System Structures

Moumita Patra

July-November 2019

Ref: Galvin, Gagne

Objectives

- To describe the services an OS provides to users, processes, and other systems
- To discuss the various ways of structuring an OS
- How do systems boot

Operating System Services

- An OS provides an environment for the execution of programs
- Set of OS services that are helpful to the user:

➔ User Interface

➔ Program execution- System must be able to load a program into memory and to run that program.

➔ I/O operations- A running program may require I/O which may involve a file or an I/O device

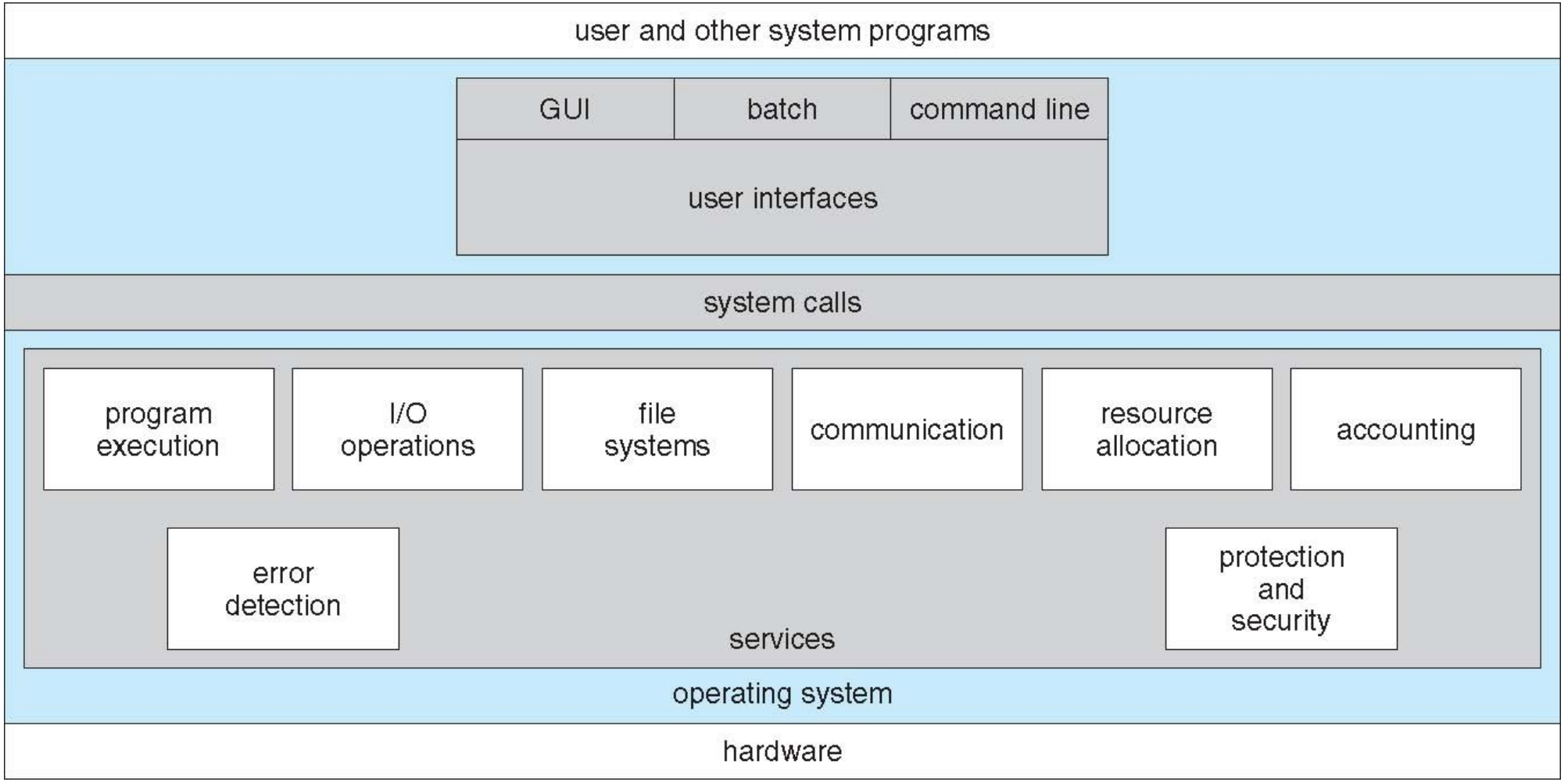
- File-system manipulation
- Communications- Proceses may exhange information, share them.
- Error detection

Operating System Services

Set of OS functions for ensuring efficient operation of the system:

- Resource allocation- When multiple users or multiple jobs are running concurrently
- Accounting- to keep track of which users use what and how much of resources
- Protection and security

