Teaching assistants:

Minghao Zheng, mz2986@nyu.edu
Xinyuan Zhang, xz3065@nyu.edu

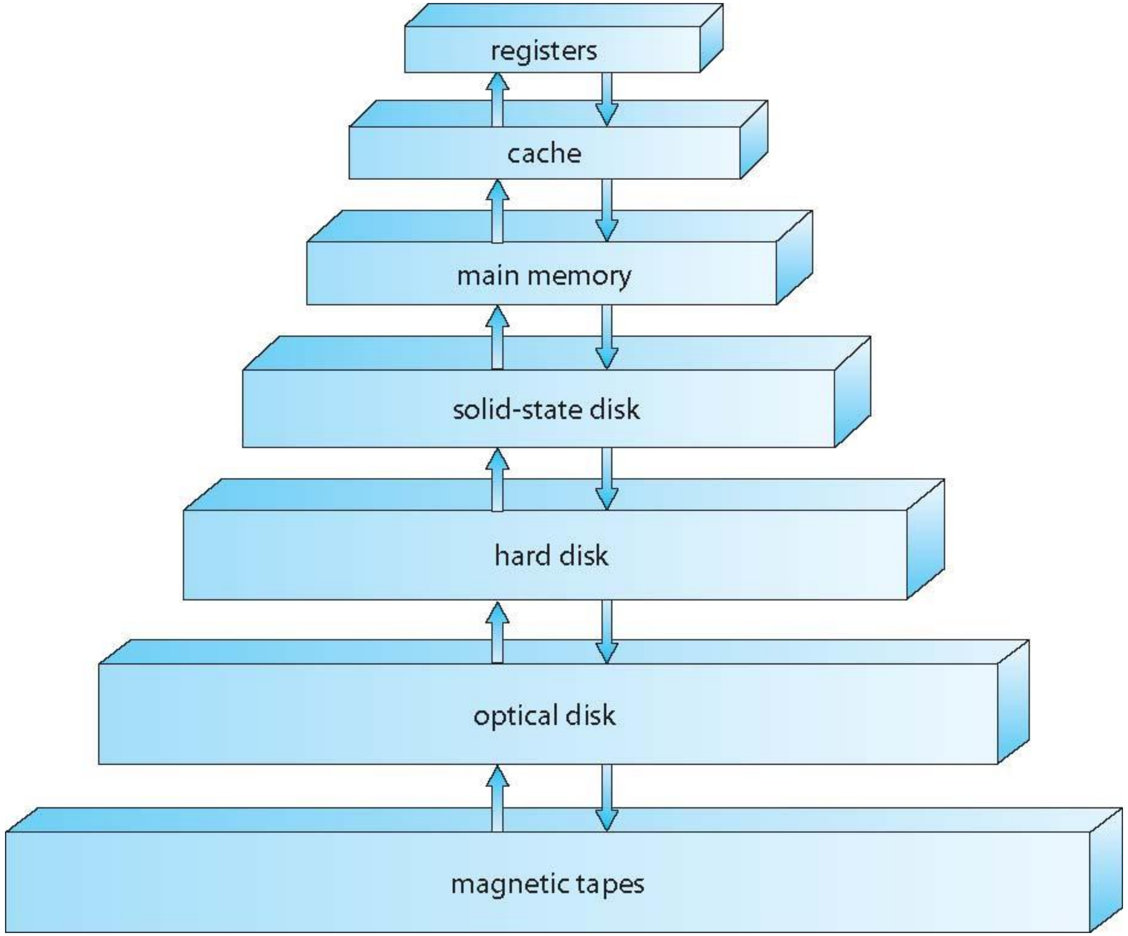
Storage Structure

- Main memory – is the only storage media that the CPU can access **directly**:
  - RAM: random access memory, volatile
  - ROM: read only memory, non-volatile (e.g. ROM, EEPROM or flash)
- Secondary storage – extension of main memory that provides large nonvolatile storage capacity, however, the CPU can only access this memory **indirectly** via a device controller (using its control/status and data interfaces)
  - **Hard disks** – rigid metal or glass platters covered with magnetic recording material.
    - Disk surface is logically divided into tracks, which are subdivided into sectors.
  - **Solid-state disks** – faster than hard disks, also nonvolatile
    - Becoming more popular

Storage Hierarchy

- Storage systems organized in hierarchy
  - Speed
  - Cost

  (usually, the larger the memory, the slower it is)
- **Caching** – As a concept, it means copying information into faster storage system; main memory can be viewed as a cache for secondary storage



Caching

- Important principle, performed at **many levels** in a computer (in hardware, operating system, software)
- Information in use is copied from slower to faster storage temporarily
  - Faster storage (cache) checked first to determine if information is there
    - If it is, information used directly from the cache (cache hit).
    - If not, data copied to cache and used there (cache miss).
- Why is it advantageous to use cache?
- Cache management is an important design problem
  - Cache size (affects speed + cost)
  - Replacement policy (e.g. LIFO, LRU, etc.)

Direct Memory Access Structure

- Used for high-speed I/O devices able to transmit information at close to memory speeds (e.g. gigabit ethernet)
- Device controller transfers blocks of data from buffer storage directly to main memory without CPU intervention
- Only one interrupt is generated per block of data, rather than the one interrupt per word or byte.

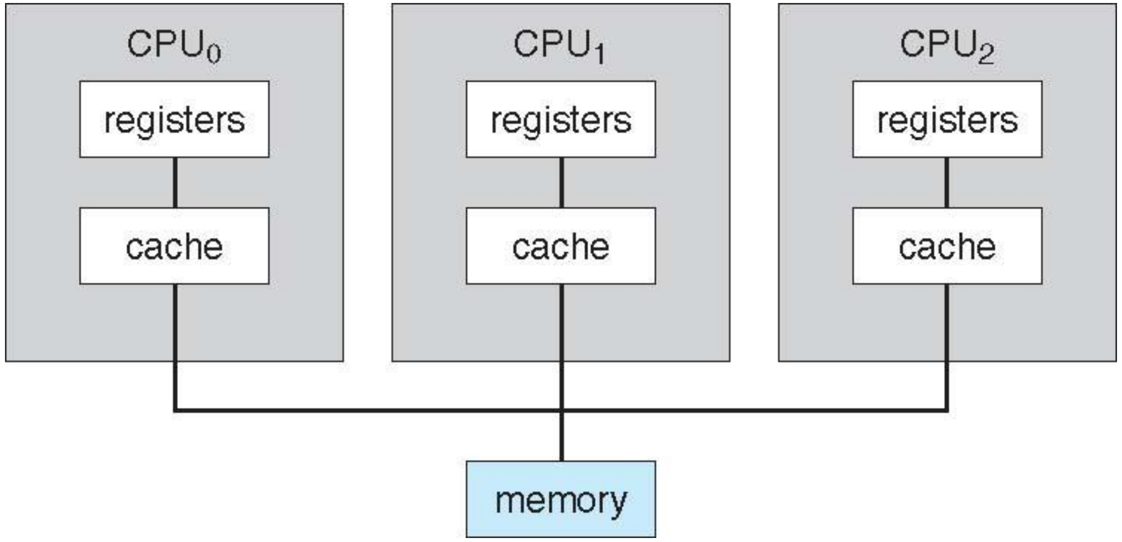
Computer-System Architecture

- Many systems use a **single general-purpose processor**
  - Also often referred to as **application processor**.
  - Most systems have special-purpose processors as well (e.g. a GPU or a DSP), but these do not make the system a multiprocessor system.

- **Multiprocessors** systems growing in use and importance
  - Also known as **parallel systems**, **tightly-coupled systems**
  - Advantages include:
    1. **Increased throughput**
    2. **Economy of scale**
    3. **Increased reliability** – graceful degradation or fault tolerance

Computer-System Architecture – cont.

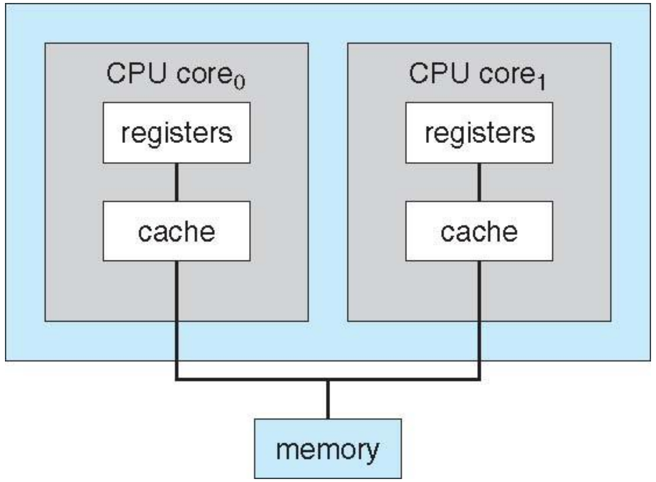
- **Multiprocessors** - Two types:
  1. **Asymmetric Multiprocessing** – processors are not treated as equal.
     - Processors may be dedicated to specific tasks
     - e.g. boss and worker processors
  2. **Symmetric Multiprocessing (SMP)** – all processors are treated equally
     - Single instance of the OS.
     - Each processor is **capable** of performing any task, such as handling interrupts, running the OS kernel, running applications, etc.



