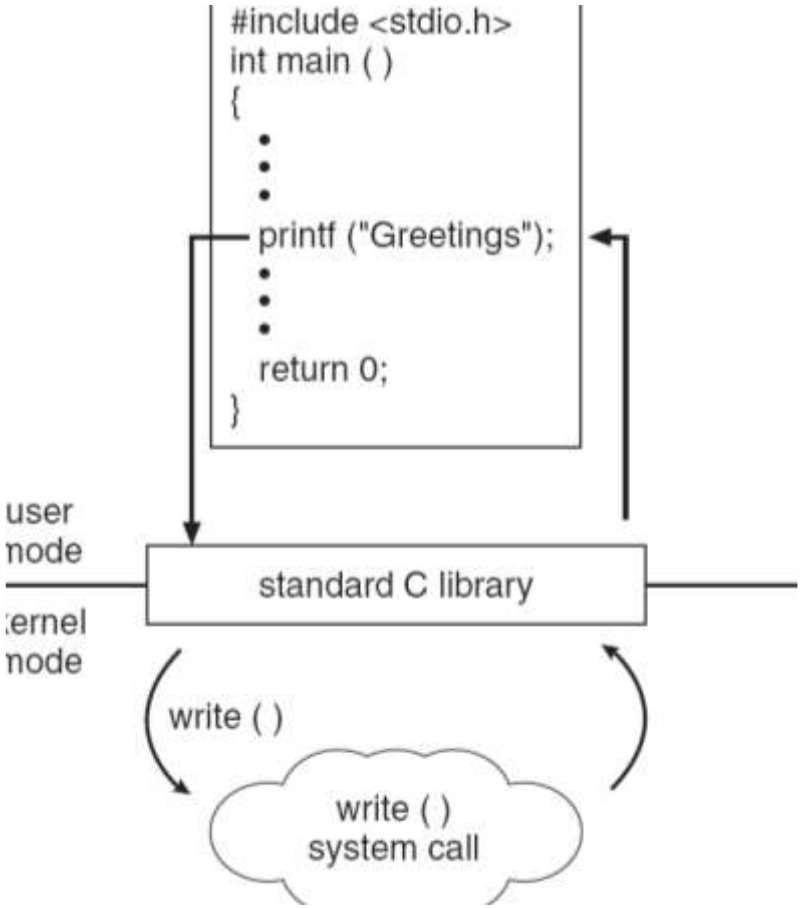
# Linux system calls

around 300-400 system calls

| Number | Generic Name of System Call | Name of Function in Kernel |
|---|---|---|
| 1 | exit | sys_exit |
| 2 | fork | sys_fork |
| 3 | read | sys_read |
| 4 | write | sys_write |
| 5 | open | sys_Open |
| 6 | close | sys_Close |
| … | | |
| 39 | mkdir | sys_Mkdir |
| … | | |

