Chapter 2: Operating-System Structures

Operating System Services

- Operating systems provide an *environment for execution* of programs and *services* to programs and users
- A 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 Interface (**CLI**), Graphical User Interface (**GUI**), Touch-screen
  - Program execution - The system must be able to *load* a program into memory, to *run* that program, and *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

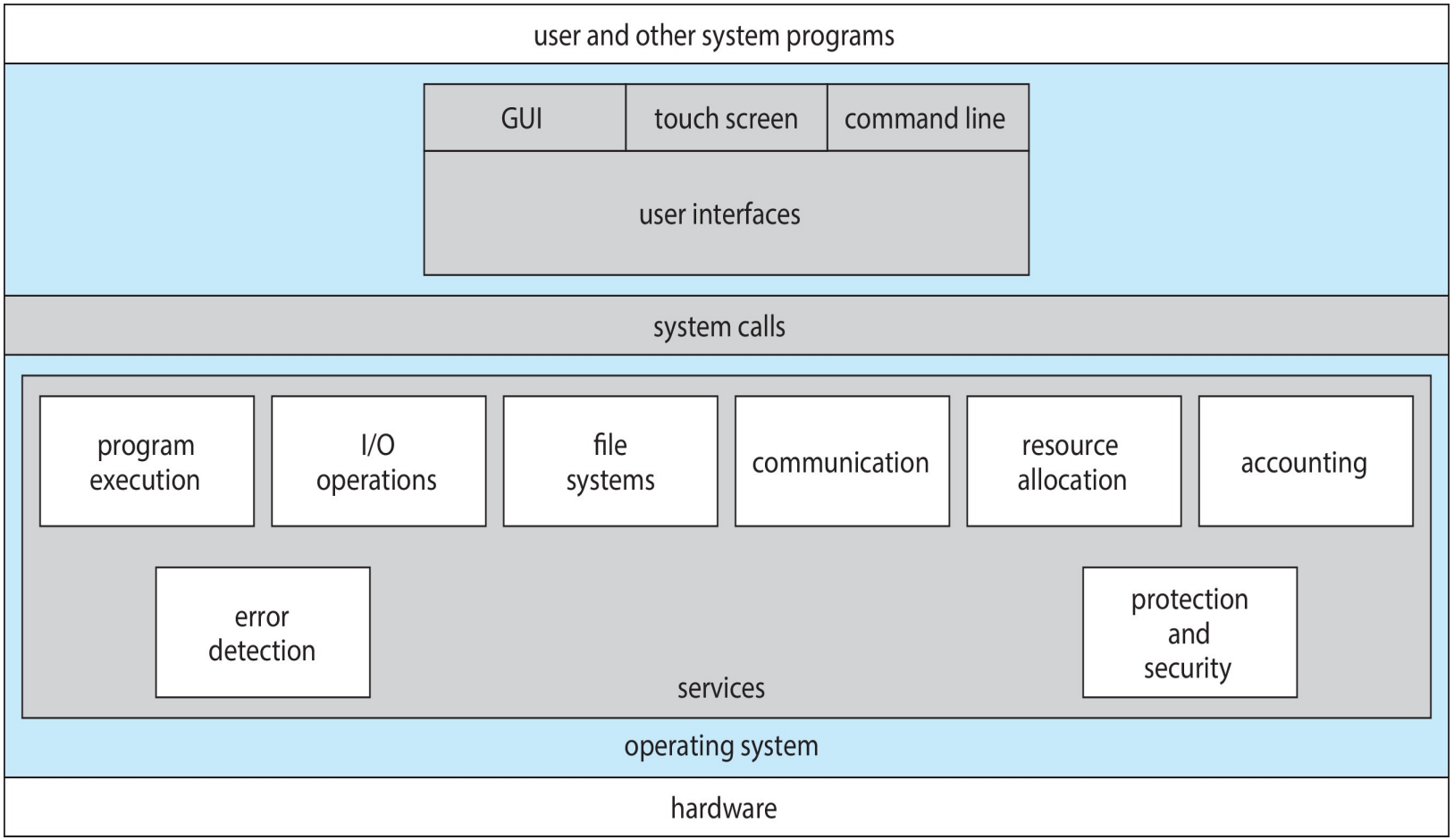
Operating System Services (Cont.)

- 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.
  - Logging - 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