Operating System Services

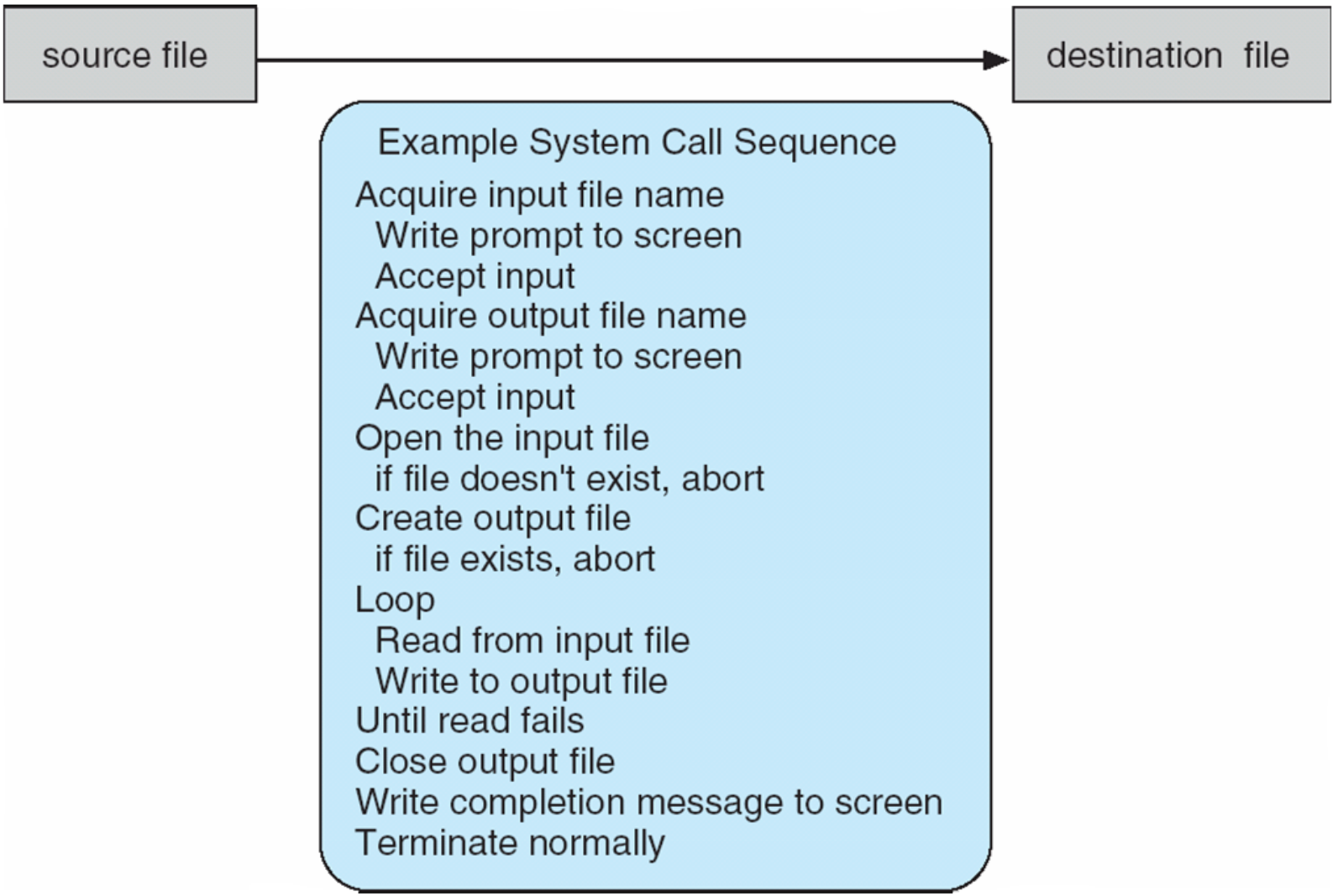


System Calls

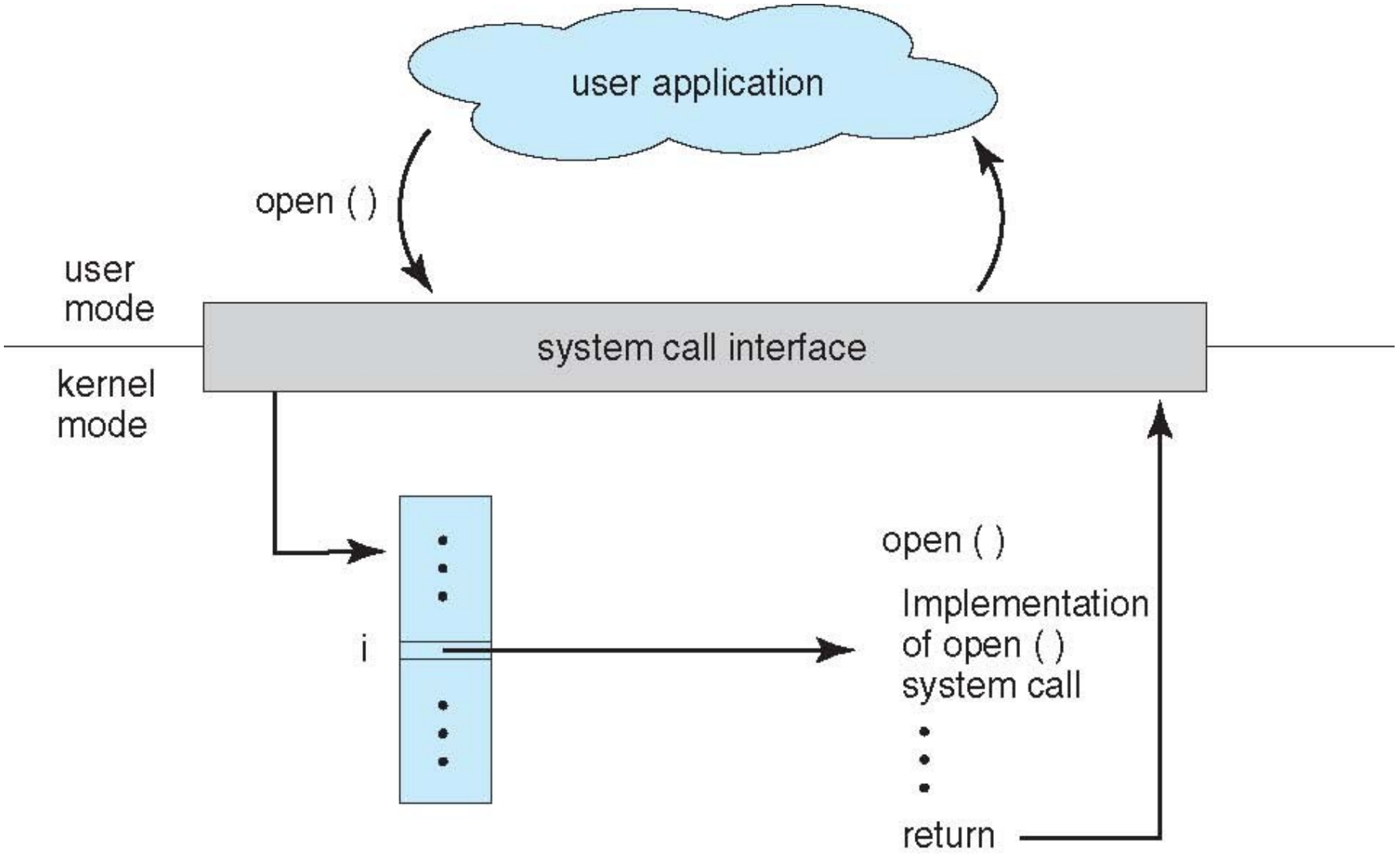
- Programming interface to the services provided by the OS.
- Typically written in a high-level language (C or C++)
- Mostly accessed by programs via a high-level Application Programming Interface (API).
- **Application Programming Interface-** Set of subroutine definitions, protocols, and tools for communication between various software components.