# Standard C Library Example

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

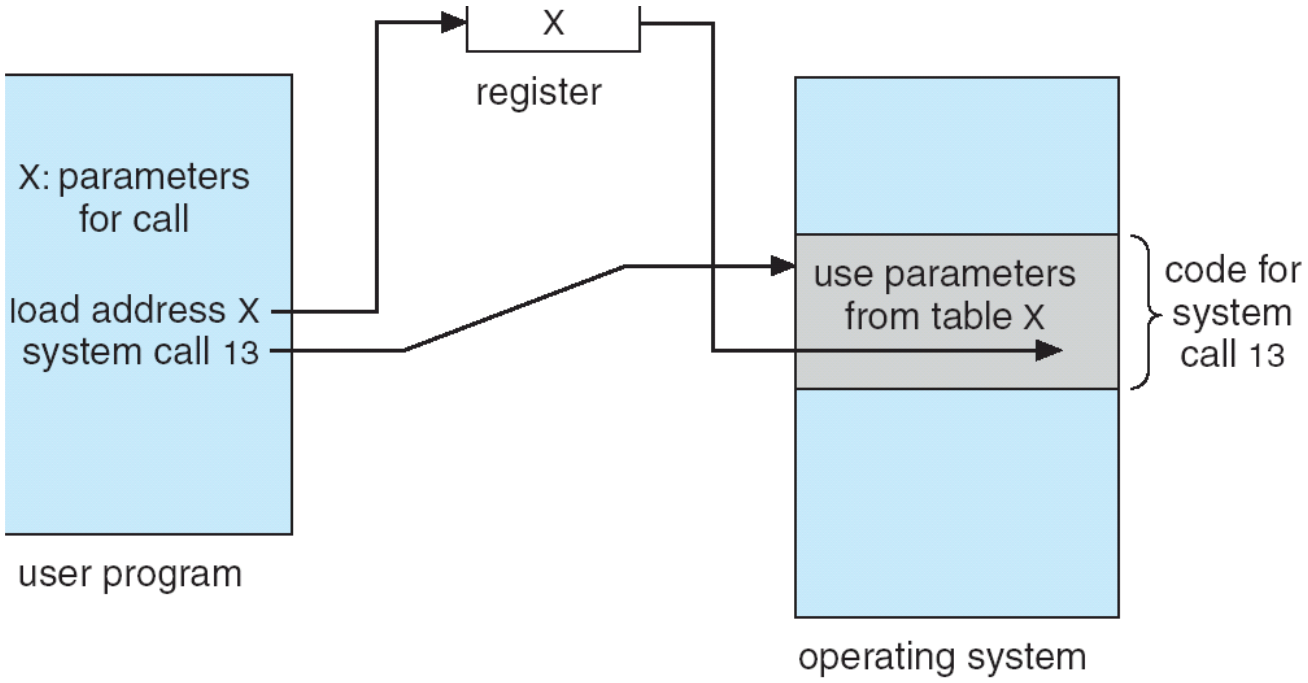


# System Call Parameter Passing

- Often, more information is required than the identity of the 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
  - 1) Simplest: pass the parameters in *registers*
    - In some cases, may be more parameters than registers
  - 2) Parameters stored in a *block,* or table, in memory, and address of block passed as a parameter in a register
  - 3) Parameters placed, or *pushed,* onto the *stack* by the program and *popped* off the stack by the operating system

