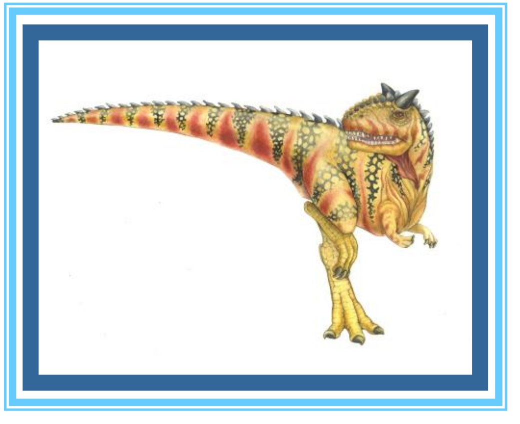
Chapter 2: System Structures



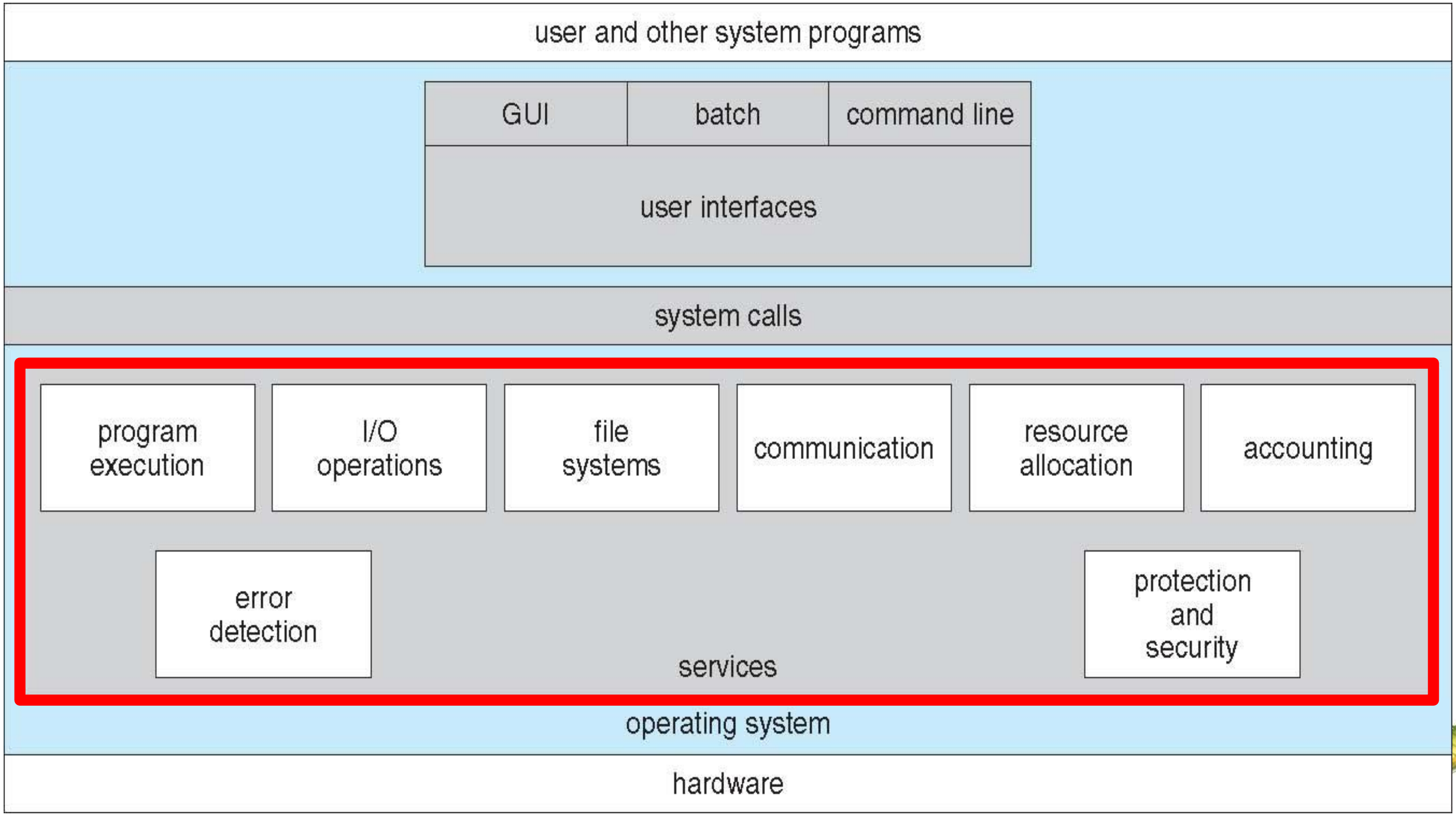
Chapter 2: System Structures

- Operating System Services
  - User Operating System Interface
  - System Calls
- Operating System Design and Implementation