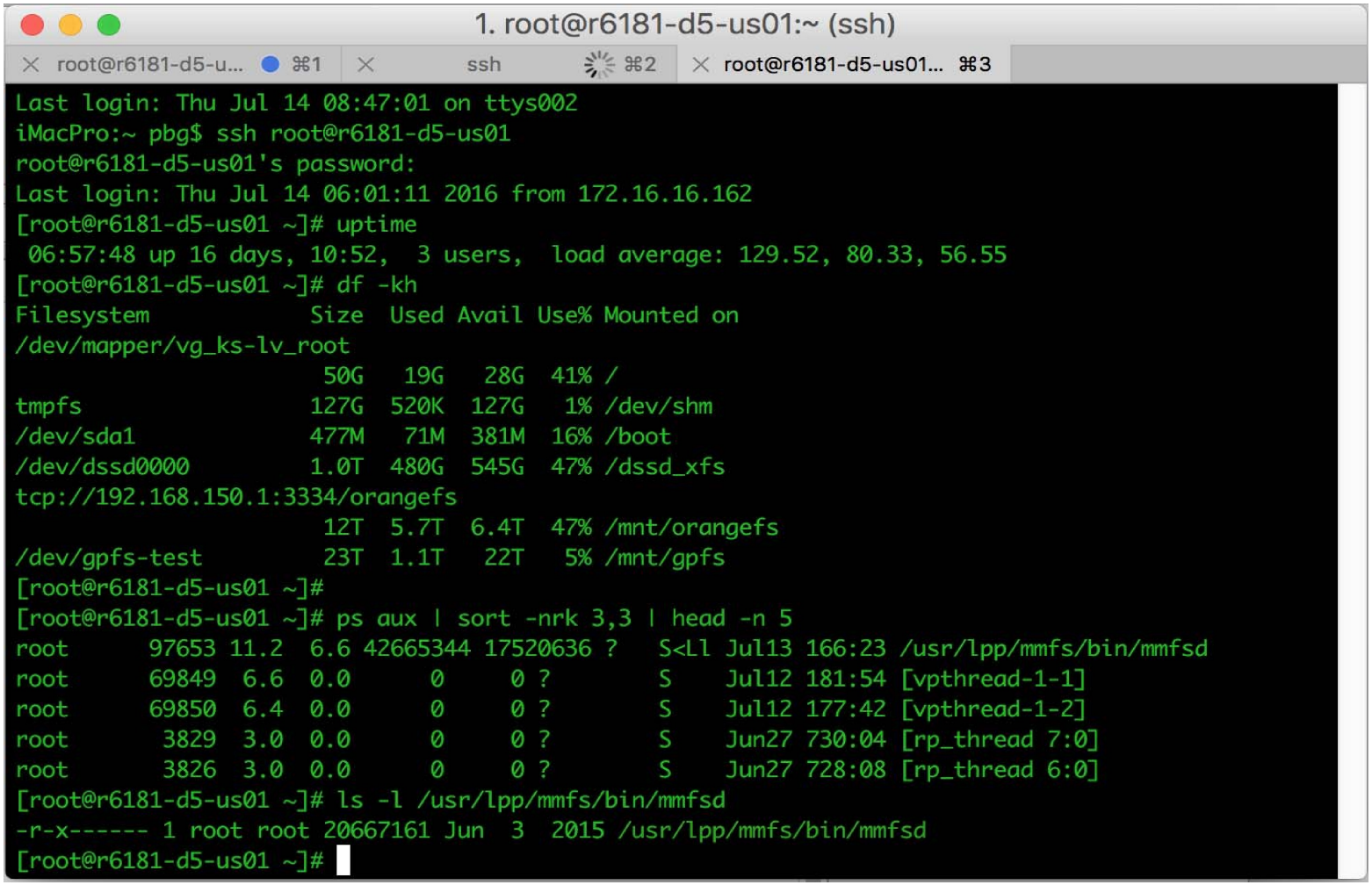
A View of Operating System Services



User Operating System Interface - CLI

- CLI or command interpreter allows direct command entry
  - Sometimes implemented in kernel, sometimes by system programs
  - 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

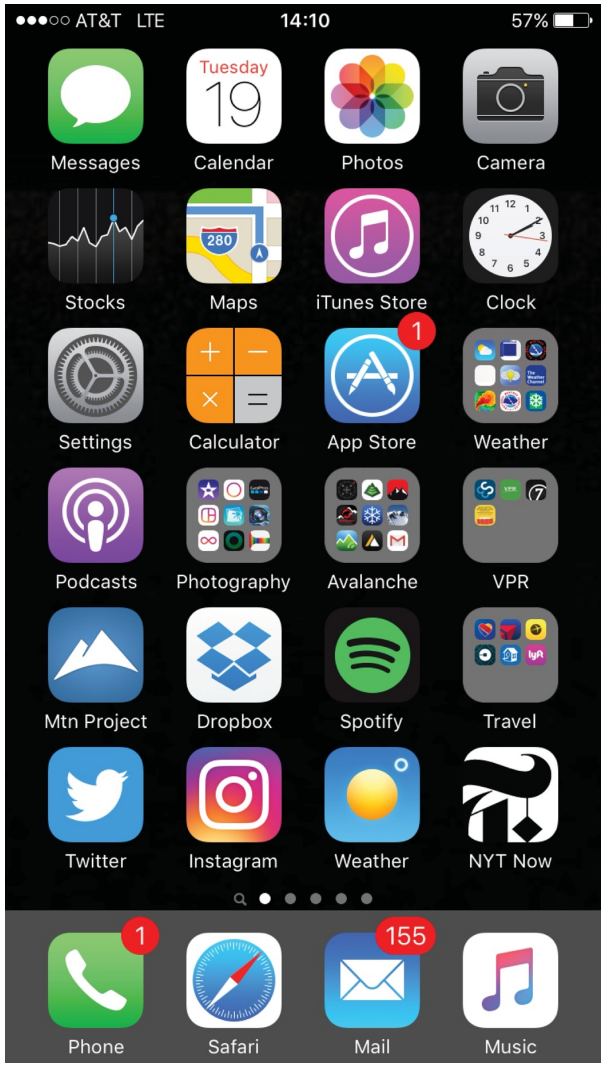
Bourne Shell Command Interpreter



User Operating System Interface - GUI

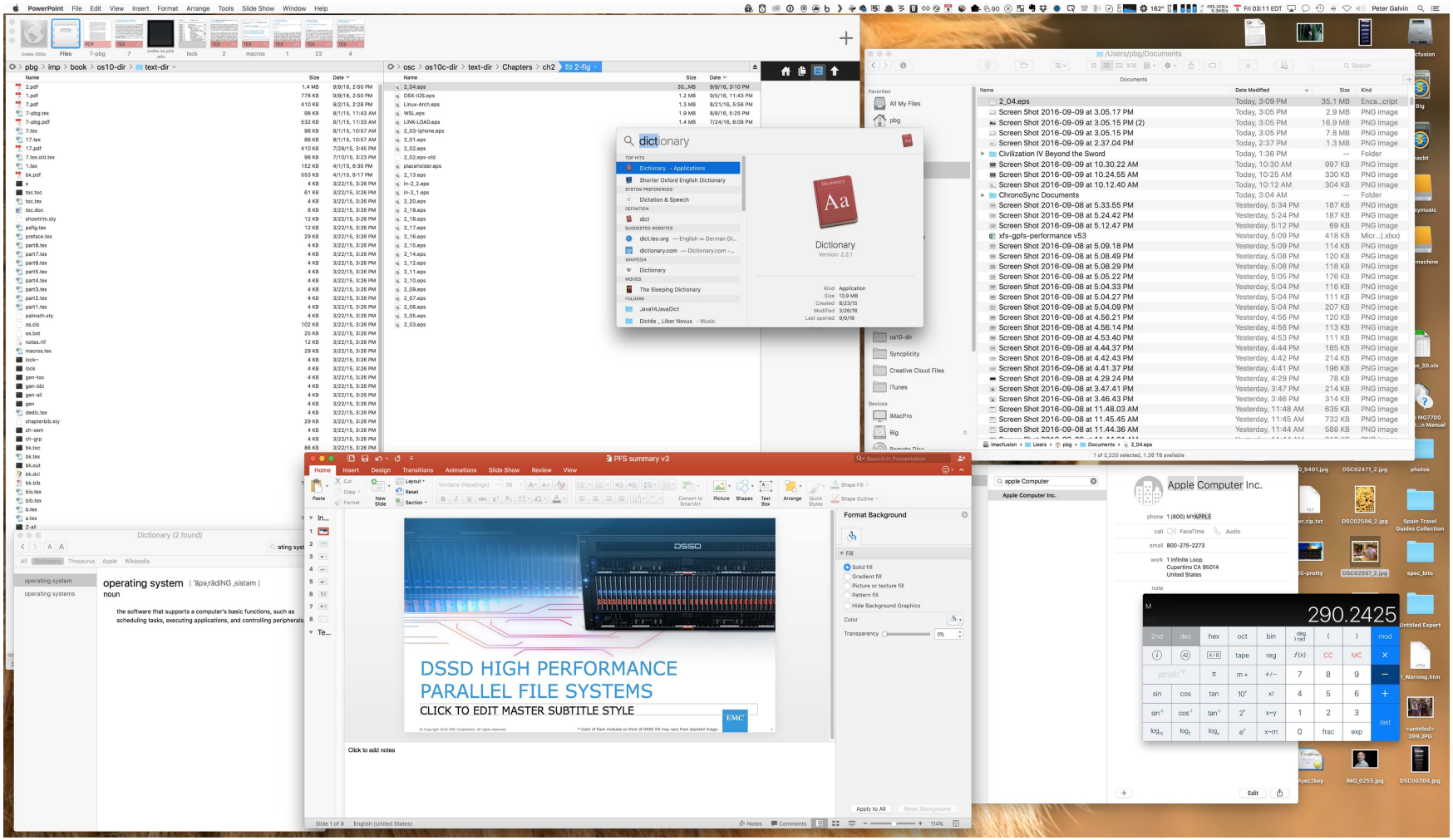
- User-friendly desktop metaphor interface
  - Usually mouse, keyboard, and monitor
  - Icons represent files, programs, actions, etc.
  - Various mouse buttons over objects in the interface cause various actions providing information, options, execute function, open directory
  - Invented at Xerox PARC
- 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 (e.g., CDE, KDE, GNOME)