Last two methods do not limit the number or length of parameters being passed

# Parameter Passing via Table



# Types of System Calls

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

# System Call types Examples of Windows and Unix System Calls

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

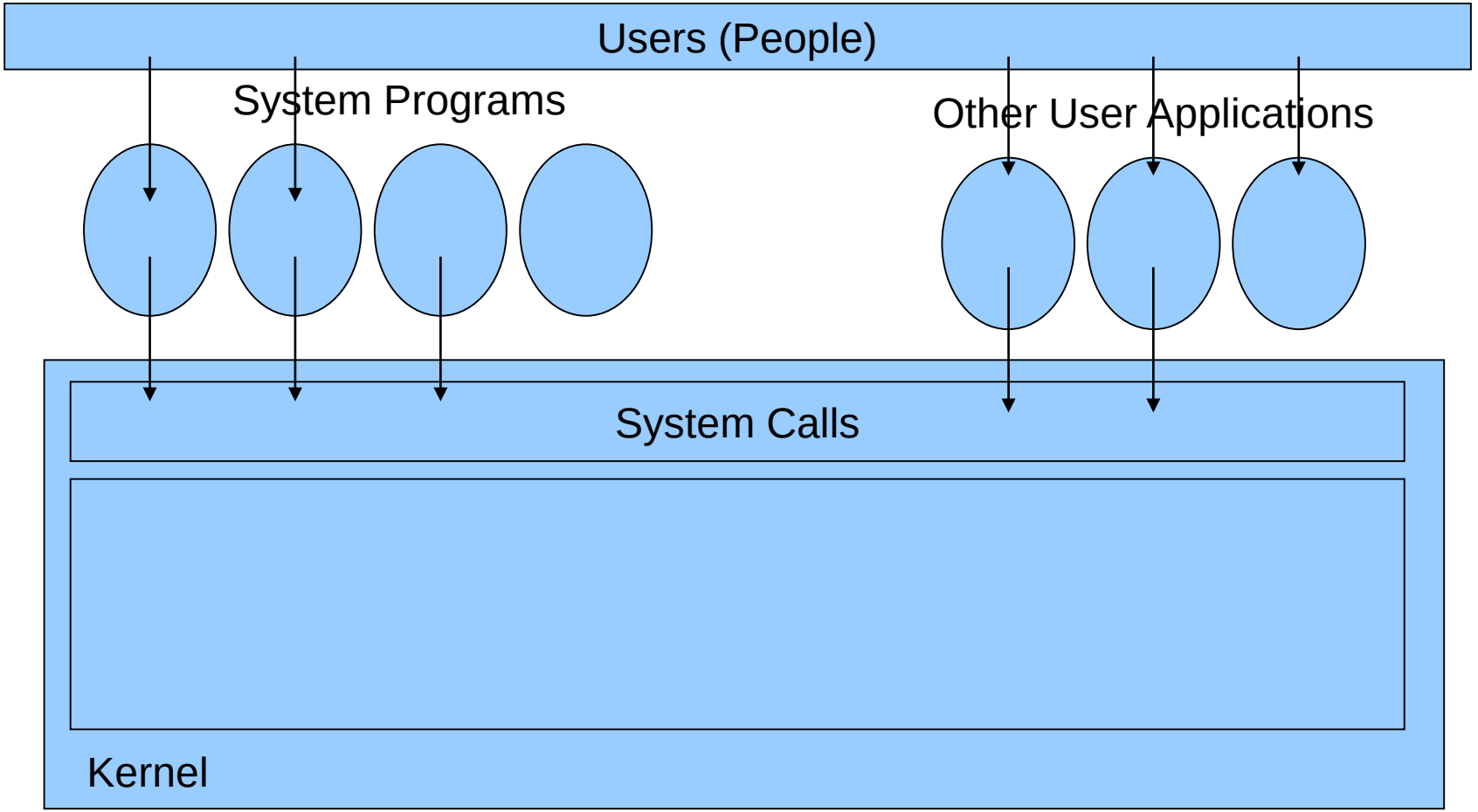
# System Programs

- System programs provide a convenient environment for program development and execution. They can be divided into:
  - File manipulation (create, delete, copy, rename, print, list, …)
  - Status information (date, time, amount of available memory, disk space, who is logged on, …)
  - File modification (text editors, grep, …)
  - Programming language support (compiler, debuggers, …)
  - Program loading and execution (loaders, linkers)
  - Communications (ftp, browsers, ssh, …)
  - Other System Utilities/Applications may come with OS CD (games, math solvers, plotting tools, database systems, spreadsheets, word processors, …)

# System 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
    - create file: simple system program that can just call "create" system call or something similar
    - compiler: complex system program

# System Programs




From OS's view: system+user programs are all applications

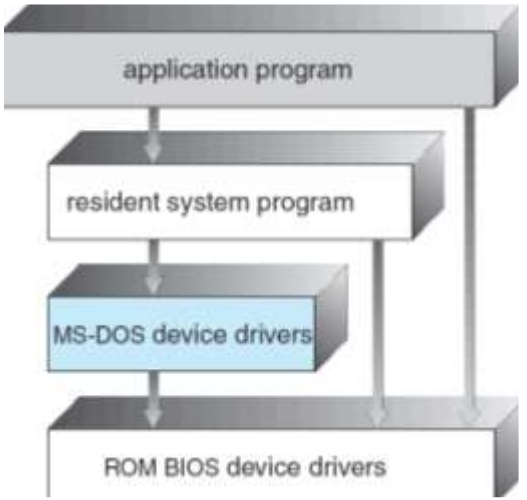
# Structuring OS (Kernel)

# OS Structure

- Simple Structure (MSDOS)
- Layered Approach
- Microkernel Approach
- Modules Approach