Example of System Calls

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



API-SystemCall- OS Relationship



System Call Implementation

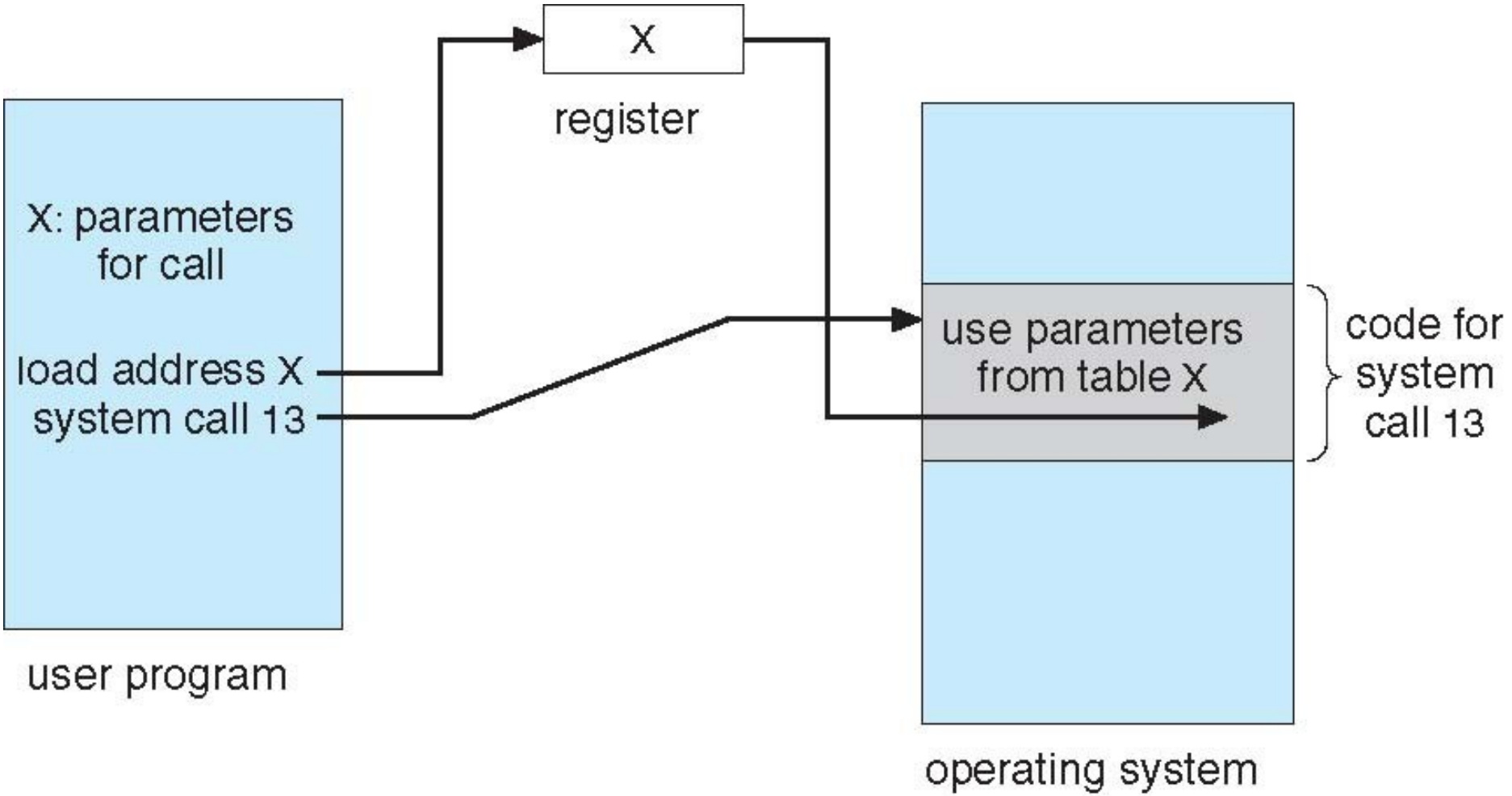
- A number is associated with each system call
- System call interface (SCI) maintains a table indexed according to these numbers
- SCI invokes the intended system call in OS kernel and returns status of the system call and any return values

System Call Parameter Passing

What if more information is required than simply identity of desired system call?

- Three general methods used to pass parameters to the OS
  - ➔ Pass the parameters in registers (simplest)
  - ➔ Parameters stored in a block, or table, in memory, and address of block passed as a parameter in a register
  - ➔ Parameters placed, or pushed, onto the stack by the program and popped off the stack by the OS

Parameters Passing via Table



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
- File management
  - Create file, delete file
  - Open, close file
  - Read, write, reposition
  - Get file attributes, 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 process, file, or device attributes
  - Set process, file, or device attributes