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

The Mac OS X GUI



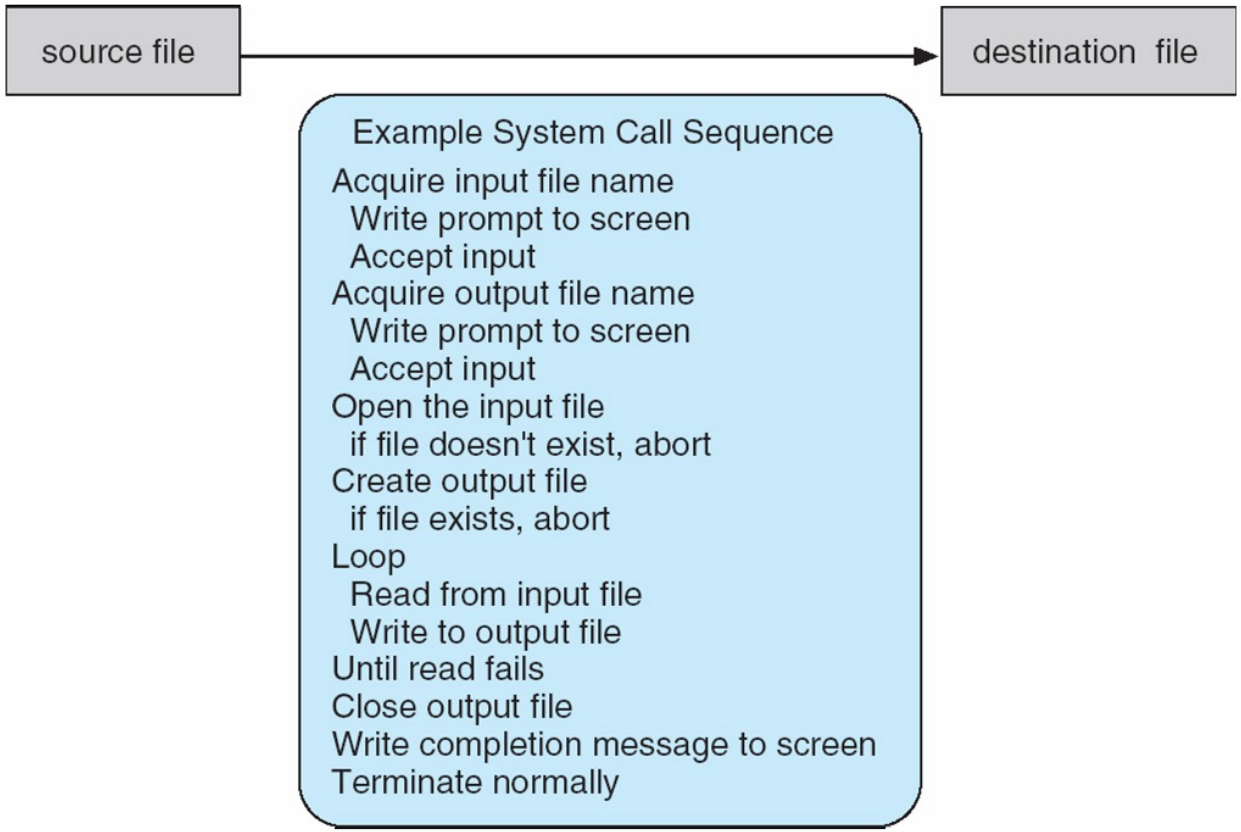
System Calls

- *Programming interface* to the services provided by the OS
- Typically written in a high-level language (e.g., 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)

Example of System Calls

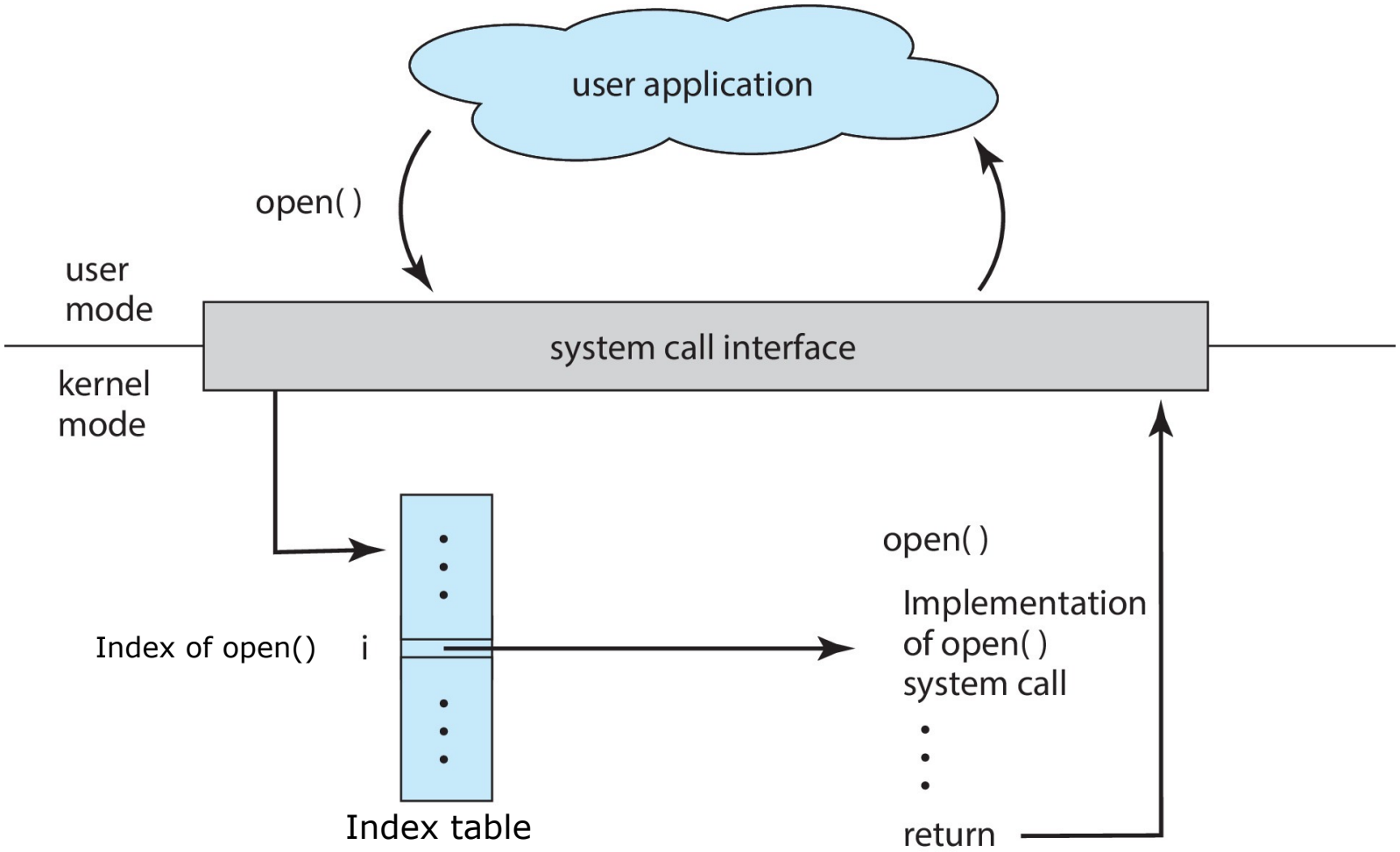
- System call sequence to copy the contents of one file to another file