Operating System Services

- Operating systems provide an environment for execution of programs and services to programs and users

A View of Operating System Services



Operating System Services

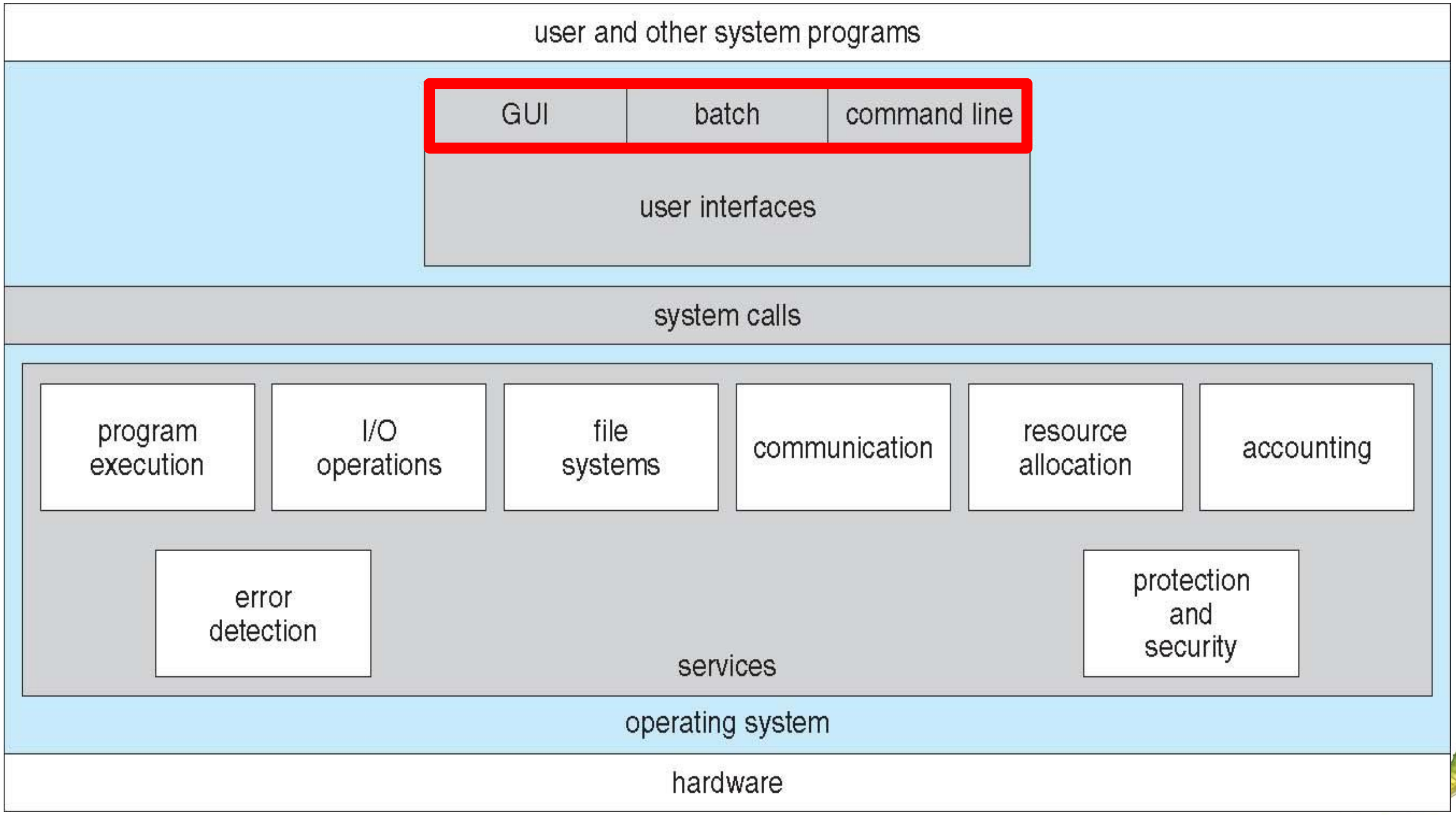
- System call services
  - **Program execution** - The system must be able to load a program into memory and to run that program, end execution, either normally or abnormally (indicating error)
  - **I/O operations** - A running program may require I/O, which may involve a file or an I/O device
  - **File-system manipulation** - 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
    - via shared memory or through message passing
  - **Error detection** – OS needs to be constantly aware of possible errors

Operating System Services (Cont.)