Symmetric multiprocessors

Multi-Core Designs

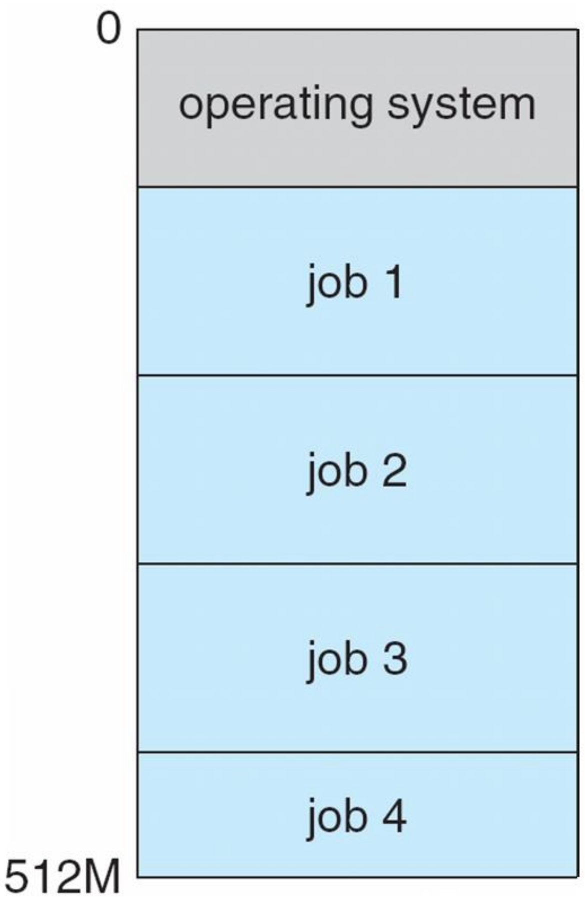
- A **multicore** system may have multiple cores in a single chip, and is thus a multiprocessor system.
- Systems may be built of multiple chips, each with multiple cores.



Operating System Structure

- **Multiprogramming batch systems**
  - A bit historical
  - Needed for efficiency: Single user cannot keep CPU and I/O devices busy at all times
  - Multiprogramming organizes jobs (code and data) so CPU always has one to execute
  - A subset of total jobs in system is kept in memory
  - One job selected and run via **job scheduling**
  - When it has to wait (for I/O for example), OS switches to another job
  - Typically used **non-preemptive = cooperative** multitasking

- **Timesharing/interactive systems**
  - Logical extension in which CPU switches jobs so frequently that users can interact with each job while it is running, creating **interactive** computing (**response time** should be < 1 second) -> **preemptive**
  - Each user has at least one program executing in memory ⇨**process**
  - If several jobs ready to run at the same time ⇨ **CPU scheduling** selects one of them

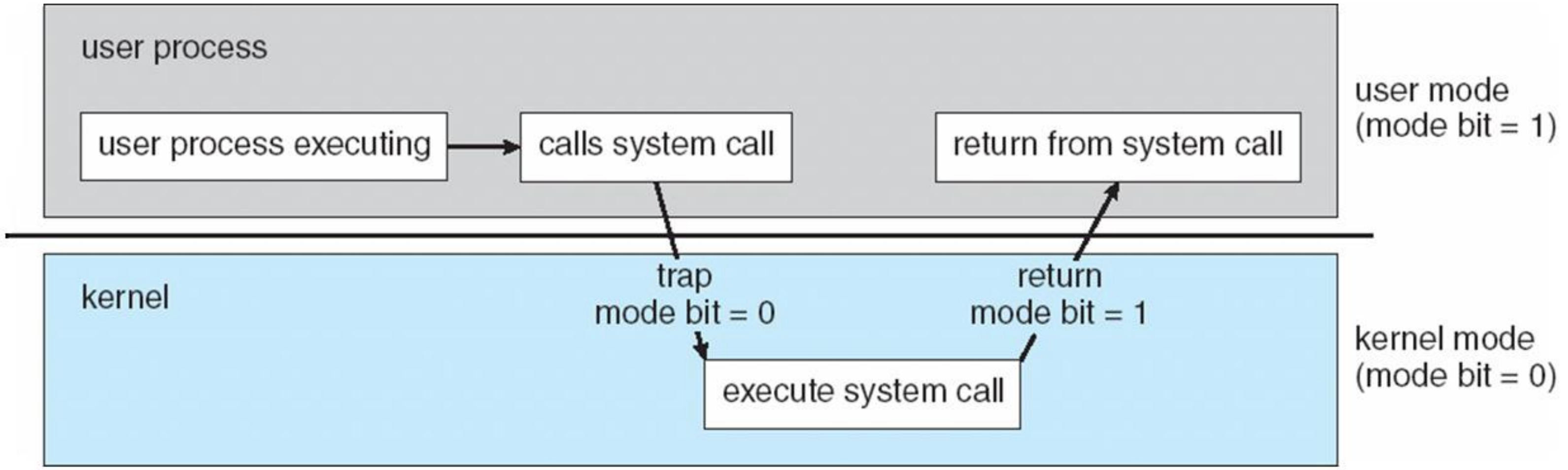


Operating-System Operations

- The operating system is invoked via interrupts. Invocation may be due to either:
  - Hardware interrupts by one of the devices.
    - A **timer** interrupt is a hardware interrupt caused by an on-chip timer, and is used to preempt applications and invoke the OS kernel on regular intervals (called **OS tick**)
      - The interrupt interval is usually 1 to 50 mS.
    - Interrupts **from device controllers**.
  - Software interrupts (**system request**/**trap):**
    - Operating system services are requested using the **trap** instruction, which is a software interrupt.
    - Illegal operations:
      - Software error (e.g., division by zero) causes an exception
      - Illegal instruction or illegal access also cause exceptions.
- Note that the operating system may also be running its own threads (kernel threads).
  - The OS scheduler thus schedules user jobs/processes AND kernel threads

Operating-System Operations (cont.)

- **Dual-mode** operation allows OS to **protect itself** and other system components
  - **Mode bit** provided by hardware determines whether the CPU is in **User mode** or **kernel mode.**
    - Provides ability to distinguish when system is running user code or kernel code
    - Some **instructions** designated as **privileged**, only executable in kernel mode
    - Some **memory locations** may be configured to be only accessible in kernel mode.
  - System call (using the trap instruction) changes mode to kernel-mode.
  - Return from a system call resets the mode bit back to user-mode



Transition from user mode to kernel mode

Operating-System Operations (cont.)

- Increasingly CPUs support multi-mode operations:
  - Privileged/kernel modes: e.g. interrupts, kernel threads, etc.
  - Non-privileged/user modes: e.g. user threads/processes, **virtual machine manager** (**VMM**) mode for guest **VMs**, etc.

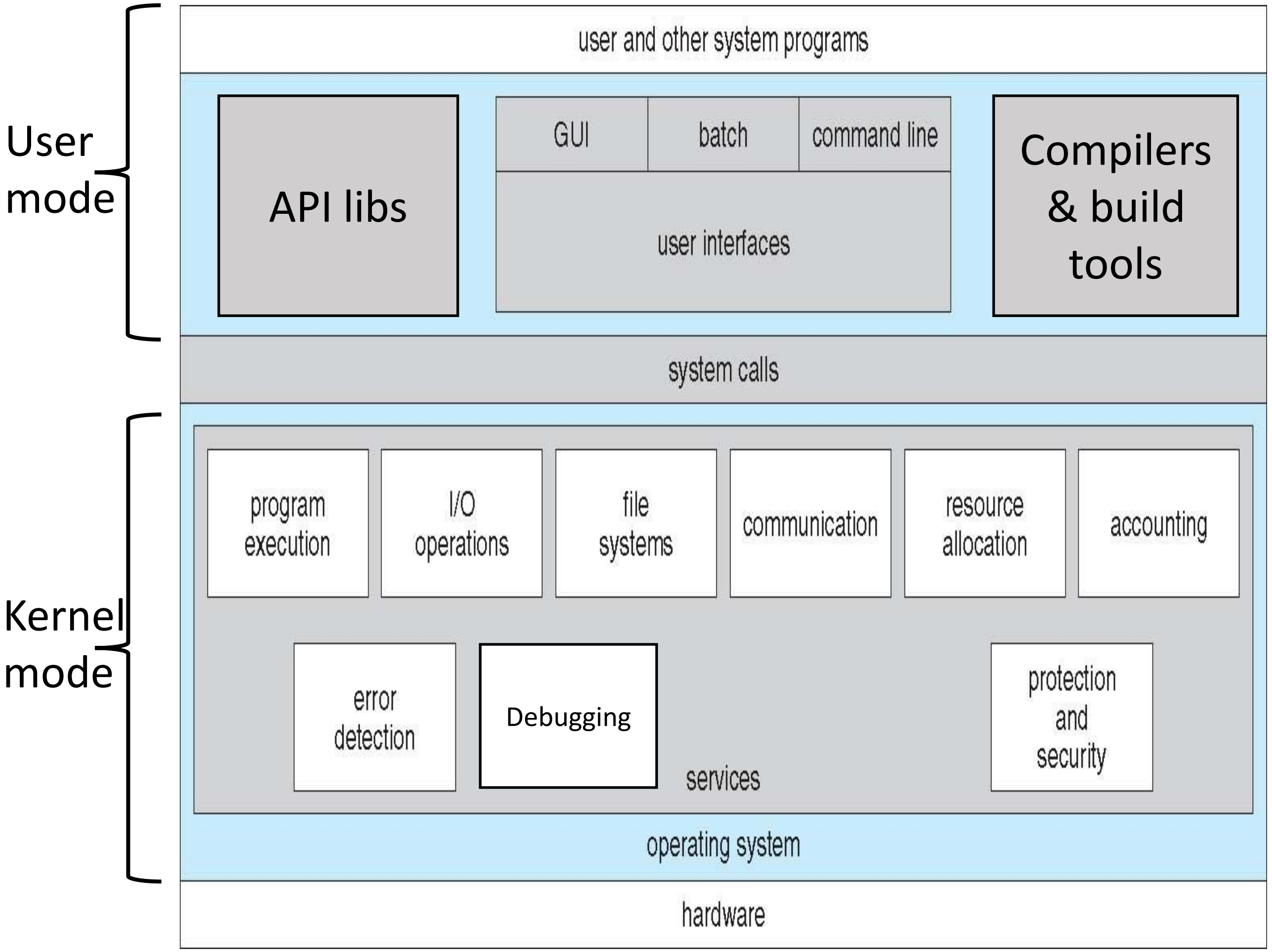
Operating-System Operations (cont.)