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 the 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 are hidden from programmer by API
    - Managed by run-time support library (set of functions built into libraries included with compiler)

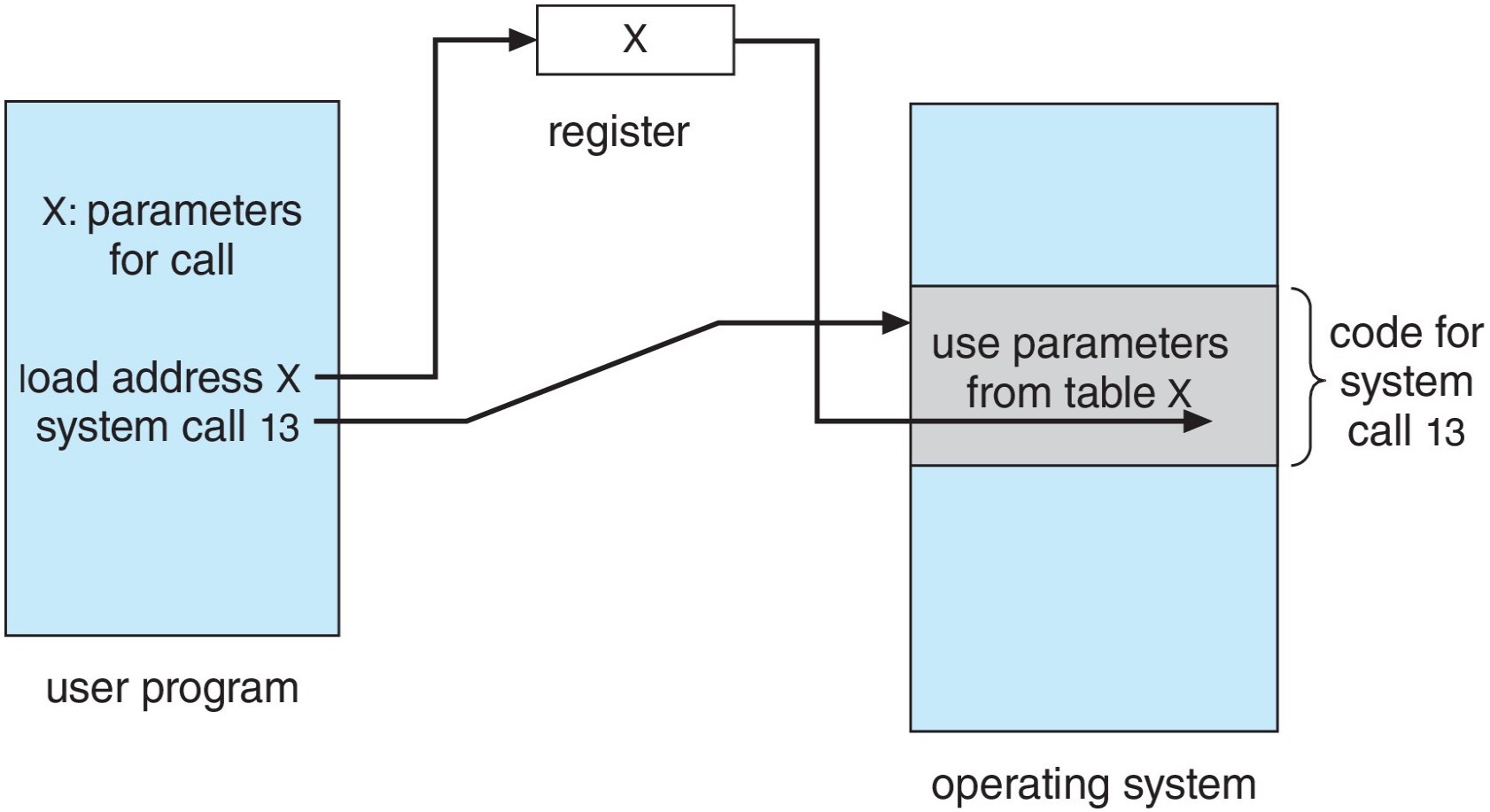
API – System Call – OS Relationship



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** *(see Memory structure)* 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**
  - create process, terminate process
  - end, abort process execution
  - 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 (cont.)

- **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

Types of System Calls (Cont.)

- **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 if using *message passing model* to *host name* or *process name*
  - *Shared-memory model* create and gain access to memory regions
  - transfer status information
  - attach and detach remote devices

Types of System Calls (Cont.)

- **Protection**
  - Control access to resources
  - Get and set permissions
  - Allow and deny user access

Examples of Windows and Unix System Calls

EXAMPLES OF WINDOWS AND UNIX SYSTEM CALLS

The following illustrates various equivalent system calls for Windows and UNIX operating systems.

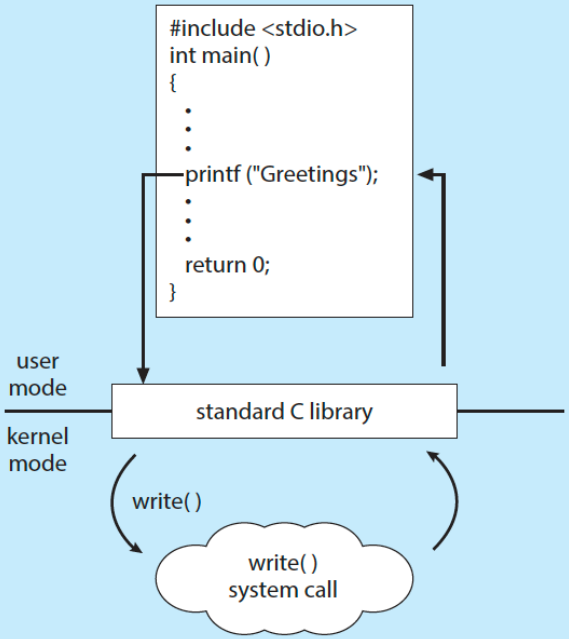
	Windows	Unix
Process control	`CreateProcess()`	`fork()`
	`ExitProcess()`	`exit()`
	`WaitForSingleObject()`	`wait()`
File management	`CreateFile()`	`open()`
	`ReadFile()`	`read()`
	`WriteFile()`	`write()`
	`CloseHandle()`	`close()`
Device management	`SetConsoleMode()`	`ioctl()`
	`ReadConsole()`	`read()`
	`WriteConsole()`	`write()`
Information maintenance	`GetCurrentProcessID()`	`getpid()`
	`SetTimer()`	`alarm()`
	`Sleep()`	`sleep()`
Communications	`CreatePipe()`	`pipe()`
	`CreateFileMapping()`	`shm_open()`
	`MapViewOfFile()`	`mmap()`
Protection	`SetFileSecurity()`	`chmod()`
	`InitializeSecurityDescriptor()`	`umask()`
	`SetSecurityDescriptorGroup()`	`chown()`