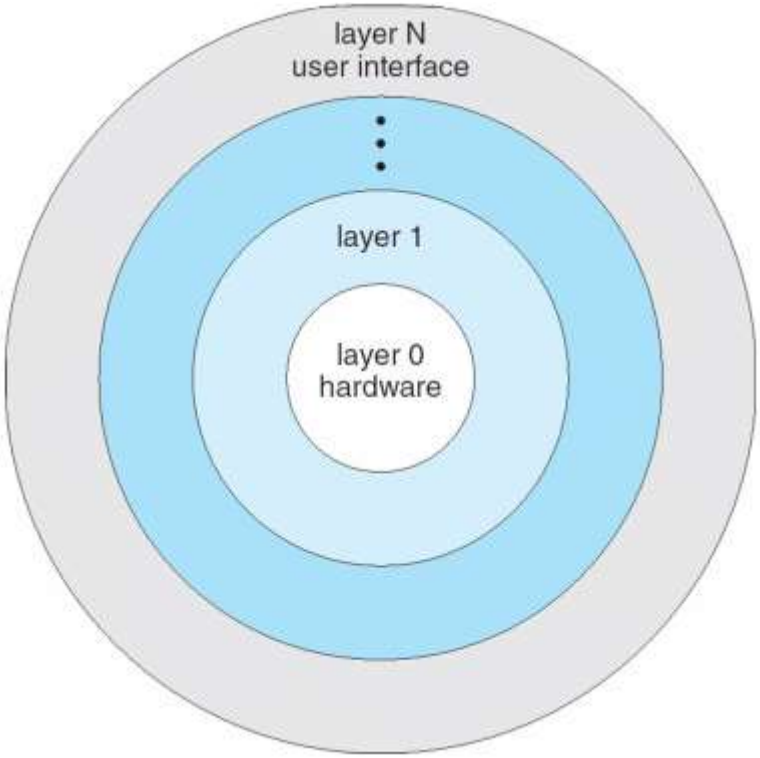
# Simple Structure

- 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



# Layered Approach

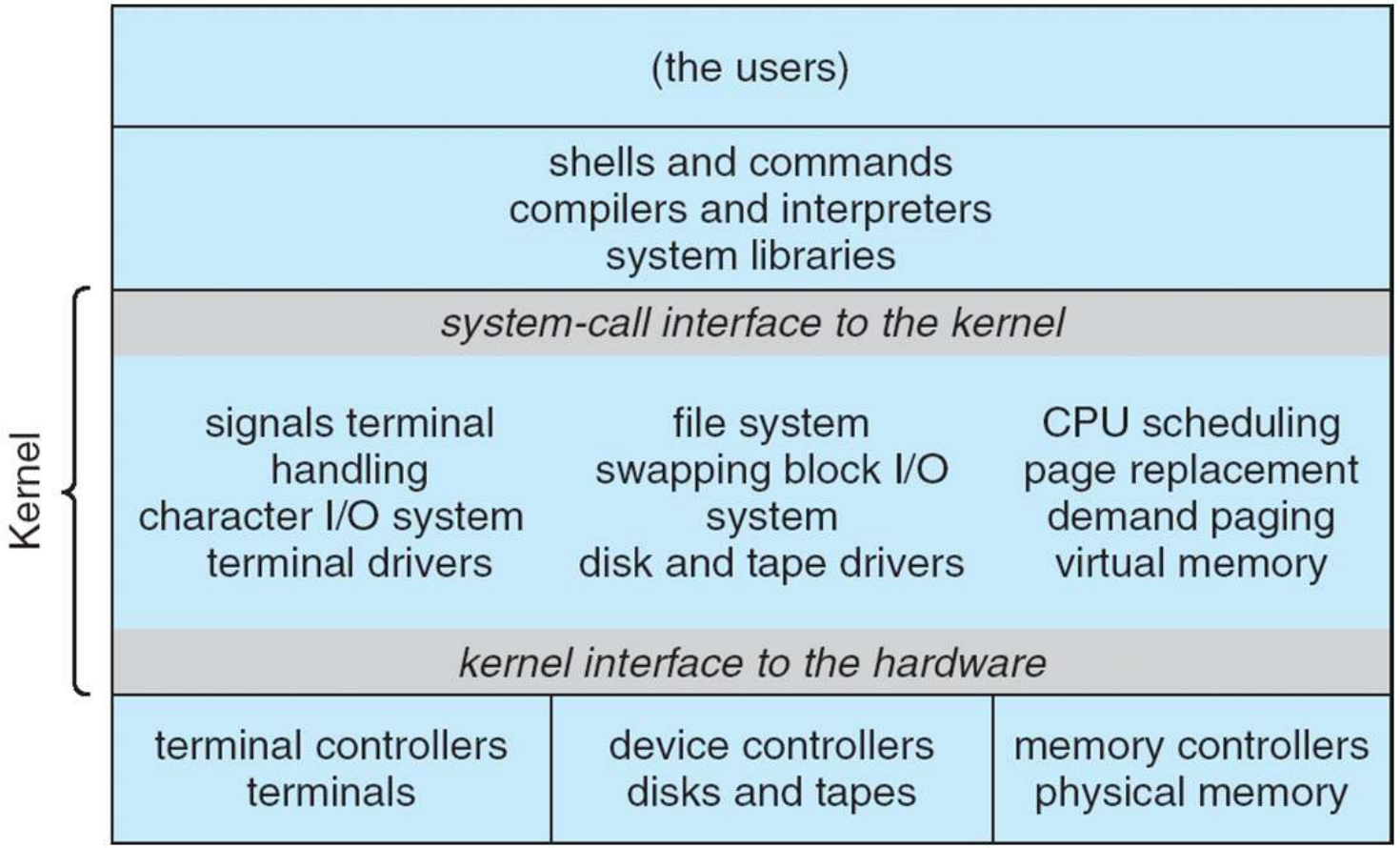
- The operating system is divided into a number of layers (levels), each built on top of lower layers. The bottom layer (layer 0), is the hardware; the highest (layer N) is the user interface.
- With modularity, layers are selected such that each uses functions (operations) and services of only lower-level layers