Types of System Calls

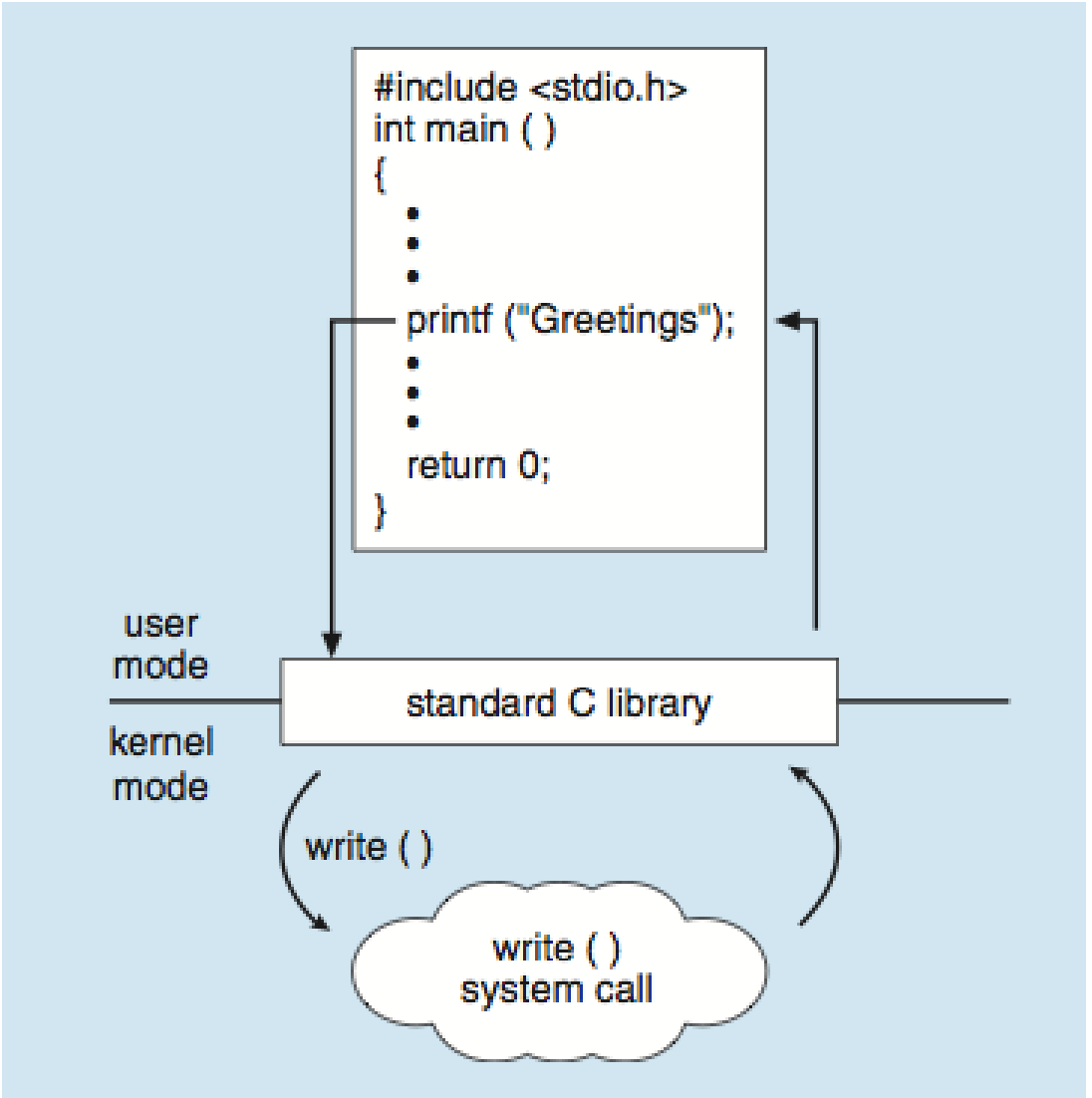
- Communications
  - Create, delete communication connection
  - Send, receive messages
  - Transfer status information
  - Attach or detach remote devices

Examples of Wndows 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()

Standard C Library Example

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



System Programs

- System programs provide a convenient environment for program development and execution.
- They are divided into following categories:

- File management
- Status information
- File modification
- Programming language support
- Program loading and execution
- Communications
- Background services

Operating System Design and Implementation

Design Goals:

- User goals- The system should be convenient to use, easy to learn and to use, reliable, safe, and fast
- System goals- The system should be easy to design, implement and maintain and it should be flexible, reliable, error-free, and efficient