Standard C Library Example

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

THE STANDARD C LIBRARY

The standard C library provides a portion of the system-call interface for many versions of UNIX and Linux. As an example, let's assume a C program invokes the `printf()` statement. The C library intercepts this call and invokes the necessary system call (or calls) in the operating system—in this instance, the `write()` system call. The C library takes the value returned by `write()` and passes it back to the user program:



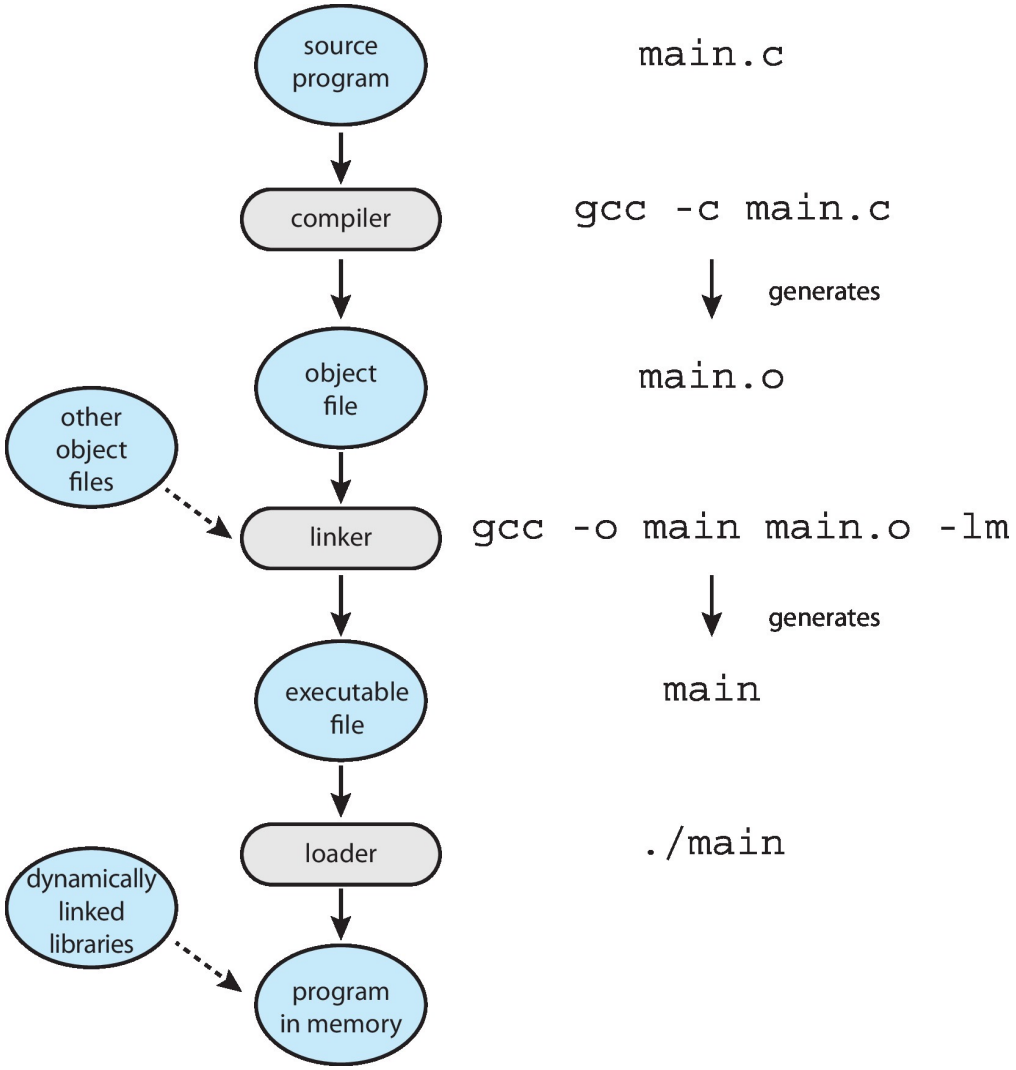
System Services

- System programs provide a convenient environment for *program development* and *execution*. They can be divided into:
  - File manipulation
  - Status information sometimes stored in a file
  - Programming language support
  - Program loading and execution
  - Communications
  - Background services
  - Application programs
- Most users' view of the operation system is defined by *system programs*, not the actual *system calls*

Linkers and Loaders

- *Source code* compiled into *object files* designed to be loaded into any physical memory location – *relocatable object file*
- **Linker** combines these (also, brings in libraries) into single *binary executable file*
- Program resides on secondary storage as binary executable and must be brought into memory by **loader** to be executed
  - *Relocation* assigns final addresses to program parts and adjusts code and data in program to match those addresses
- *Modern general purpose systems don't link libraries into executables*
  - Rather, *dynamically linked libraries* (in Windows, *DLLs*) are loaded as needed, shared by all that use the same version of that same library
- Object, executable files have standard formats, so operating system knows how to *load* and *start* them

The Role of the Linker and Loader



Operating System Structure

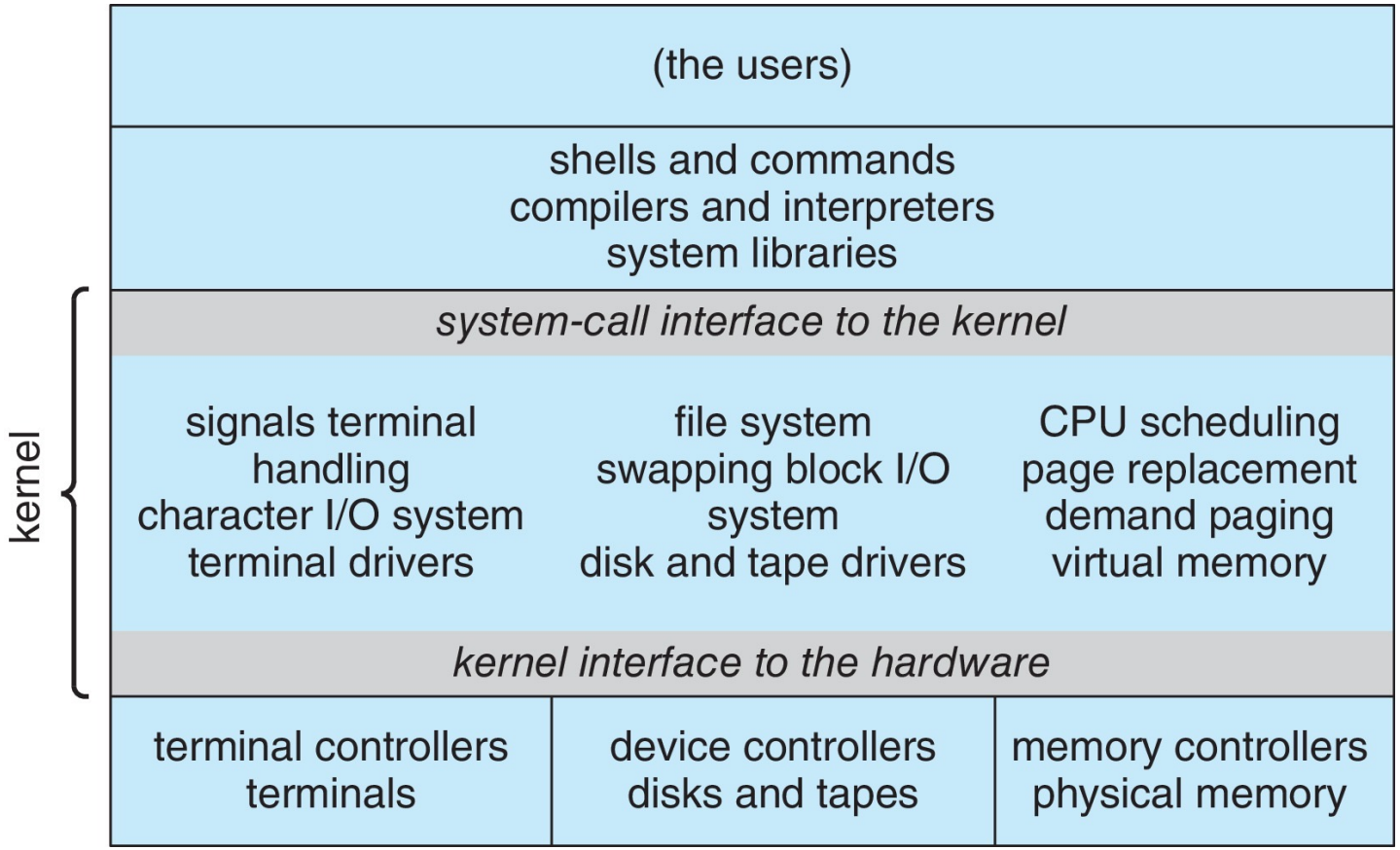
- *General-purpose OS* is very large program
- Various ways to structure ones
  - Simple structure – **MS-DOS**
  - More complex – **UNIX**
  - Layered – an abstraction
  - Microkernel – **Mach**