- Timer to prevent infinite loop / process hogging resources (**in preemptive multitasking**)
  - Timer is set to interrupt the CPU after some time period (e.g. 1 – 50 ms)
    - The interrupt is handled by the OS kernel.
    - The timer registers are memory-mapped to a memory area that can be accessed only in privileged mode.

  - The kernel sets up the timer for the next interrupt (timer tick) before scheduling a process to run.
    - This is in order to regain control or preempt a running process that exceeds its allotted time

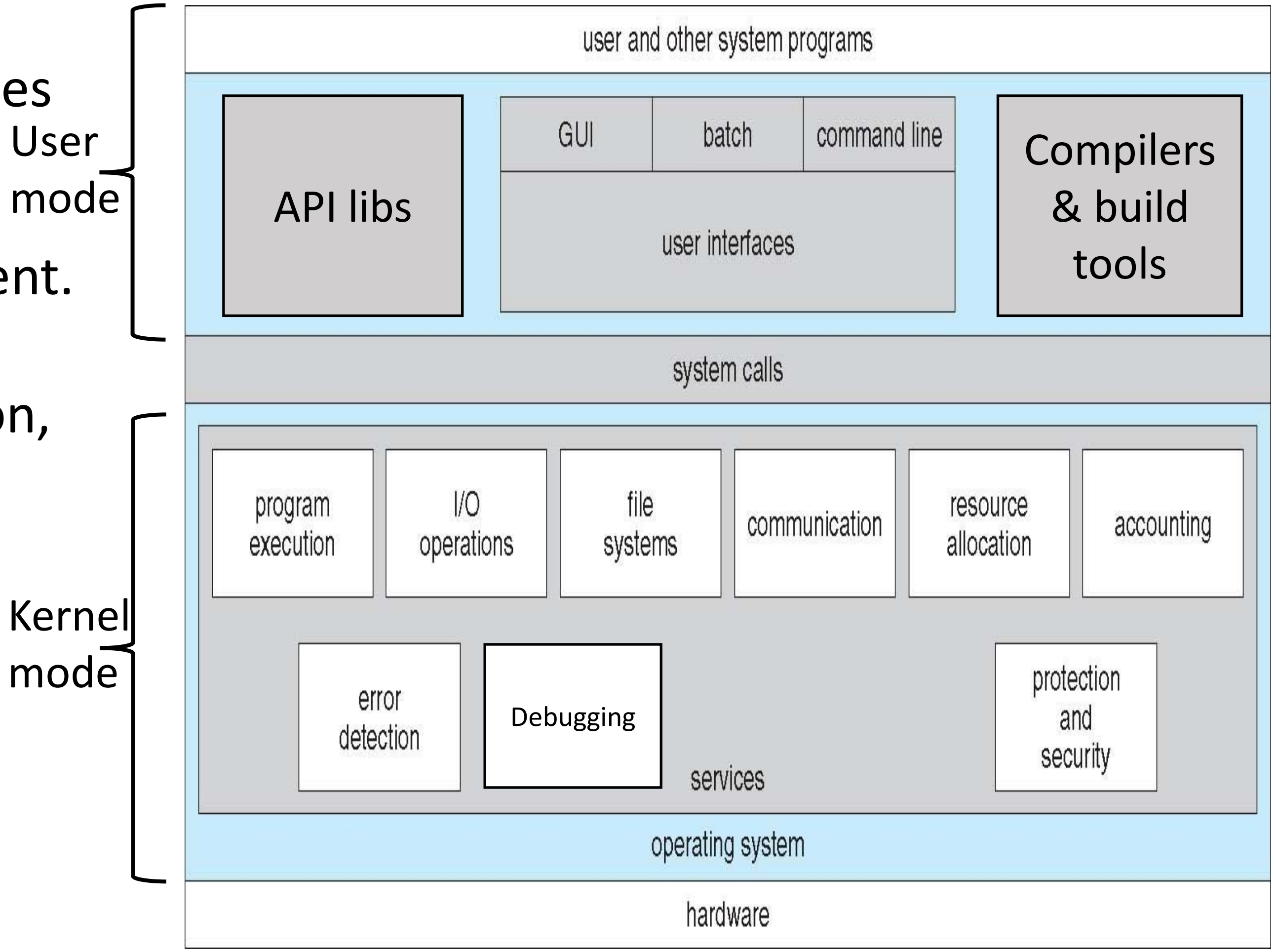
2.1 Operating System Services

- One of the operating systems' main tasks is to provide **an environment and services** for application programs to run:
  - **API Libraries**
  - **Compilers and build tools**
  - **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 and also end its execution (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



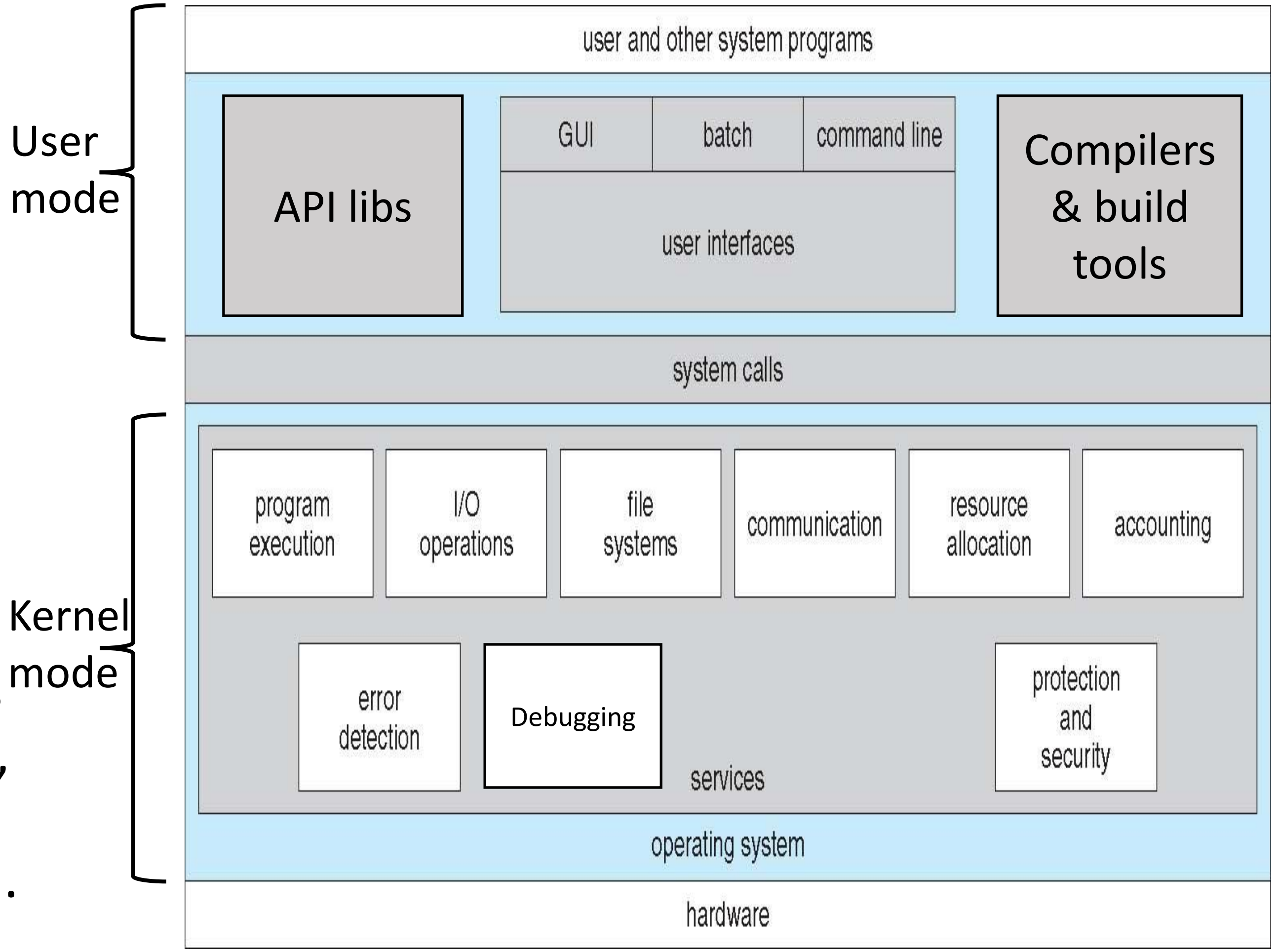
Operating System Services (Cont.)