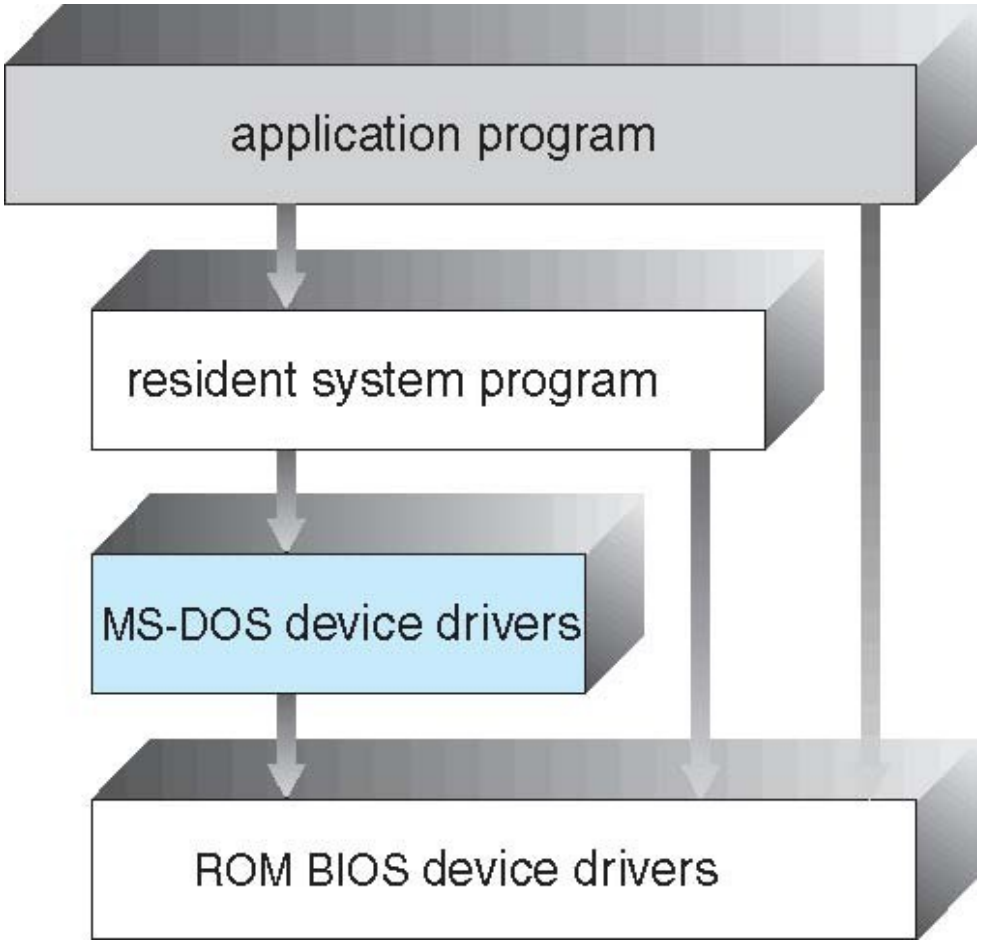
Implementation

- Early operating systems in assembly language
- Usually a mix of languages
- Disadvantage of implementing an OS in a higher level language are reduced speed and increased storage requirements