Monolithic Structure – Original UNIX

- UNIX – limited by hardware functionality, the original UNIX operating system had limited structuring. The UNIX OS consists of two separable parts
  - **Systems programs**
  - **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*

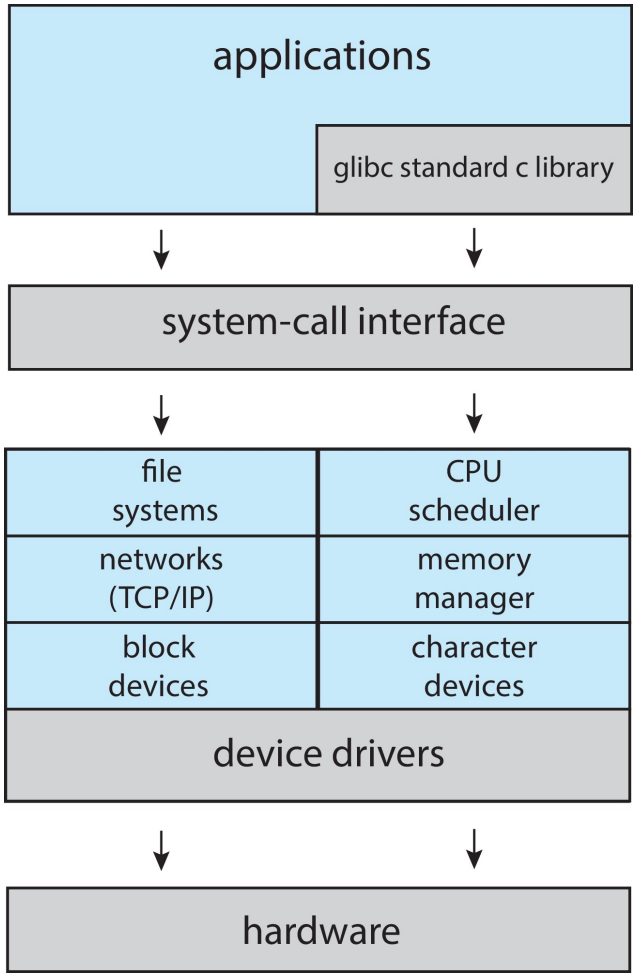
Traditional UNIX System Structure

- Beyond simple but not fully layered



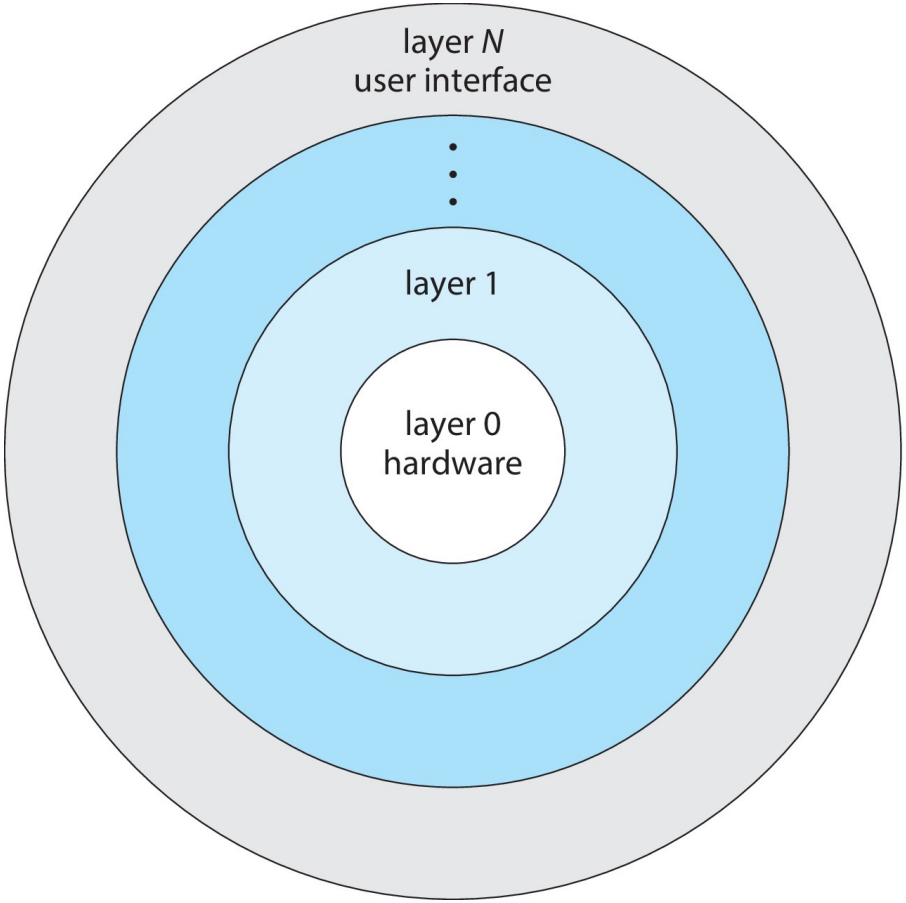
Linux System Structure

- Monolithic plus modular design



Layered Approach

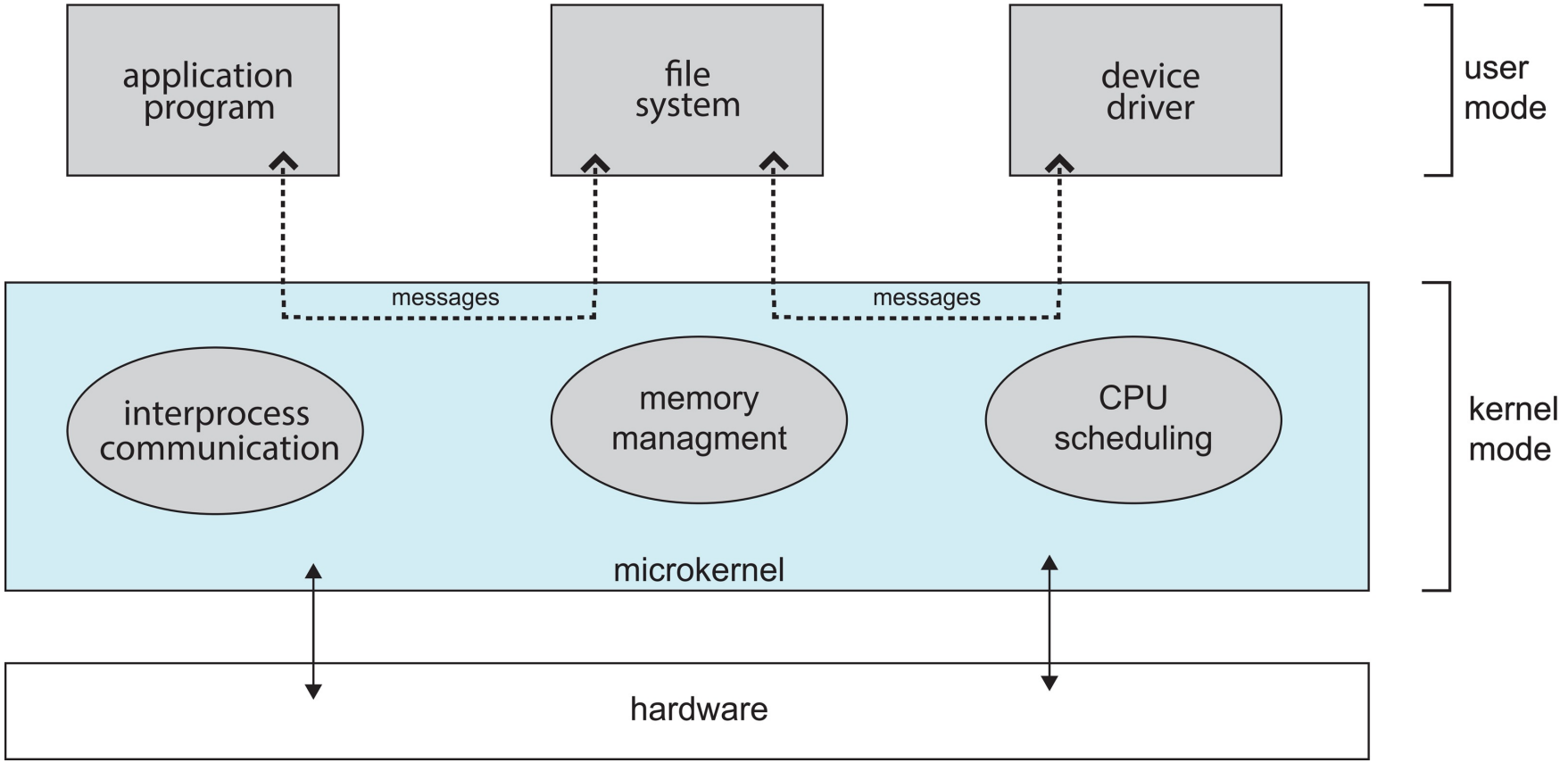
- The operating system is divided into a *number of layers* (levels), each built on top of lower layers. The bottom layer (layer 0), is the *hardware*; the highest (layer N) is the *user interface*.