- **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

Operating System Services

- **User interface** - Almost all operating systems have a user interface
  - Varies between **Command-Line (CLI)**, **Graphics User Interface (GUI)**, **Batch**



User Operating System Interface - CLI

- **Command-line interpreter (CLI)**
  - allows direct command entry
  - Sometimes implemented in kernel, sometimes by systems program
  - Sometimes multiple CLIs are implemented – **shells**
  - Primarily fetches a command from user and executes it
    - commands built-in, or just names of programs

      If the latter, adding new features doesn't require shell modification

```
PBG-Mac-Pro:~ pbg$ ls
Applications                    Music               WebEx
Applications (Parallels)        Pando Packages      config.log
Dropbox                         Thumbs.db           panda-dist
PBG-Mac-Pro:~ pbg$ pwd
/Users/pbg
PBG-Mac-Pro:~ pbg$ ping 192.168.1.1
PING 192.168.1.1 (192.168.1.1): 56 data bytes
64 bytes from 192.168.1.1: icmp_seq=0 ttl=64 time=2.257 ms
64 bytes from 192.168.1.1: icmp_seq=1 ttl=64 time=1.262 ms
^C
PBG-Mac-Pro:~ pbg$
```

Bourne Shell Command Interpreter

User Operating System Interface - GUI

- **Graphic User Interface (GUI)**
  - 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)

User Operating System Interface - Touchscreen Interfaces

- **Touchscreen Interface**

  Touchscreen devices require new interfaces because

  - mouse not possible or not desired
  - Actions and selection based on gestures
  - Virtual keyboard for text entry