- **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.
- **Interprocess 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 or messages moved by the OS)



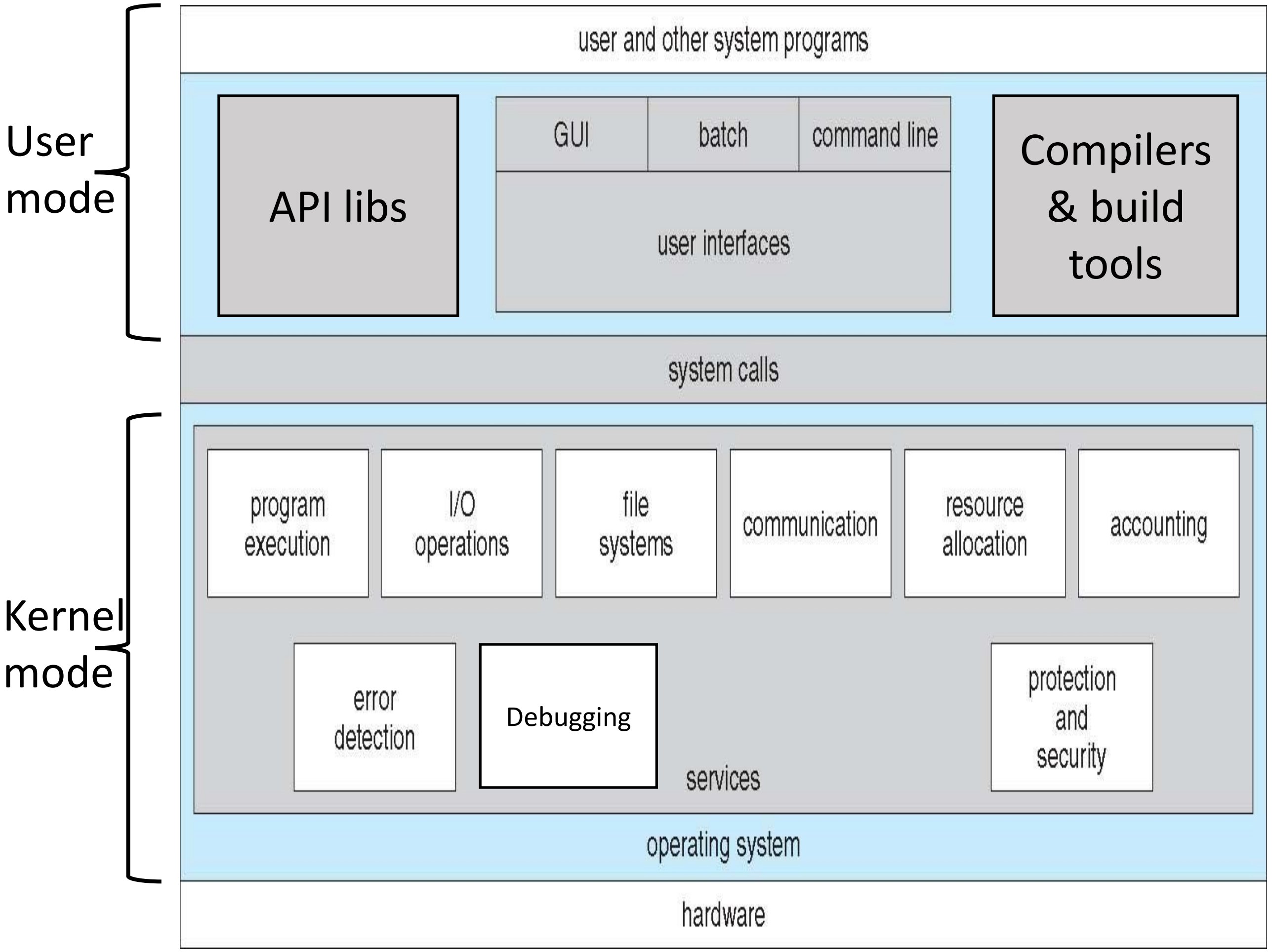
Operating System Services (Cont.)

- **Error detection** – OS needs to be constantly aware of possible errors
    - May occur in the CPU and memory hardware, in I/O devices, or in user programs.
    - For each type of error, OS should take the appropriate action to ensure correct and consistent computing
- **Debugging:** OS provides debugging facilities which can greatly enhance the user's and programmer's abilities to efficiently use the system. In Linux, the kernel component that provides this facility is the "**ptrace**" system call. User mode debuggers (e.g. **gdb**) use this kernel facility to debug user programs.



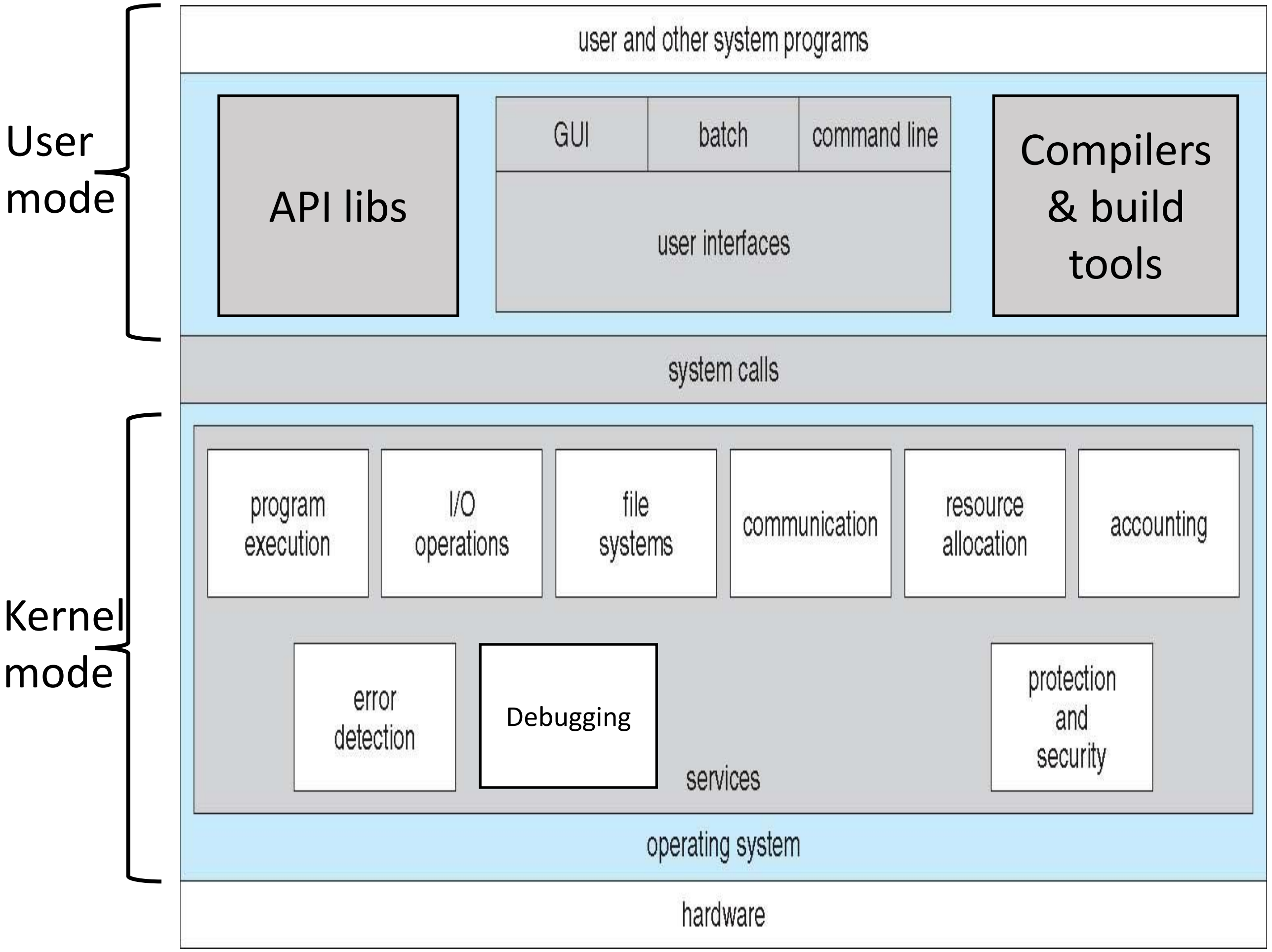
Operating System Services (Cont.)

- The second main goal of an OS is to provide functions that ensure the efficient and secure operation of the system:
  - **Resource allocation -** When multiple jobs (or processes) are 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/processes** use how much and what kinds of computer **resources** (may collect statistics useful for researchers or system admins for detecting system misuse or intrusion).



Operating System Services (Cont.)