- With *modularity*, layers are selected such that *each uses functions (operations) and services of only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



Microkernels

- Moves as much from the *kernel* into *user space*
- **Mach** is an example of *microkernel*
  - Mac OS X kernel (i.e., **Darwin**) partly based on Mach
- Communication takes place between user modules using *message passing model*
- 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 System Structure



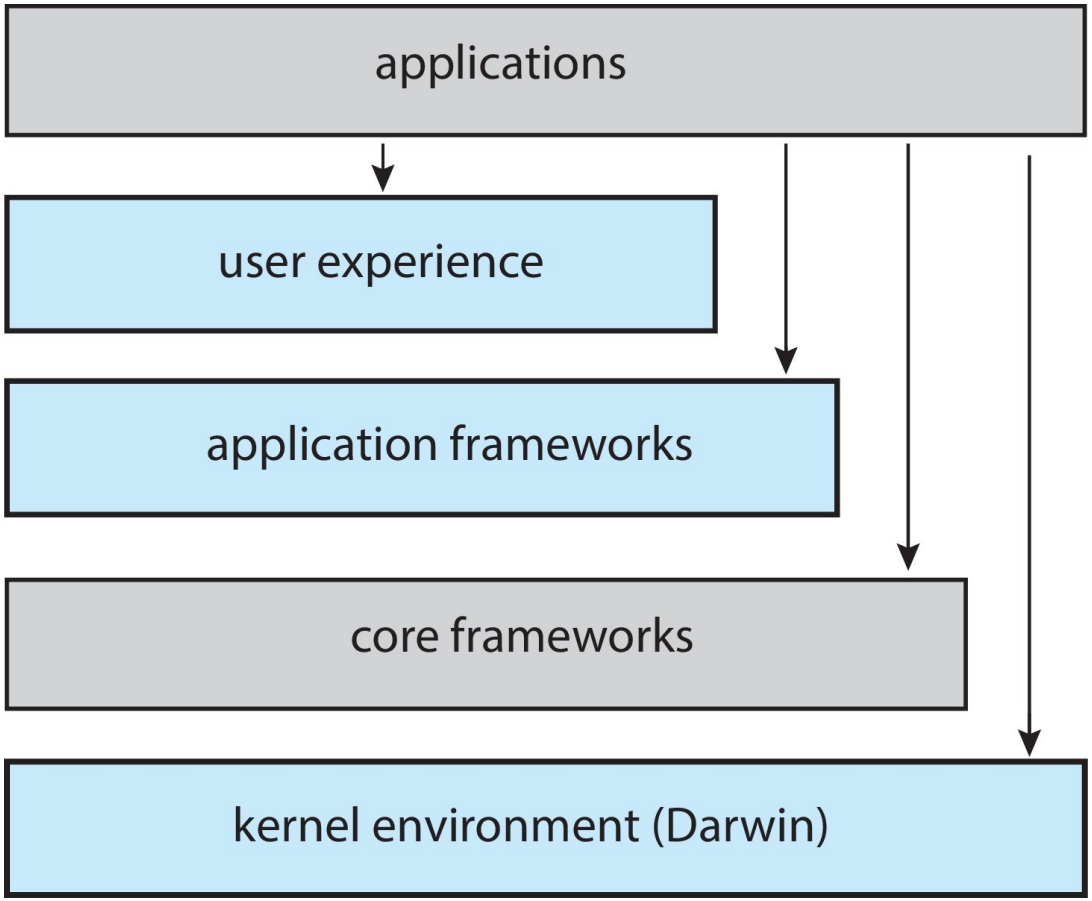
Modules

- Many modern operating systems implement **Loadable Kernel Modules** (**LKM**s)
  - 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, etc.