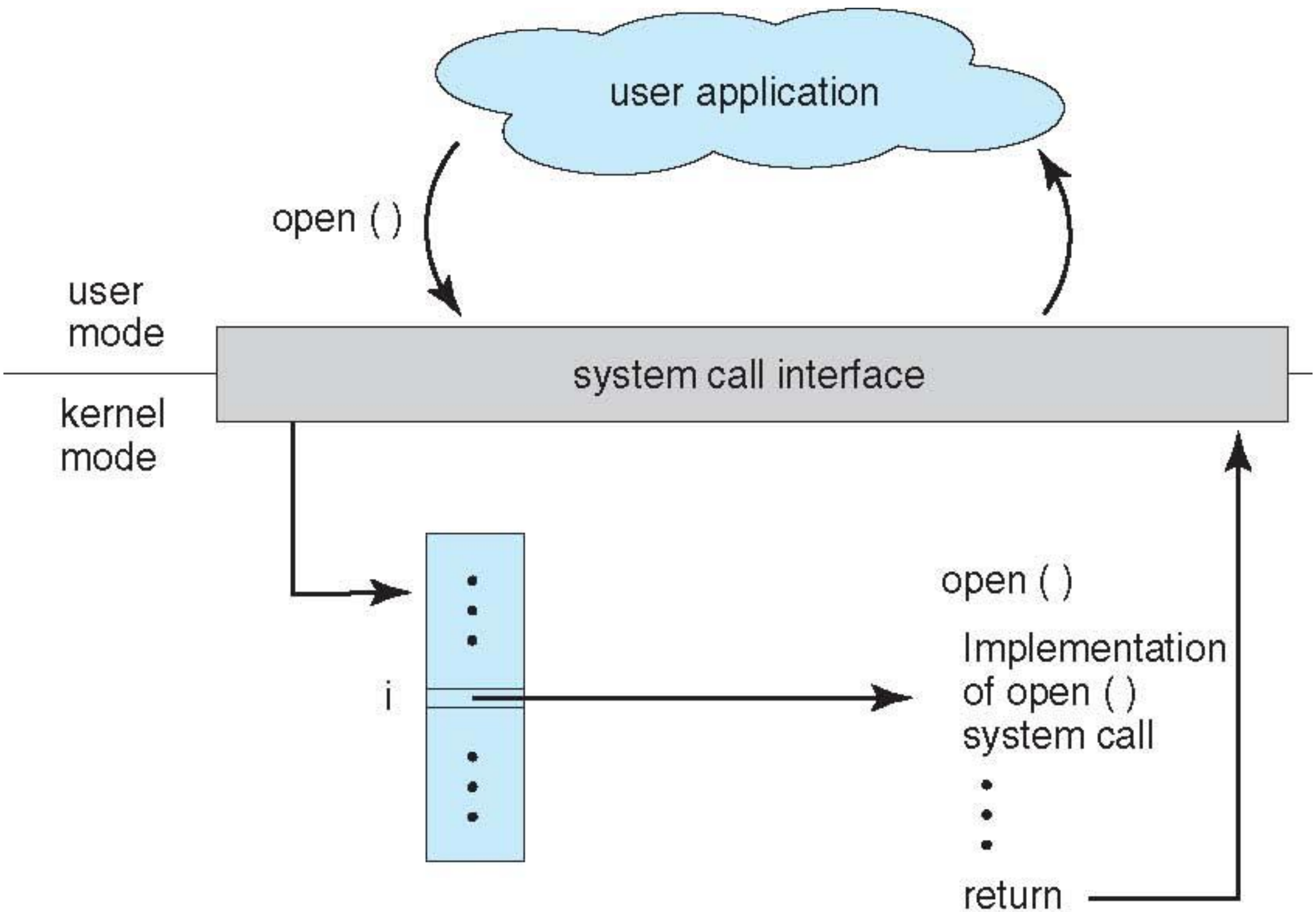
Chapter 2: System Structures

- Operating System Services
  - User Operating System Interface
  - System Calls
- Operating System Design and Implementation

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.
  - **Win32 API** for Windows,
  - **POSIX API** for POSIX-based systems (including virtually all versions of UNIX, Linux, and Mac OS X)
  - **Java API** for the Java virtual machine (JVM)
- The API invokes intended system call in OS kernel and returns status of the system call and any return values
  - Most details of system calls hidden from programmer by API
  - Just needs to obey API and understand what OS will do as a result call
- Typically, a number is associated with each system call and maintains a table indexed according to these numbers