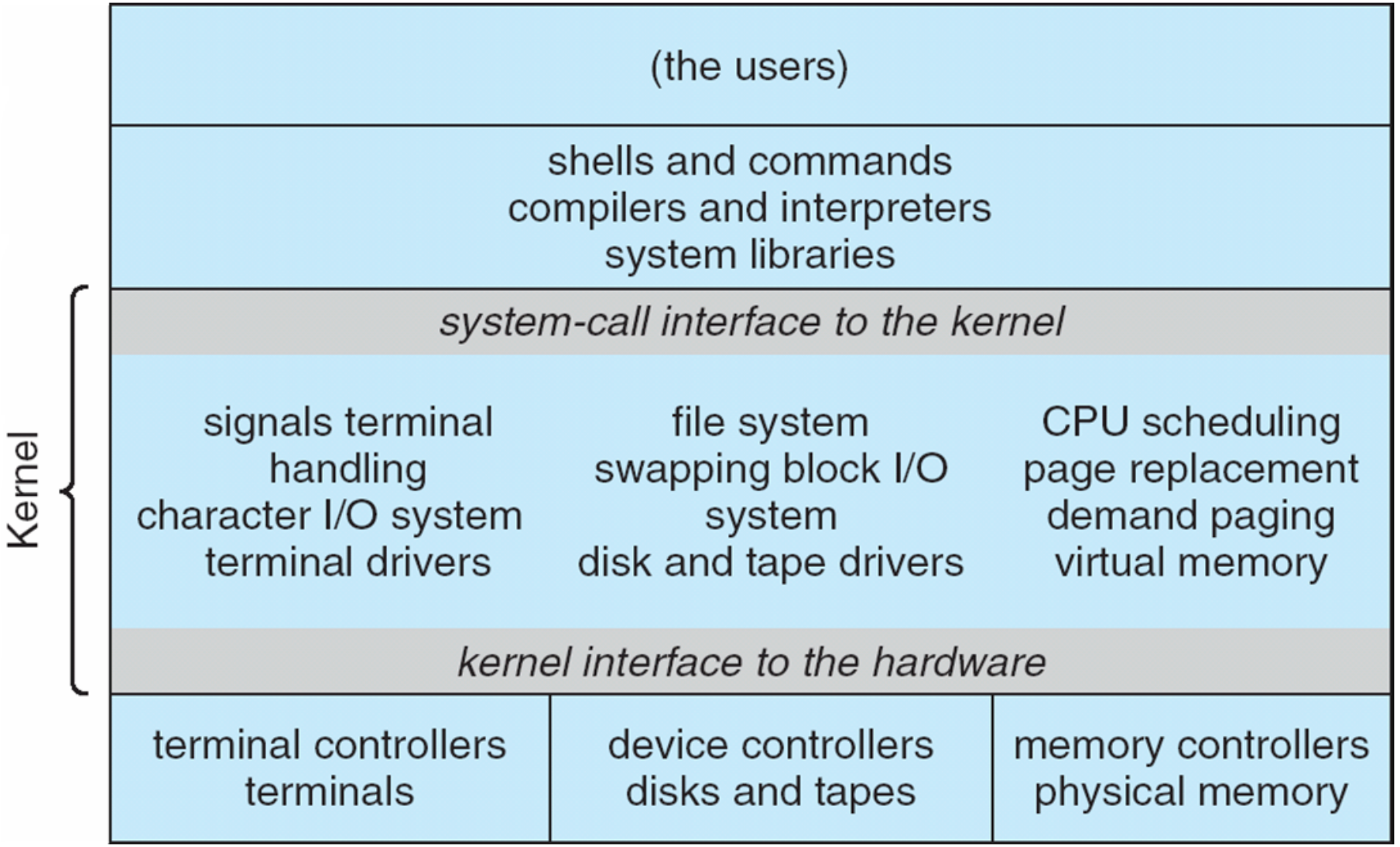
Operating System Structure

Simple Structure- MS-DOS

- Written to provide the most functionality in least space
- Not divided into modules
- Interface levels of functionality are not well separated