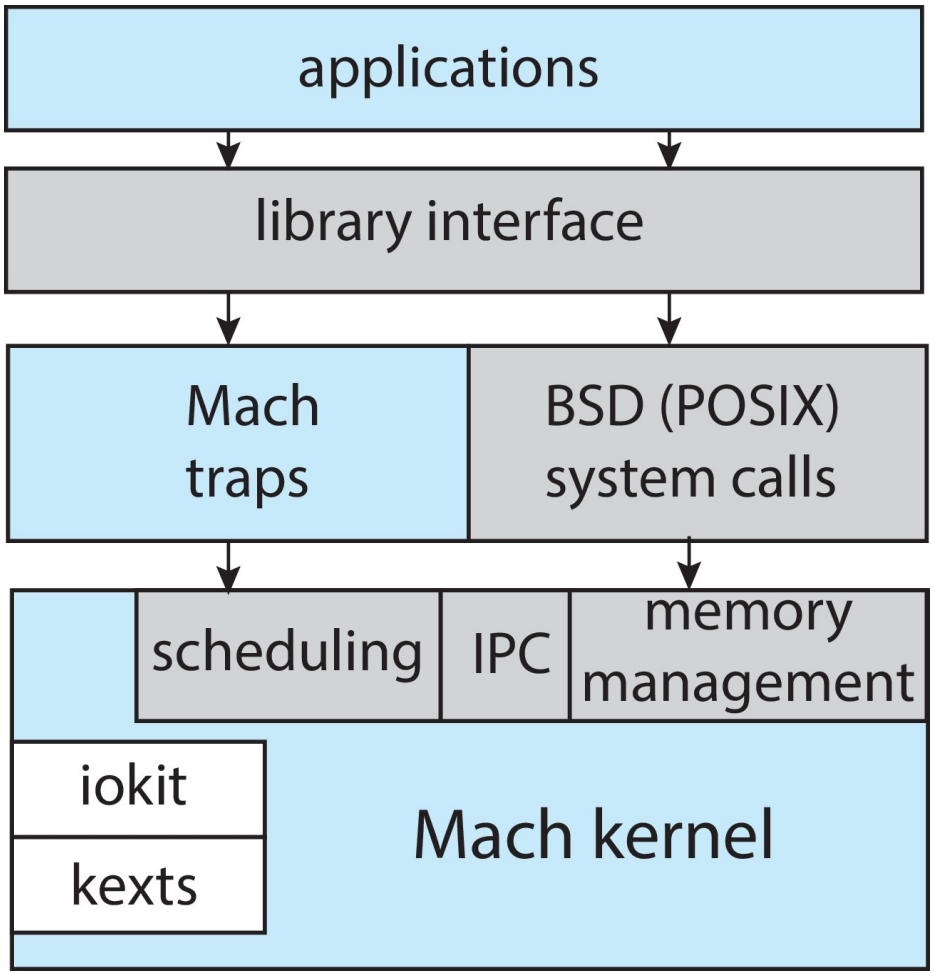
Hybrid Systems

- Most modern operating systems are 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 for different *subsystem personalities*
- 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*)

macOS and iOS Structure



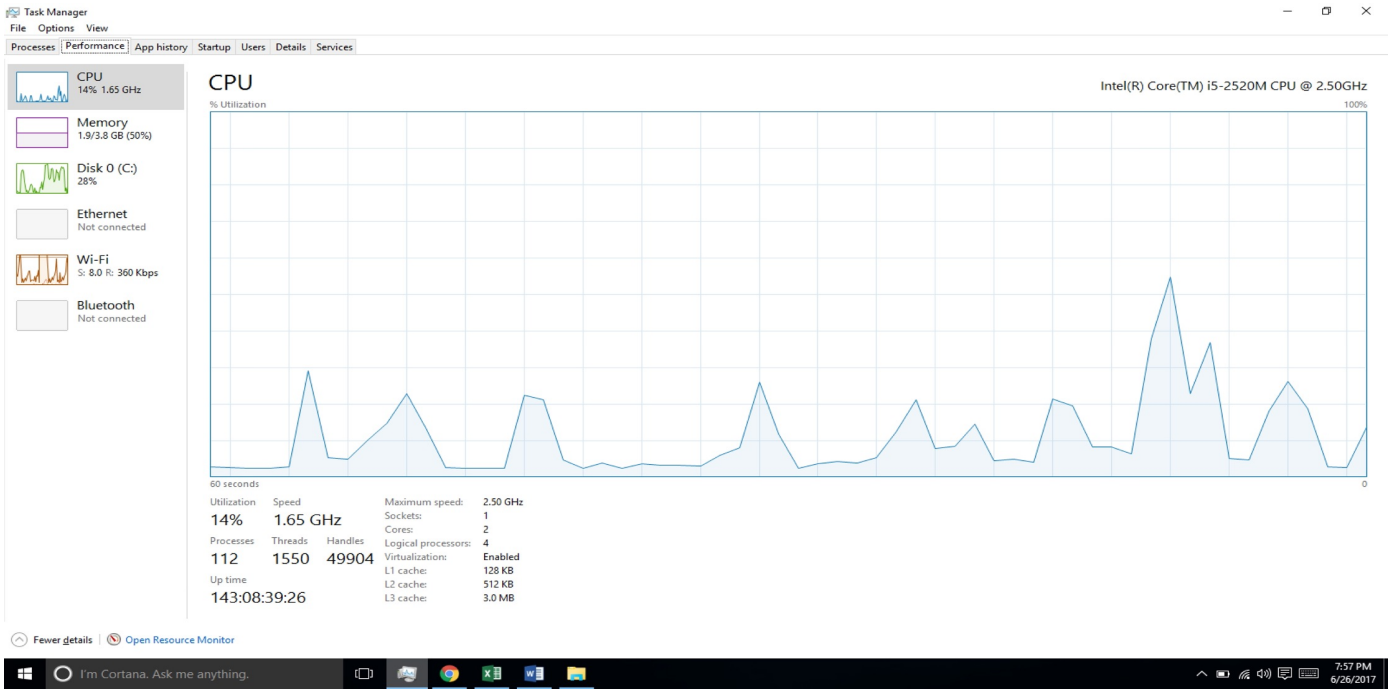
Darwin


applications
library interface
Mach traps
BSD (POSIX) system calls
scheduling
IPC
memory management
iokit
kexts
Mach kernel

Building and Booting Linux

- Download Linux source code (http://www.kernel.org)
- Configure kernel via "**`make menuconfig`**"
- Compile the kernel using "**`make`**"
  - Produces **`vmlinuz`**, the kernel image
  - Compile kernel modules via "**`make modules`**"
  - Install kernel modules into **`vmlinuz`** via "**`make modules_install`**"
  - Install new kernel on the system via "**`make install`**"

Performance Tuning



- Improve performance by removing *bottlenecks*
- OS must provide means of computing and displaying measures of system behavior
- For example, "`top`" Linux program or **Windows Task Manager**