- **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



2.2 Operating System’s User Interface - CLI

CLI or **command interpreter** allows direct command entry

- Usually implemented as a systems program
- Sometimes multiple flavors implemented – **shells** (e.g. c shell, Bourne shell, bash, korn shell, ash, etc.).
- Primarily fetches a command from user and executes it
- Two different implementations:
  - Commands are built into the shell (e.g. **BusyBox**, widely used in embedded Linux).
  - Commands are just names of other programs that execute the command. In this implementation, adding new features doesn’t require shell modification (For Linux, usually placed at /bin, as required by the Linux Foundation “File System Hierarchy”, FSH)

Bourne Shell Command Interpreter

Default

New Info Close Execute Bookmarks

Default | Default

```
PBG-Mac-Pro:~ pbg$ w
15:24  up 56 mins, 2 users, load averages: 1.51 1.53 1.65
USER     TTY      FROM              LOGIN@  IDLE WHAT
pbg      console  -                14:34      50 -
pbg      s000     -                15:05       - w
PBG-Mac-Pro:~ pbg$ iostat 5
          disk0           disk1           disk10         cpu      load average
    KB/t tps  MB/s    KB/t tps  MB/s    KB/t tps  MB/s  us sy id   1m   5m   15m
   33.75 343 11.30   64.31  14  0.88   39.67   0  0.02  11  5 84  1.51 1.53 1.65
    5.27 320  1.65    0.00   0  0.00    0.00   0  0.00   4  2 94  1.39 1.51 1.65
    4.28 329  1.37    0.00   0  0.00    0.00   0  0.00   5  3 92  1.44 1.51 1.65
^C
PBG-Mac-Pro:~ pbg$ ls
Applications                    Music                           WebEx
Applications (Parallels)        Pando Packages                  config.log
Desktop                         Pictures                        getsmartdata.txt
Documents                       Public                          imp
Downloads                       Sites                           log
Dropbox                         Thumbs.db                       panda-dist
Library                         Virtual Machines                prob.txt
Movies                          Volumes                         scripts
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
--- 192.168.1.1 ping statistics ---
2 packets transmitted, 2 packets received, 0.0% packet loss
round-trip min/avg/max/stddev = 1.262/1.760/2.257/0.498 ms
PBG-Mac-Pro:~ pbg$ 
```

User Operating System Interface - GUI

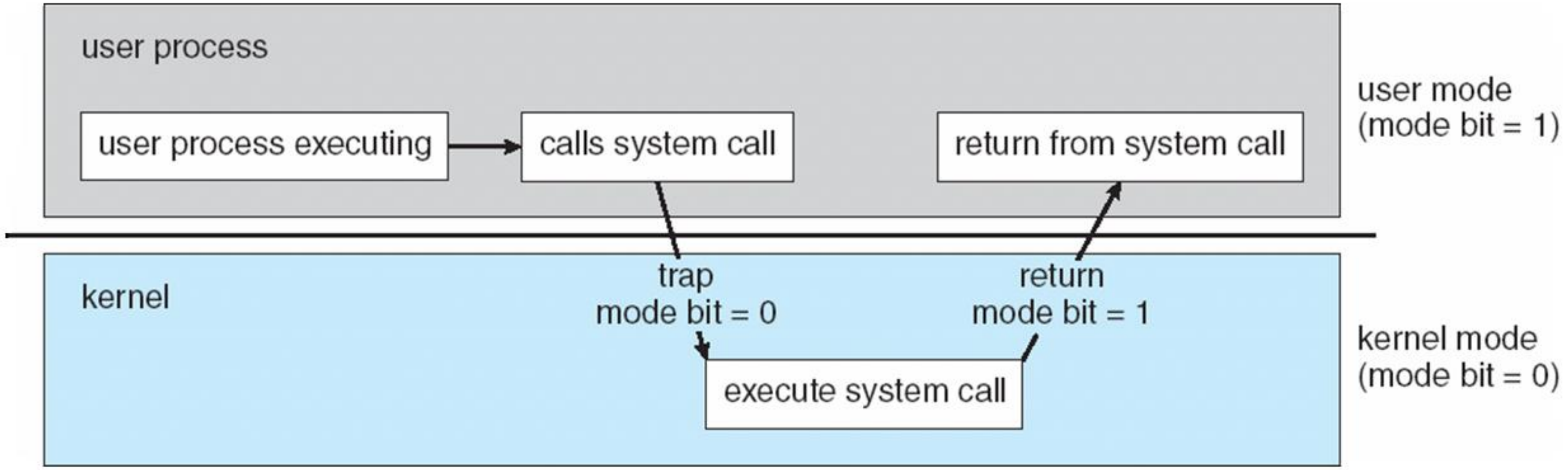
- User-friendly **desktop** metaphor interface
  - Invented at Xerox PARC in 1970 (The first computer to use an early version of the desktop metaphor was the experimental Xerox Alto and the first commercial computer that adopted this kind of interface was the Xerox Star).
  - Usually mouse, keyboard, and monitor
  - **Icons** represent files, programs, actions, etc
  - Clicking mouse buttons over objects in the interface cause different actions (provide information, options, execute function, open directory, etc.)
- Most of today's systems 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)

Touchscreen Interfaces

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

2.3 System Calls

- User-mode program's interface to the services provided by the OS
  - User mode programs **cannot** directly call driver or operating system functions. Instead they need to use system calls.
  - System calls may also be referred to as “supervisor call”, “trap” or sometimes “software interrupt”
  - The application program uses a special machine instruction to perform a system call:
    - SWI – ARM CPUs
    - INT – intel CPUs



Transition from user mode to kernel mode
(from Chapter 1)

System Calls – cont.

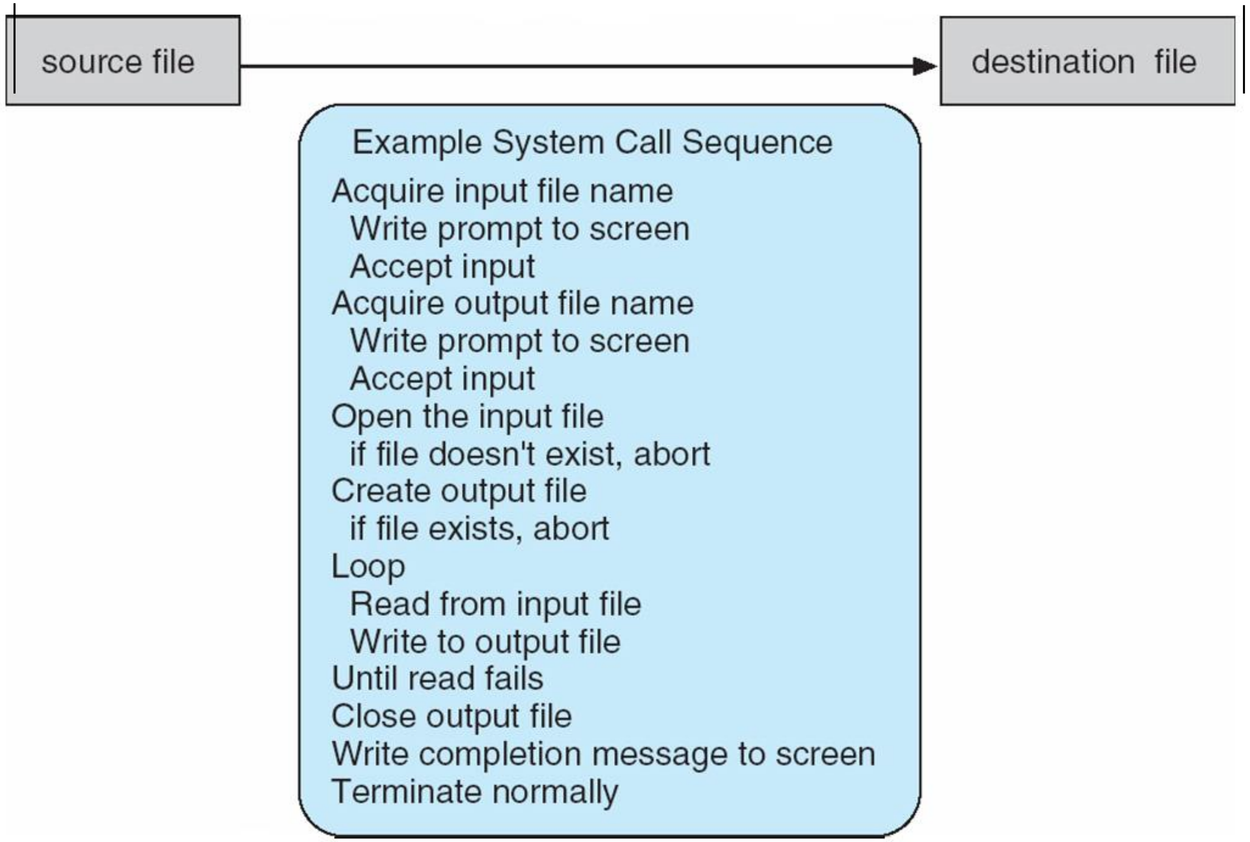
- Mostly accessed by programs via a high-level **Application Programming Interface (API)** libraries (e.g. libc for unix/linux) rather than direct system call use

- Three common APIs are:
  - 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)

Note that unless otherwise stated, the system-call names used throughout this course are generic

Example of System Calls

- The example below demonstrates the steps needed to copy the contents of one file to another file → A sequence of system calls results



Example of Standard API

EXAMPLE OF STANDARD API

As an example of a standard API, consider the `read()` function that is available in UNIX and Linux systems. The API for this function is obtained from the `man` page by invoking the command

```
man read
```

on the command line. A description of this API appears below:

```
#include <unistd.h>

ssize_t    read(int fd, void *buf, size_t count)
```

return value	function name	parameters
`ssize_t`	`read`	`(int fd, void *buf, size_t count)`