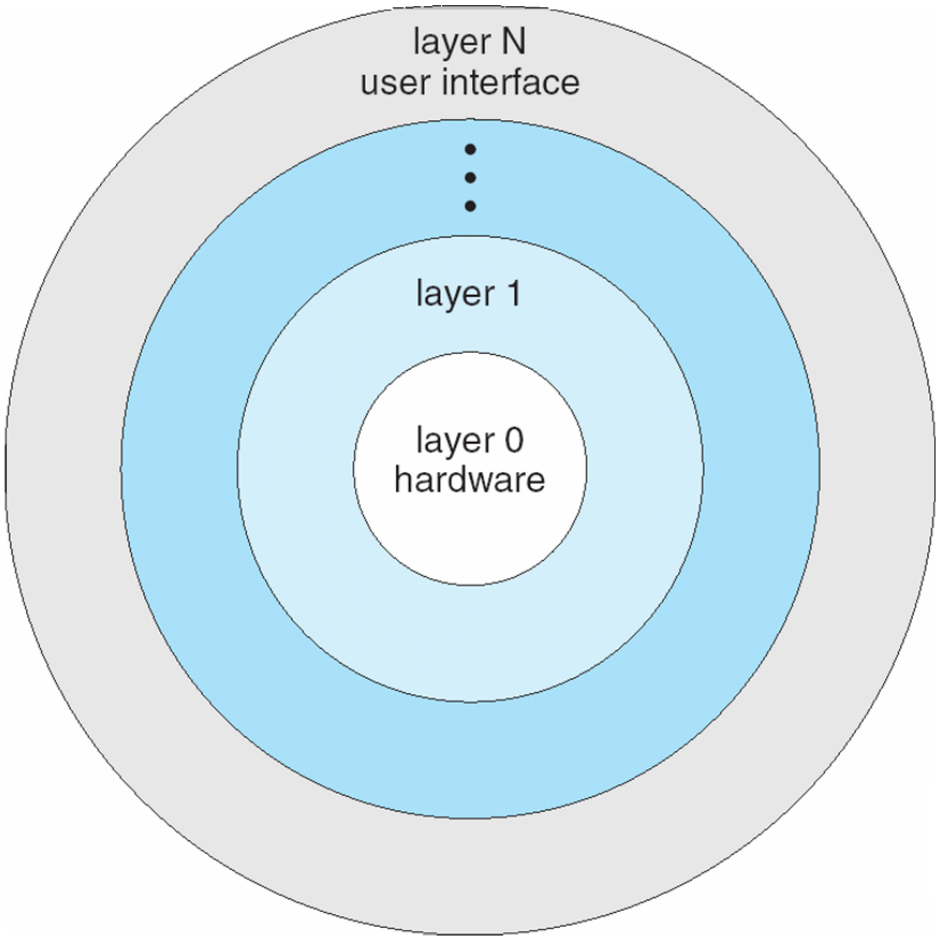


Traditional UNIX System Structure



Layered Approach

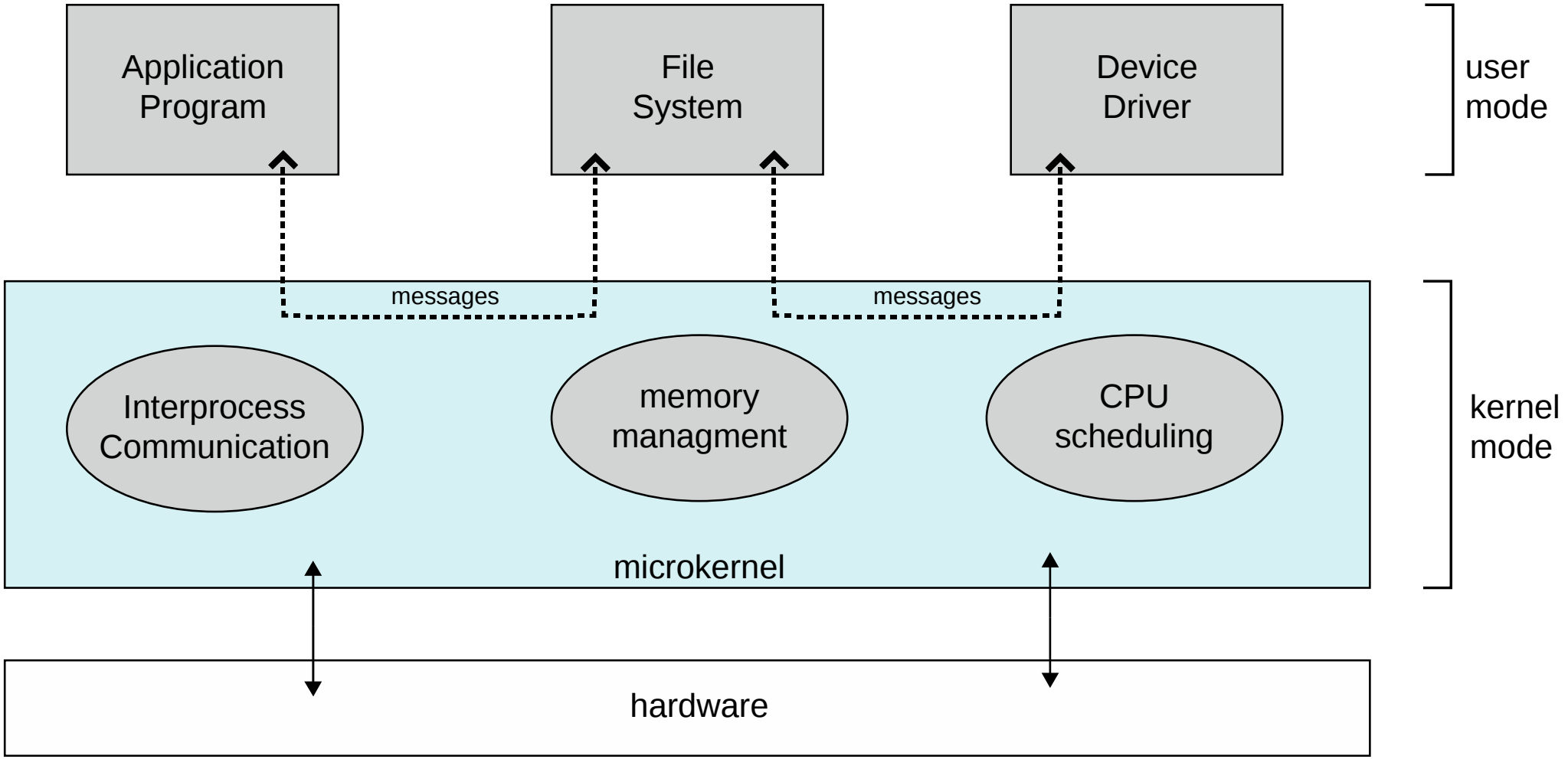
- OS is divided into a number of layers.
- Bottom layer is the hardware, highest is the user interface
- With modularity, layers are selected such that each uses functions (operations) and services of only lower-level layers



Microkernel System Structure

- Structures the OS by removing all non-essential components from the kernel and implementing them as system and user-level programs.
- Moves as much from the kernel to the user space.
- Communication takes place between user modules using message passing
- Mach- example of microkernel based OS
- Benefits-
  - Easier to extend a microkernel
  - Easier to port the OS to new architectures
  - More reliable and secure
- Disadvantage: Performance overhead of user space to kernel space communication

Microkernel System Structure



Modules

- OS design using loadable kernel modules
- Kernel has a set of core components
- Kernel links in additional services via modules, either at boot time or during run time.
- Example- Solaris, Linux, etc

Hybrid Systems

- Used by most modern operating systems
- Combines multiple approaches to address performance, security, and usability needs

System Boot

- When power is initialized on system, execution strats at a fixed memory location- Firmware/ROM which holds initial boot code
- OS must be made available to hardware at the start
- Small piece of code- bootstrap loader, stored in ROM or EEPROM locates the kernel, loads it into memory, and starts it
- Sometimes two-step process where boot block at fixed location loaded by ROM code, which loads bootstrap loader from disk
- GRUB- common bootstrap loader, allows selection of kernel from multiple disks
- Kernel loads and then system starts running