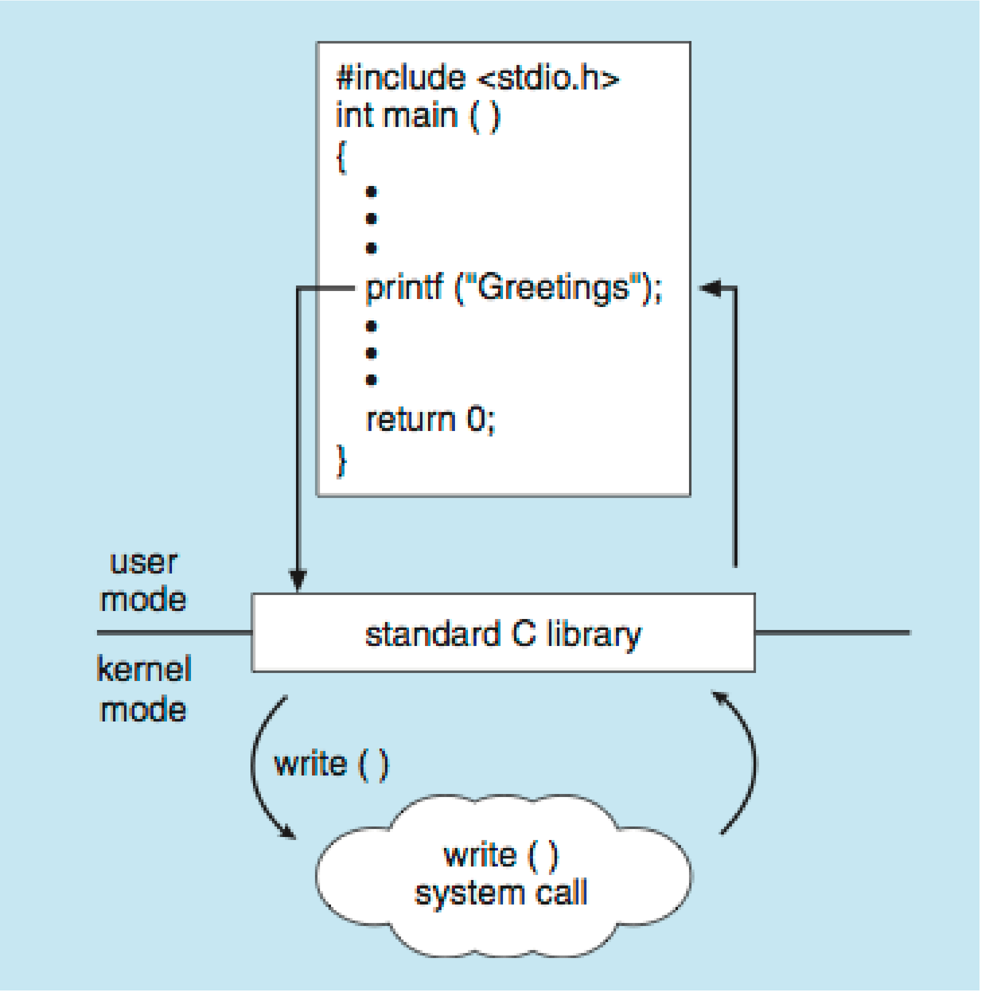
API – System Call – OS Relationship



Win32 API
POSIX API
Java API

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
  - required 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* ***pushed****, onto the* ***stack*** by the program and **popped** off the stack by the operating system
- The block or stack methods do not limit the number or length of parameters being passed

Types of System Calls

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

Types of System Calls

- **File management**
  - create file, delete file
  - open, close file
  - read, write, reposition
  - get and set file attributes
- **Device management**
  - request device, release device
  - read, write, reposition
  - get device attributes, set device attributes
  - logically attach or detach devices
- **Information maintenance**
  - get time or date, set time or date
  - get system data, set system data
  - get and set process, file, or device attributes
- **Protection**
  - Control access to resources
  - Get and set permissions
  - Allow and deny user access

Types of System Calls (Cont.)

- **Communications**
  - create, delete communication connection
  - Send, receive messages
    - **message passing model**
      - Information is exchanged by OS.
      - Useful when smaller numbers of data need to be exchanged.
      - Easier to implement than shared memory.
    - **shared-memory model**
      - exchange info. by reading and writing data in shared areas.
      - Fast speed and convenience of communications
      - The protection and synchronization problems in the shared area. (more complex to implement)
  - transfer status information

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()`

Chapter 2: System Structures

- Operating System Services
  - User Operating System Interface
  - System Calls
- Operating System Design and Implementation

Operating System Design and Implementation

- Important principle to separate

  Policy: ***What*** will be done?
  Mechanism: ***How*** to do it?

- Policies decide what will be done, mechanisms determine how to do something,
- The separation of policy from mechanism is a very important principle, it allows maximum flexibility if policy decisions are to be changed later

- Specifying and designing OS is highly creative task of **software engineering**

Implementation

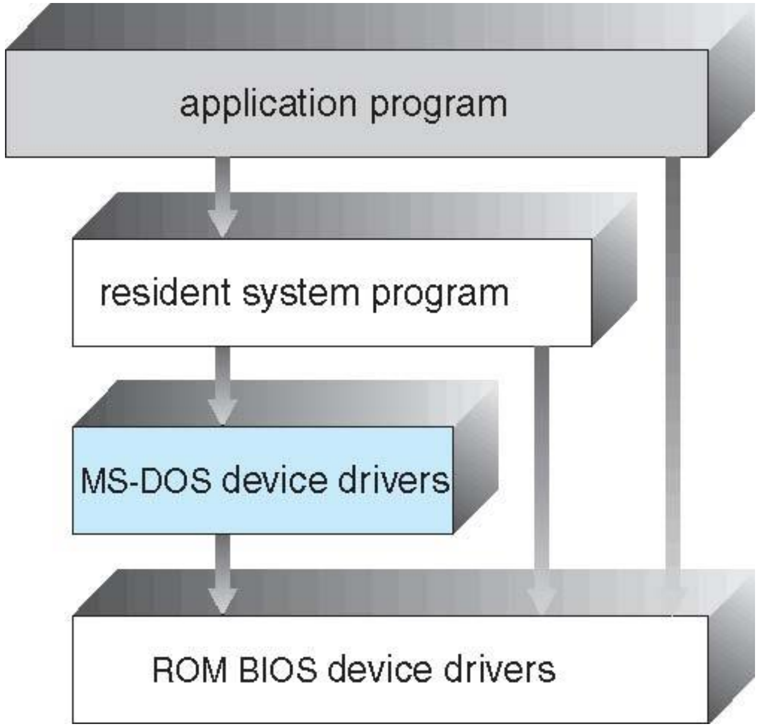
- Much variation
  - Early OSes in assembly language
  - Now C, C++
- Actually usually a mix of languages
  - Lowest levels in assembly
  - Main body in C
  - Systems programs in C, C++, scripting languages like PERL, Python, shell scripts
- More high-level language easier to **port** to other hardware
  - But running slower
- General-purpose OS is a very large program - various ways to structure one.