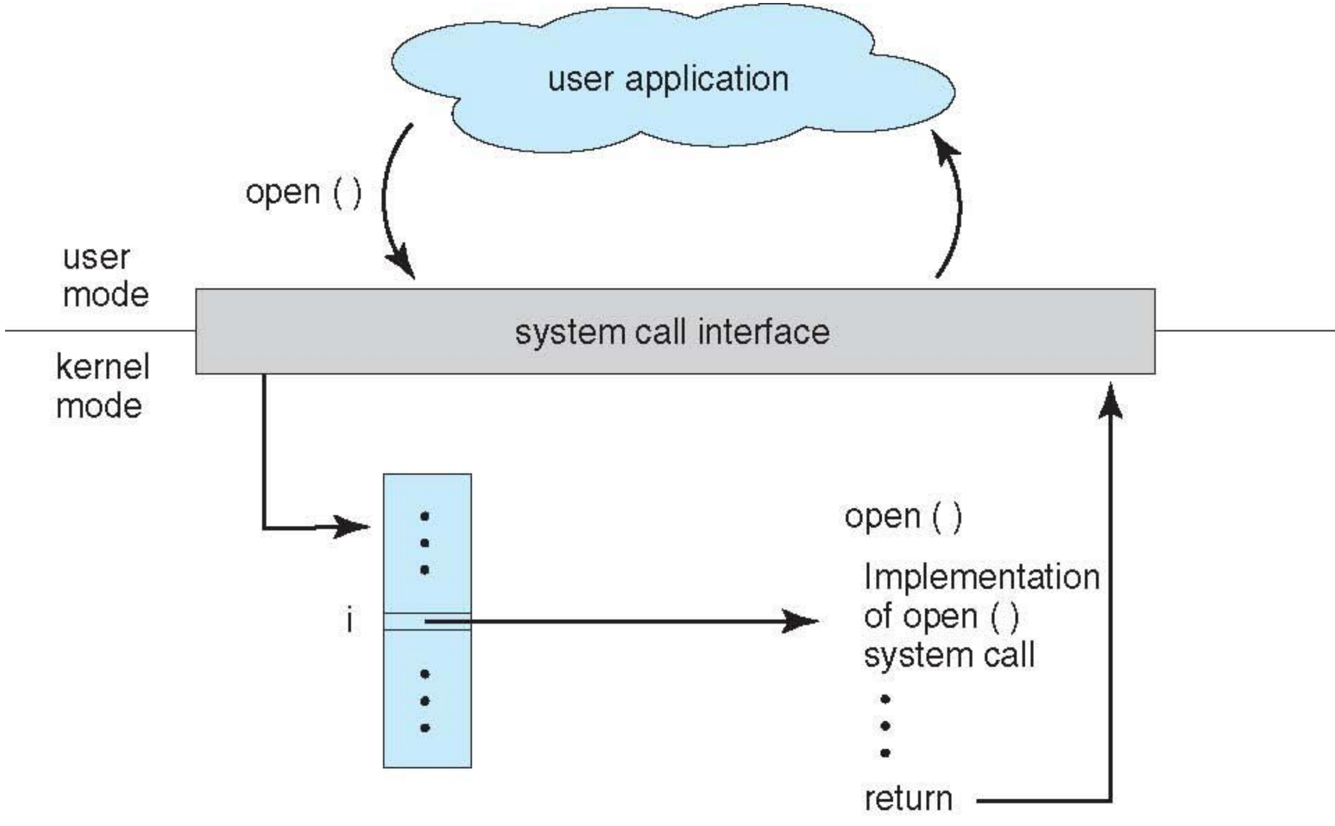
A program that uses the `read()` function must include the `unistd.h` header file, as this file defines the `ssize_t` and `size_t` data types (among other things). The parameters passed to `read()` are as follows:

- `int fd`—the file descriptor to be read
- `void *buf`—a buffer where the data will be read into
- `size_t count`—the maximum number of bytes to be read into the buffer

On a successful read, the number of bytes read is returned. A return value of 0 indicates end of file. If an error occurs, `read()` returns −1.

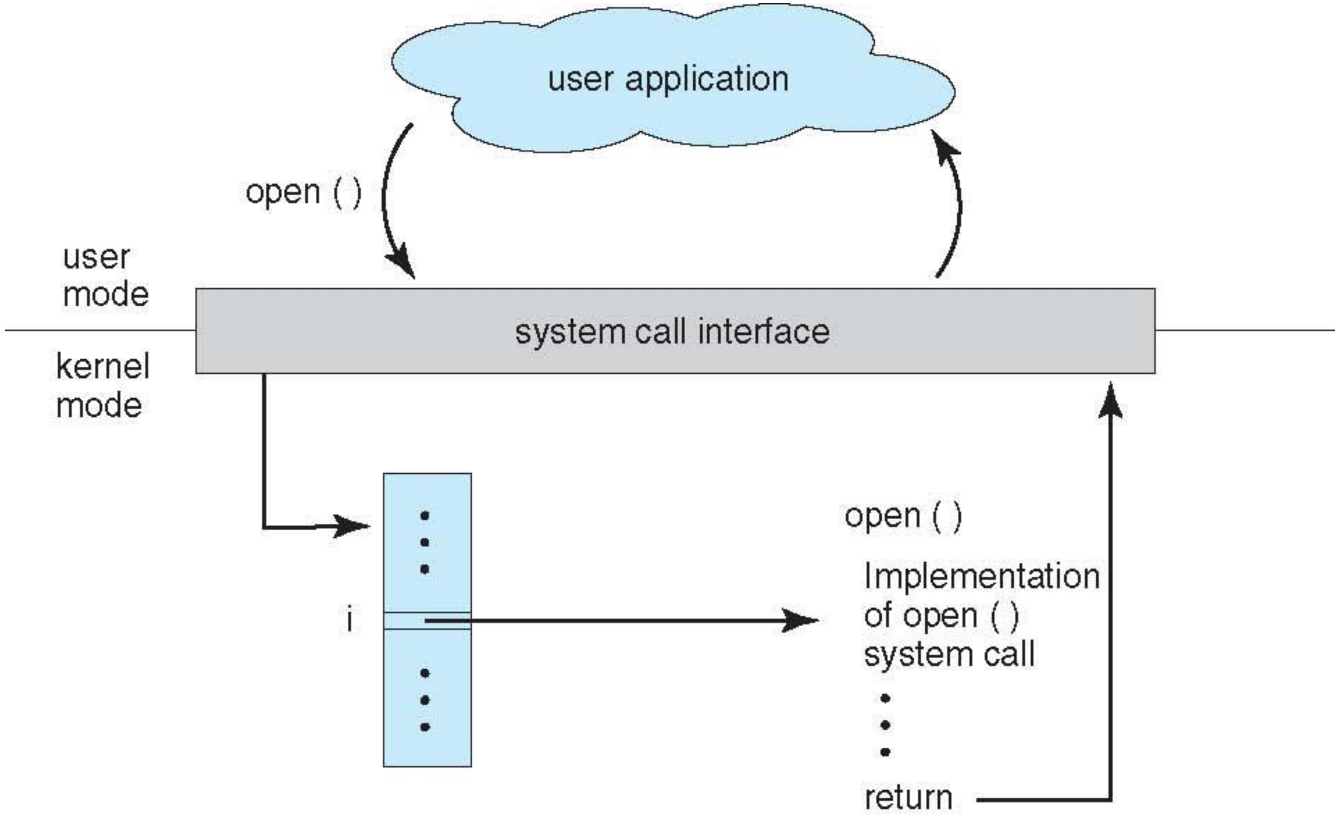
System Call Implementation

- 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



System Call Implementation (cont.)

- 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 of the 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)



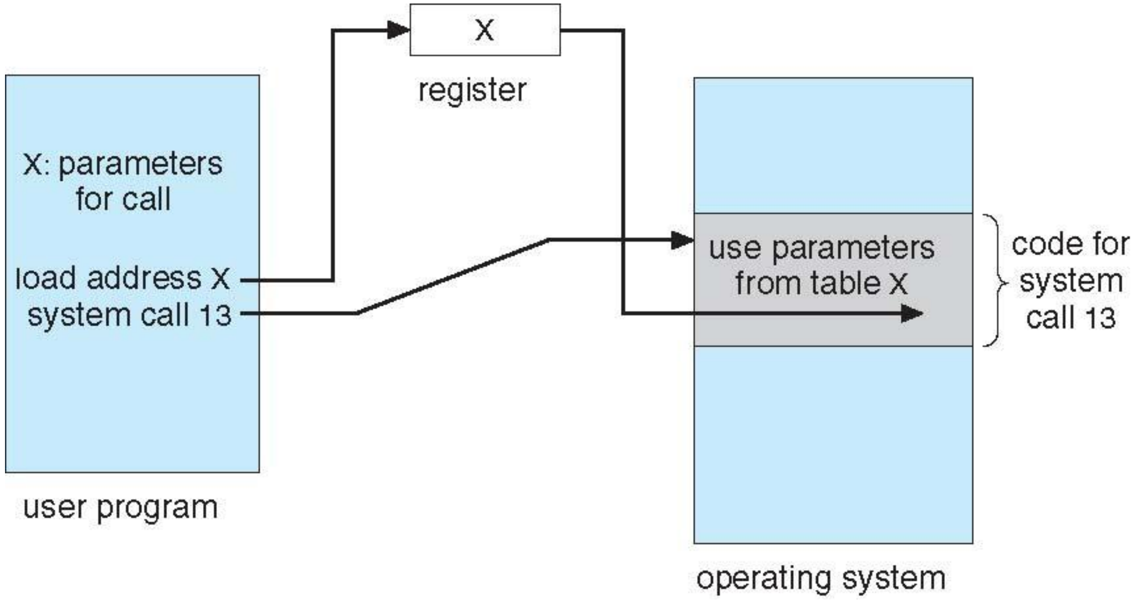
System Call Parameter Passing

- Often, more information is required than simply identity of desired system call
  - Exact type and amount of information vary according to system function called
- Three general methods used to pass parameters to the OS
  1. Simplest: pass the parameters in registers
     - Disadvantage: In some cases, may be more parameters than registers

System Call Parameter Passing – cont.