# 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

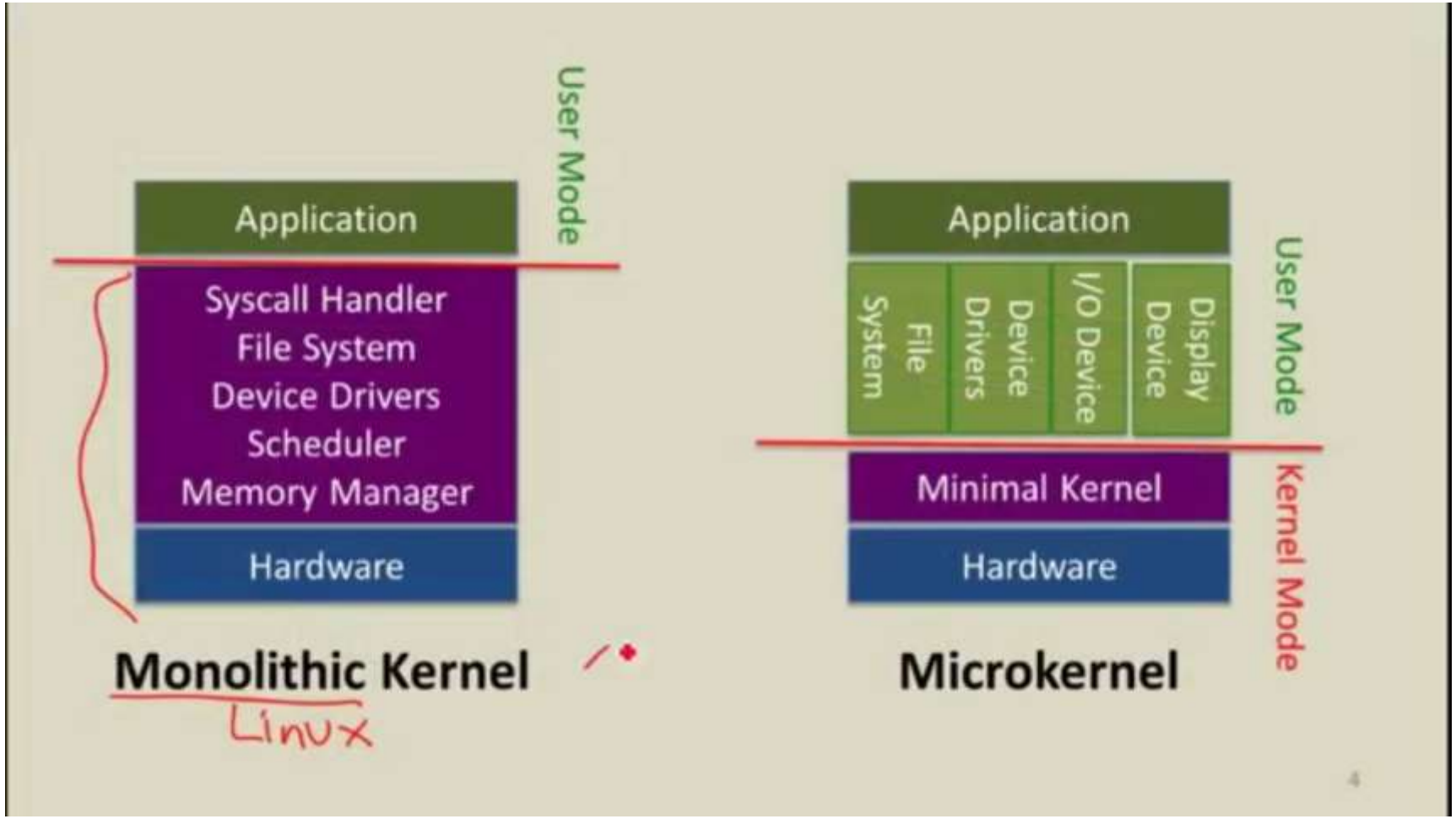
# Traditional UNIX System Structure



# 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

# Microkernel Vs Monolithic Kernel



# Microkernel Vs Monolithic Kernel

1. MICROKERNEL VS MONOLITHIC SYSTEM ~1992

**Most older operating systems are monolithic, that is, the whole operating system is a single a.out file that runs in 'kernel mode.'** This binary contains the process management, memory management, file system and the rest. Examples of such systems are UNIX, MS-DOS, VMS, MVS, OS/360, MULTICS, and many more.

**The alternative is a microkernel-based system, in which most of the OS runs as separate processes, mostly outside the kernel.** They communicate by message passing. **The kernel's job is to handle the message passing, interrupt handling, low-level process management, and possibly the I/O.** Examples of this design are the RC4000, Amoeba, Chorus, Mach, and the not-yet-released Windows/NT.