Operating System Structure

- Various ways to structure a general-purpose OS
  - Simple structure – MS-DOS
  - More complex -- UNIX
  - Layered – an abstraction
  - Microkernel – Mach
  - Module
  - Hybrid

Operating System Structure – Simple Structure

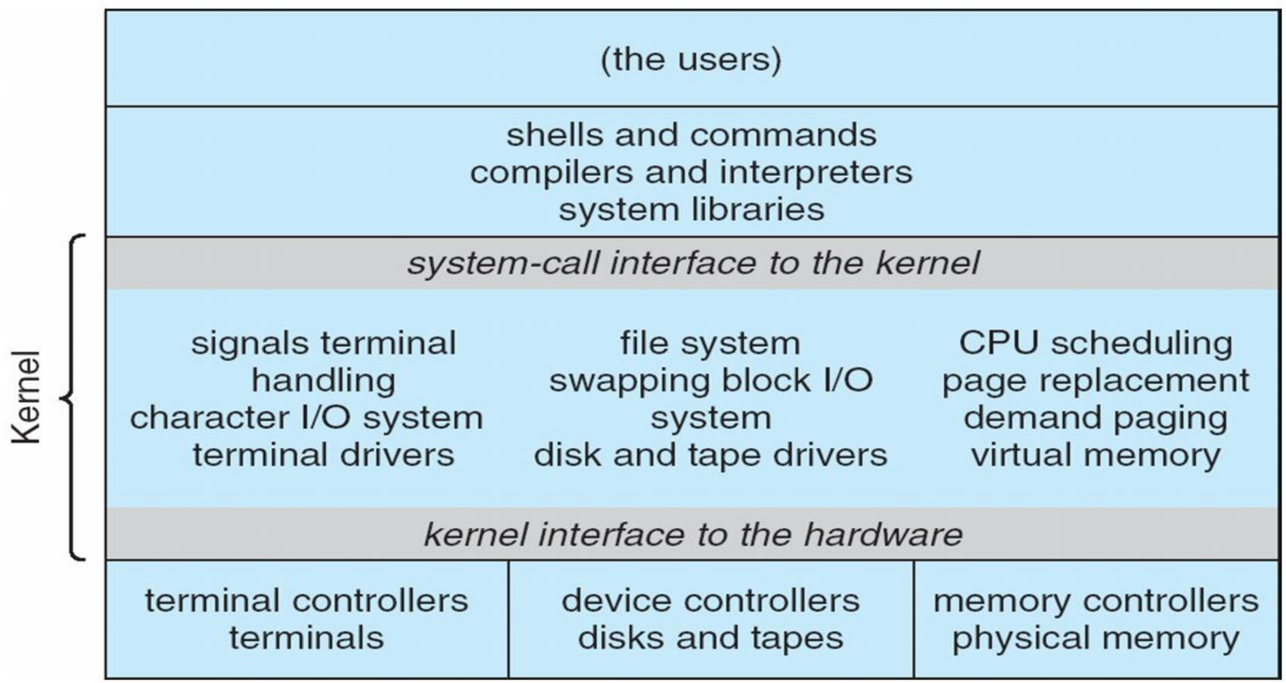
- I.e. MS-DOS – written to provide the most functionality in the least space
  - Compact and efficient
  - Not divided into modules
    - Although MS-DOS has some structure, its interfaces and levels of functionality are not well separated
  - Applications can to access basic I/O routines to write directly to devices → vulnerable to malicious programs and cause the system to crash