2. Parameters stored in a block in memory, and address of block/table passed as a parameter in a register
    - This approach taken by Linux and Solaris
3. Parameters placed, or **pushed**, onto the **stack** by the program and **popped** off the stack by the operating system

- Unlike the registers method, the block and stack methods do not limit the number or length of parameters being passed



Parameter Passing via memory block/table

2.6 Operating System Design

- There is no unique process for designing an OS, but some approaches have proven successful. Design starts by defining goals and specifications

- At highest level, design is affected by choice of **hardware** and **type of system**:
  - batch, single user, or multi user
  - Distributed?
  - real time?

- Next level are requirements for **user** goals and **system** goals
  - User goals – operating system should be
    - Easy to use
    - Responsive
  - System goals – operating system should be
    - Easy to design, implement, and maintain
    - Flexible and efficient
    - Reliable and error-free

Operating System Design

- An important design choice may be to separate

  Mechanism: ***How*** to do it?

  Policy: ***What*** specifically will be done?

  - e.g.: The OS timer construct for preempting processes is a mechanism, but the exact tick time is a policy.
- The separation of policy from mechanism is a very important principle, it allows maximum **flexibility** if policy decisions are to be changed later
  - e.g.: A scheduling mechanism may be made general-purpose to allow various policy setups such as time-sharing, batch, real-time etc.
- In some systems, both mechanisms and policy are encoded to enforce a **global look** such as in Windows and Mac OS X, but not in Unix (e.g. different window managers such as KDE, GNOME, etc.)

Operating System Implementation

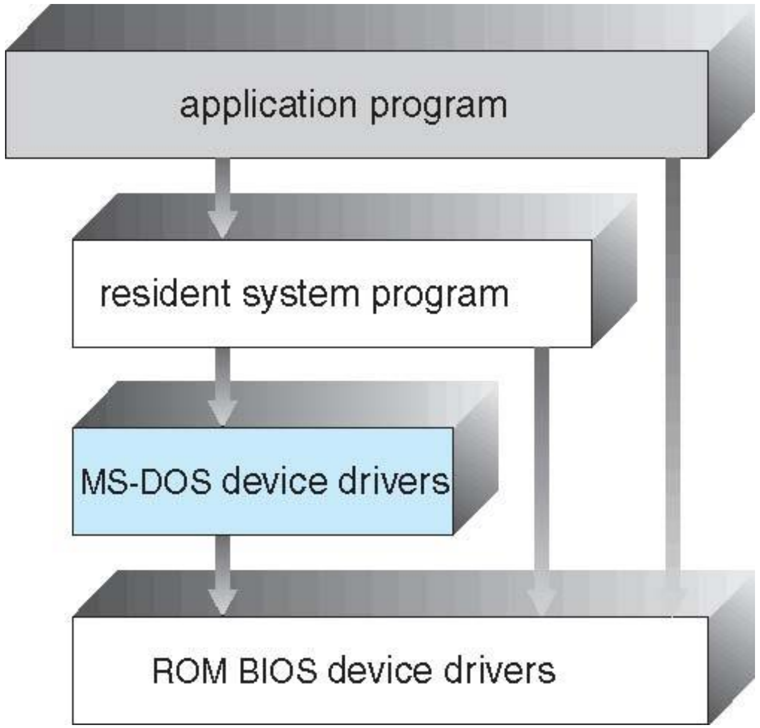
- Early OSes in assembly language (efficient but difficult to code/maintain)
- C, C++ (less memory-efficient, less performance but easier to code/maintain)
- Now a mix of languages
  - **Lowest levels** in assembly, for time critical areas, e.g. **within** interrupt handlers, device and memory managers, and schedulers
  - Main body of **kernel** is written in C. Improving the data structures and algorithms has more impact than coding in a lower level language.
  - **Systems programs** in C, C++, scripting languages like PERL, Python, shell scripts

2.7 Operating System Structure

- A general-purpose OS is very large and complex program
- Various ways to structure an OS:
  - Simple structure – e.g. MS-DOS
  - More complex – e.g. UNIX
  - Layered – provides abstraction levels
  - Microkernel – e.g. Mach and Minix