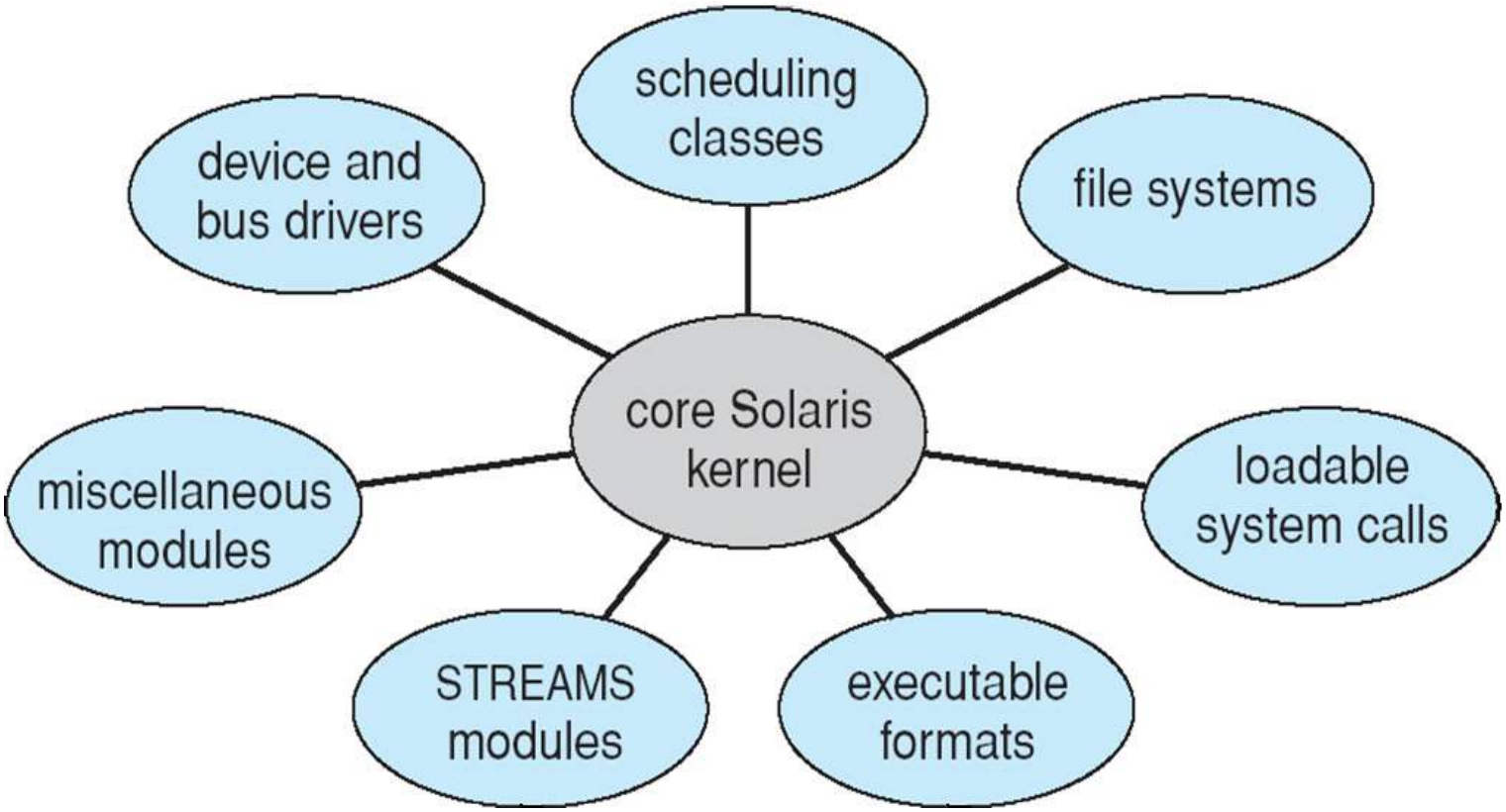
While I could go into a long story here about the relative merits of the two designs, suffice it to say that among the people who actually design operating systems, the debate is essentially over. **Microkernels have won.**