Non Simple Structure -- UNIX

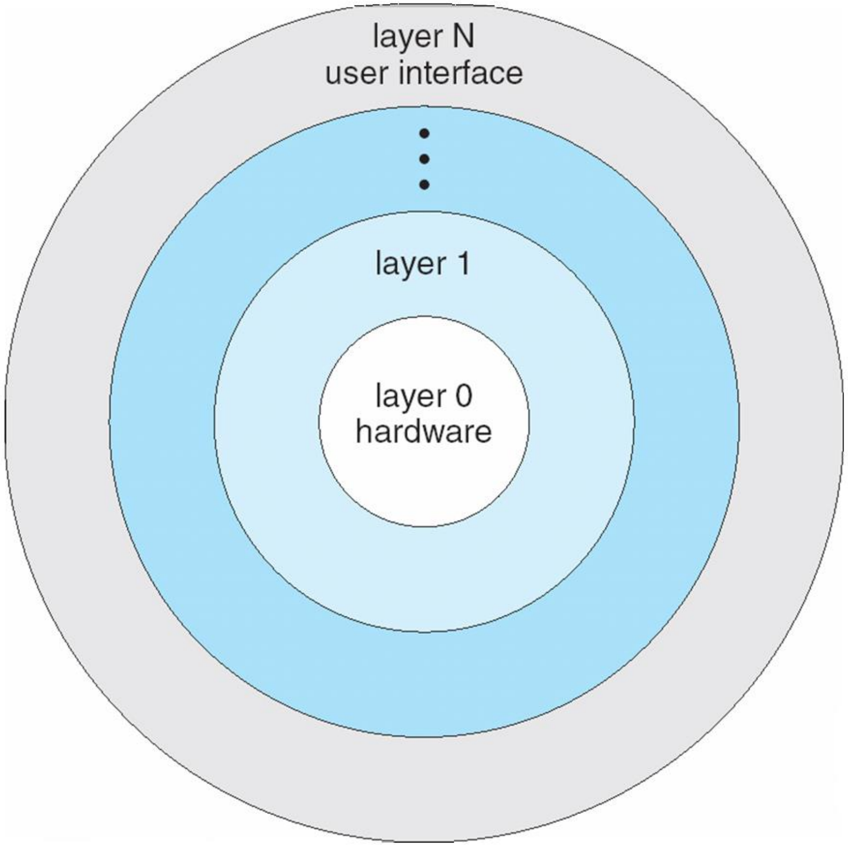
- UNIX – the original UNIX operating system had limited structuring.
- The UNIX OS consists of two separable parts
  - Systems programs
  - The kernel: Consists of everything below system-call interface and above physical hardware

- Beyond simple but not fully layered
- Difficult to implement and maintain



Operating System Structure – Layered Approach

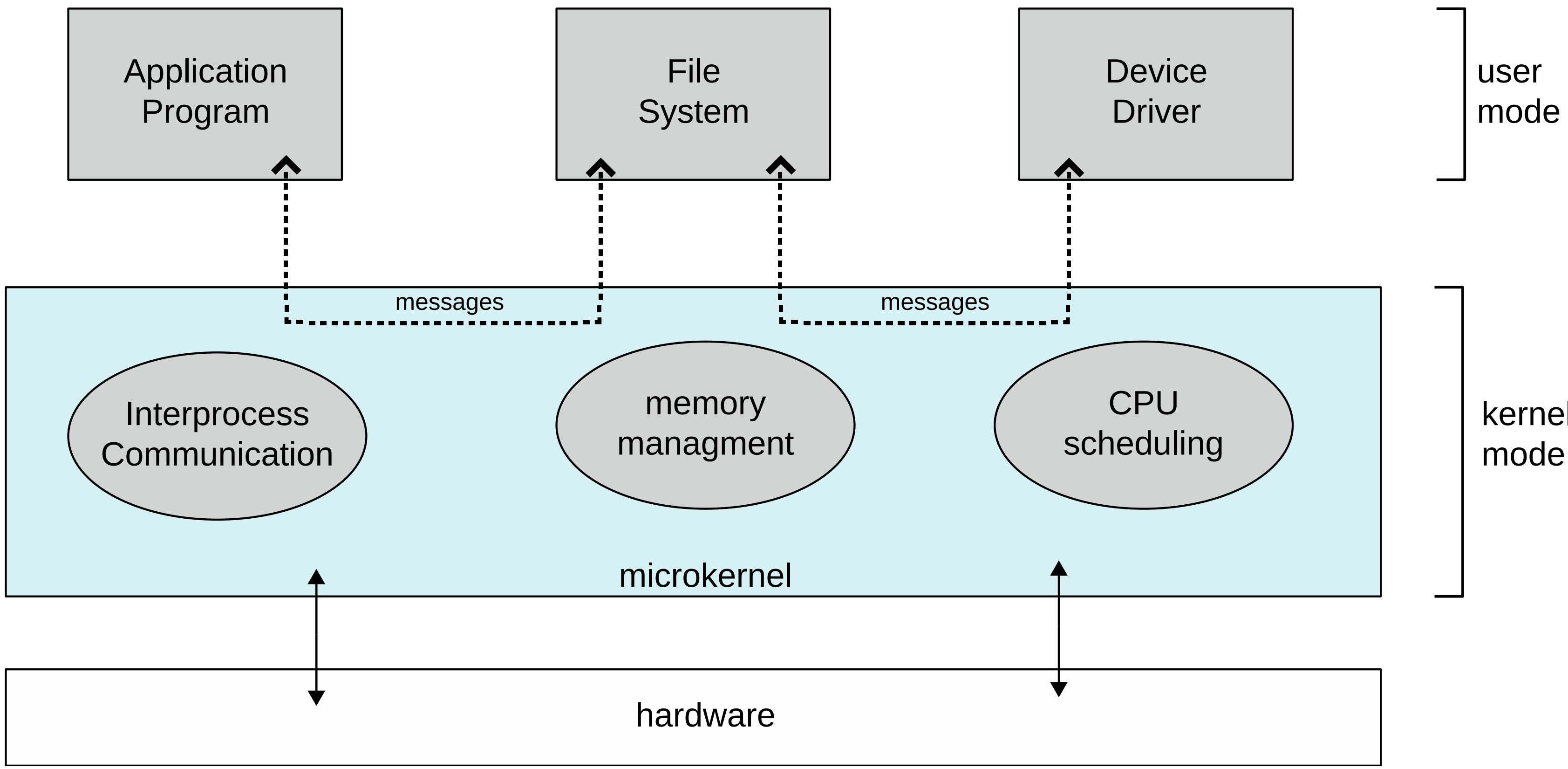
- The operating system is divided into a number of layers (levels), each built on top of lower layers. The bottom layer (layer 0), is the hardware; the highest (layer N) is the user interface.
- With modularity, layers are selected such that each uses functions and services of only lower-level layers
- Advantage: easy to debug
- Not easy to appropriately define layers and the layer approach is less efficient sometimes