# 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 more flexible

- Linux supports modules

# Solaris Modular Approach

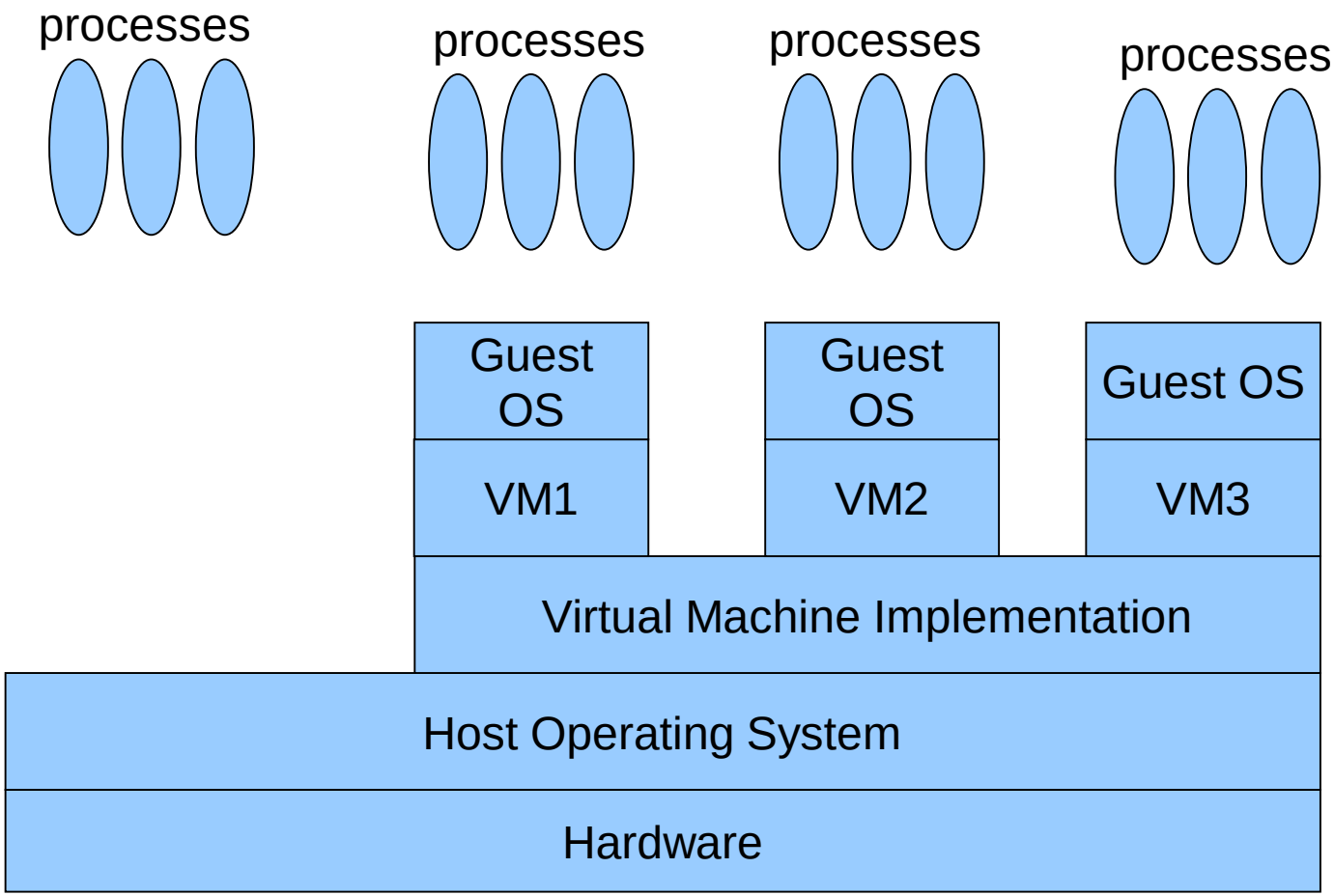


# Additional Study Material

# Virtual Machines

- Hardware is abstracted into several different execution environments
  - Virtual machines
- Each virtual machine provides an interface that is identical to the bare hardware
- A *guest* process/kernel can run on top of a virtual machine.
  - We can run several operating systems on the same *host*.
  - Each virtual machine will run another operating system.

# Virtual Machines



# Examples

- VMware
  - Abstracts Intel X86 hardware
- Java virtual machine
  - Specification of an abstract computer
- .NET Framework

# Operating System Debugging

- Failure analysis
  - Log files
  - Core dump
  - Crash dump

- Performance tuning
  - Monitor system performance
    - Add code to kernel
    - Use system tools like “top”

- DTrace
  - Facility to dynamically adding probes to a running system (both to processes and to the kernel)
  - Probes can be queries using D programming language to obtain info

# Operating System Generation

- Configure the kernel
- Compile the kernel

# System Boot

- Bootstrap program (loader) locates the kernel, loads it and starts the kernel.
  - This can be a two-step procedure.
  - Bootstrap program loads another more complex boot program
  - That boot program loads the kernel
- Then control is given to kernel.
- Kernel starts the environment and makes the computer ready to interact with the user (via a GUI or command shell).
- Details depend on the system