Operating System Structure – Microkernel

- Moves as much from the kernel into user space
- **Mach:** example of **microkernel**
  - Only *CPU scheduling*, *memory management* and *interprocess communication* are in kernel space
  - Mac OS X kernel (Darwin) partly based on Mach
- 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 to kernel space communication

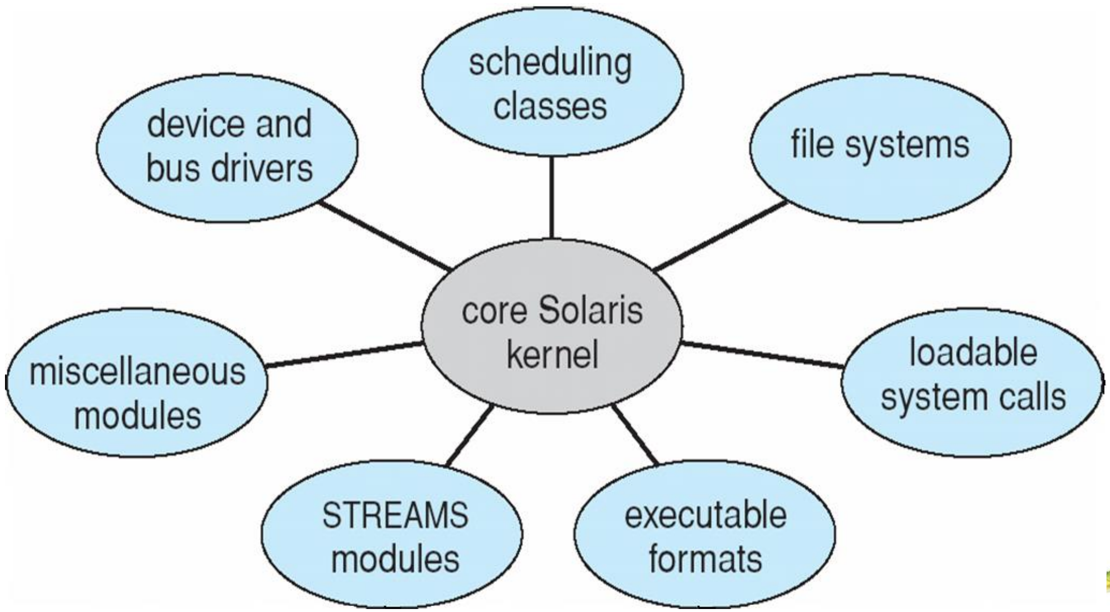
Operating System Structure – Microkernel



Operating System Structure – Modules

- Most modern operating systems implement **loadable 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
  - Solaris

Solaris loadable modules



Operating System Structure – Hybrid Systems

- Most modern operating systems actually not one pure model
- Hybrid combines multiple approaches to address performance, security, usability needs
  - **Linux** and **Solaris** kernels in kernel address space, so monolithic, plus modular for dynamic loading of functionality
  - **Windows** mostly monolithic, plus microkernel and providing separate subsystems that run as user-mode process
  - **Apple Mac OS X** hybrid, layered, **Aqua** UI plus **Cocoa** programming environment. Below is kernel consisting of Mach microkernel and BSD Unix parts, plus I/O kit and dynamically loadable modules (called kernel extensions)

Hybrid System Example – Mac OS X