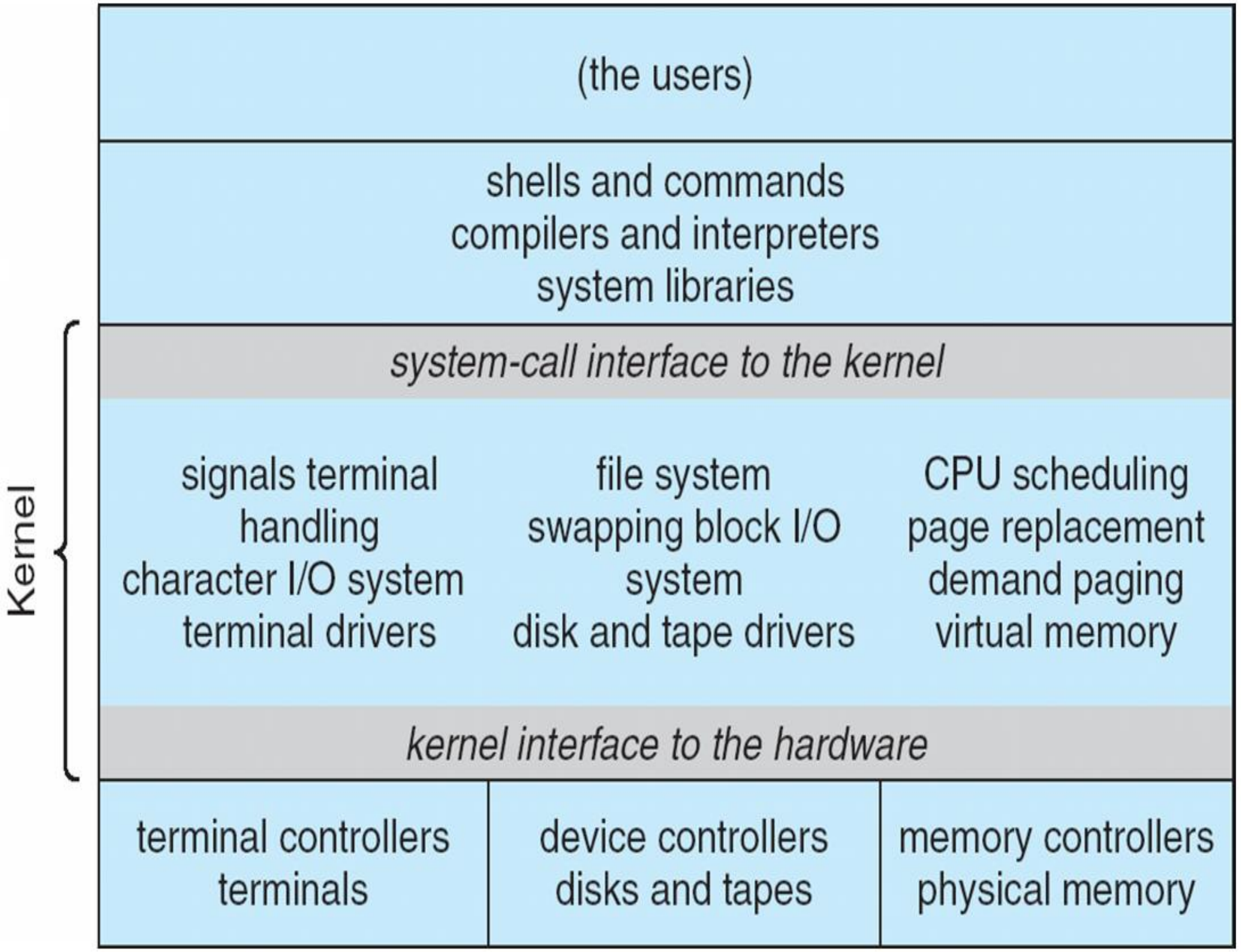
Simple Structure -- MS-DOS

- 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



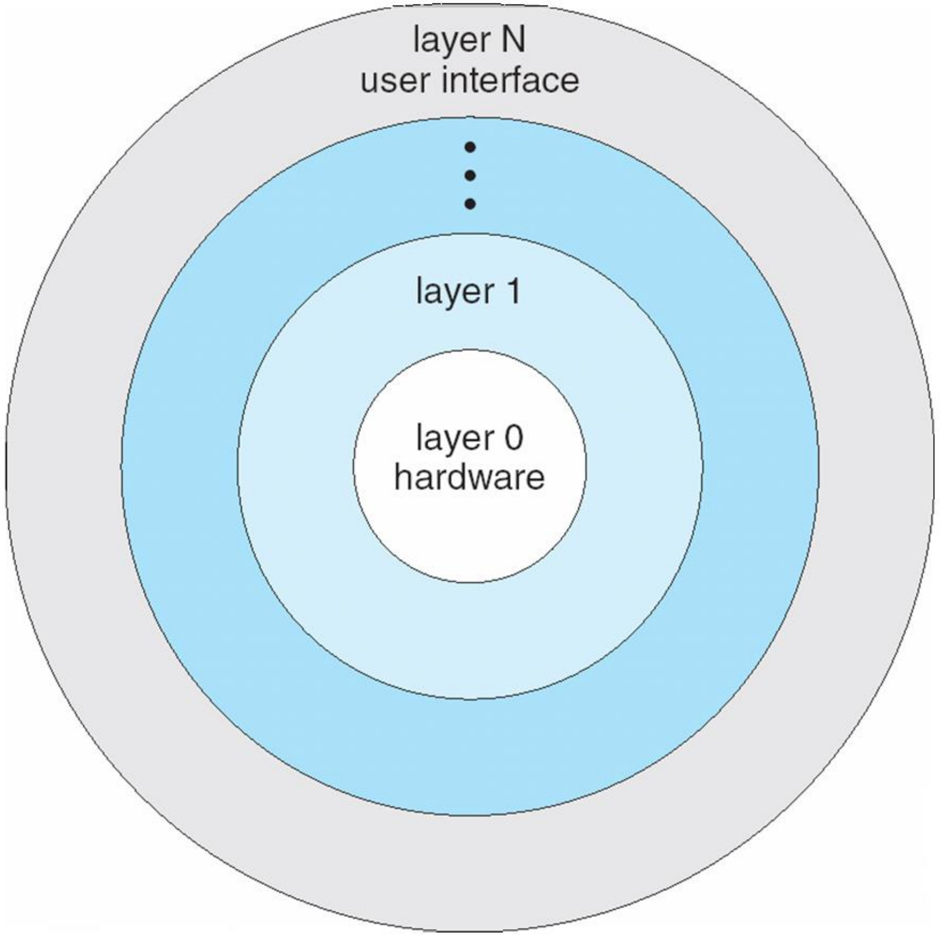
Simple Structure -- The original Unix

- UNIX – limited by hardware functionality, the original UNIX operating system had limited structuring. The UNIX OS consists of two separable parts
  - Systems programs
  - 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
    - That’s large number of functions for a monolithic one-level system.
  - Beyond simple, but not fully layered.



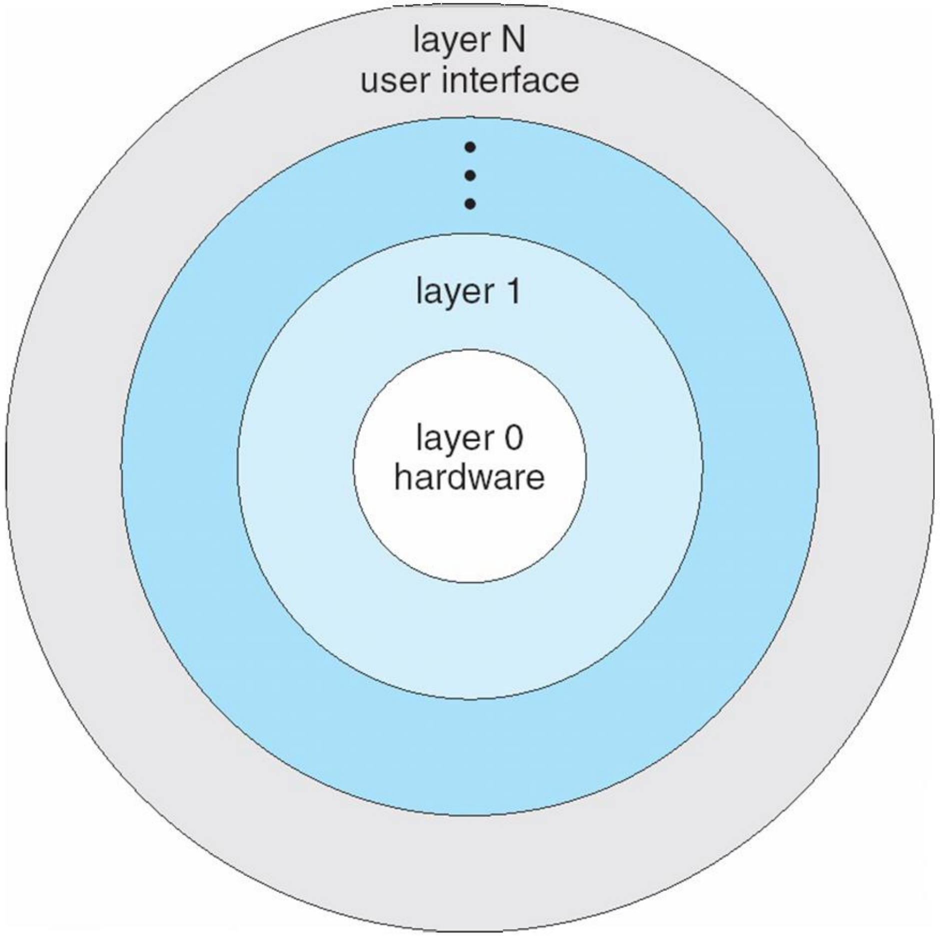
Layered (hierarchial) Approach

- 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.



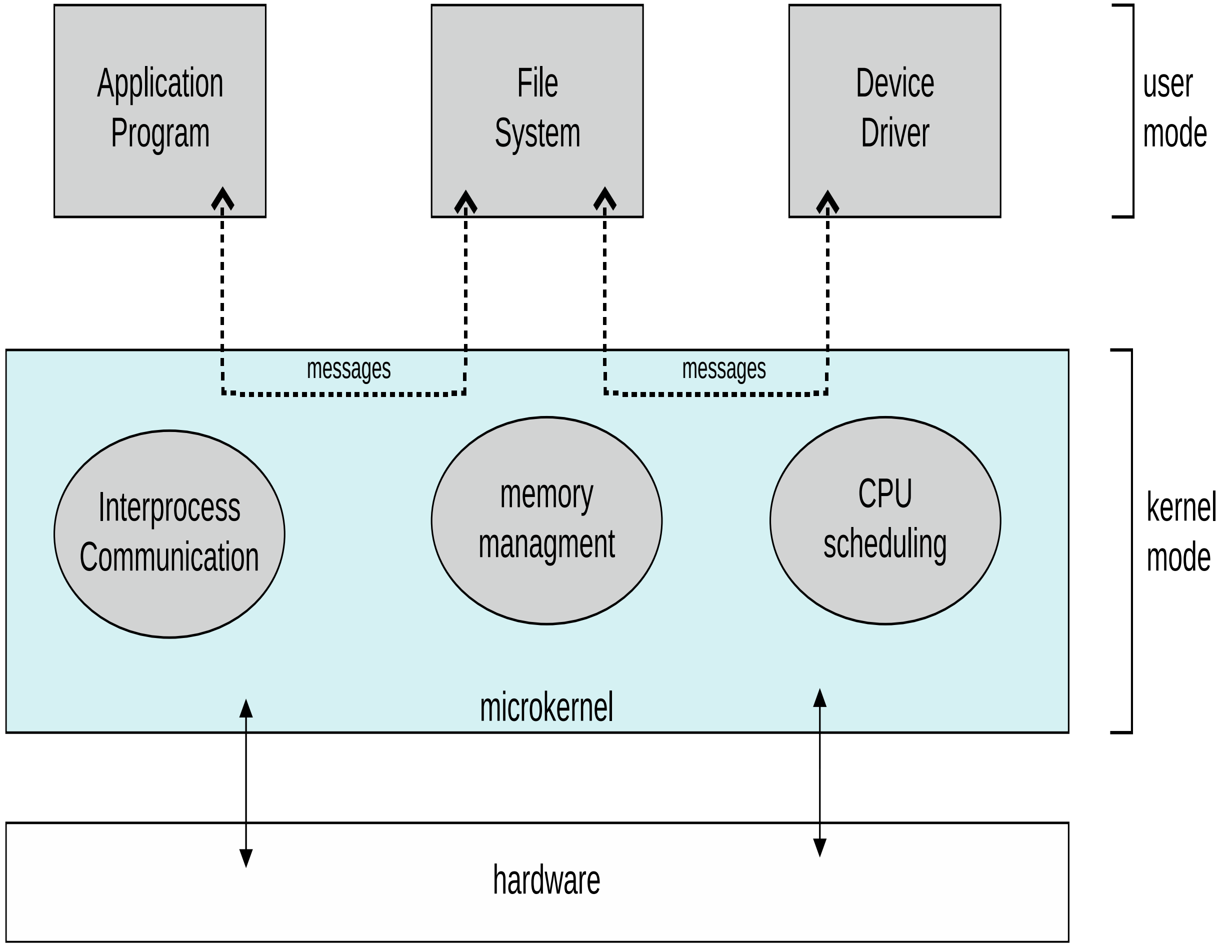
Layered Approach (cont.)

- **Advantages:** Ease of implementation and debugging.
  - The lowest layer may be debugged first (without debugging the higher layers), and once that's done, the next layer can then be debugged, and so on.
- **Disadvantages:** (caused the layered approach to fall out of favor)
  - Difficulty in realizing layered levels (e.g. both a memory manager and disk manager may want use each other's services, same for scheduler/disk manger)
  - Less efficient – a call from upper layer propagates through many lower layers.



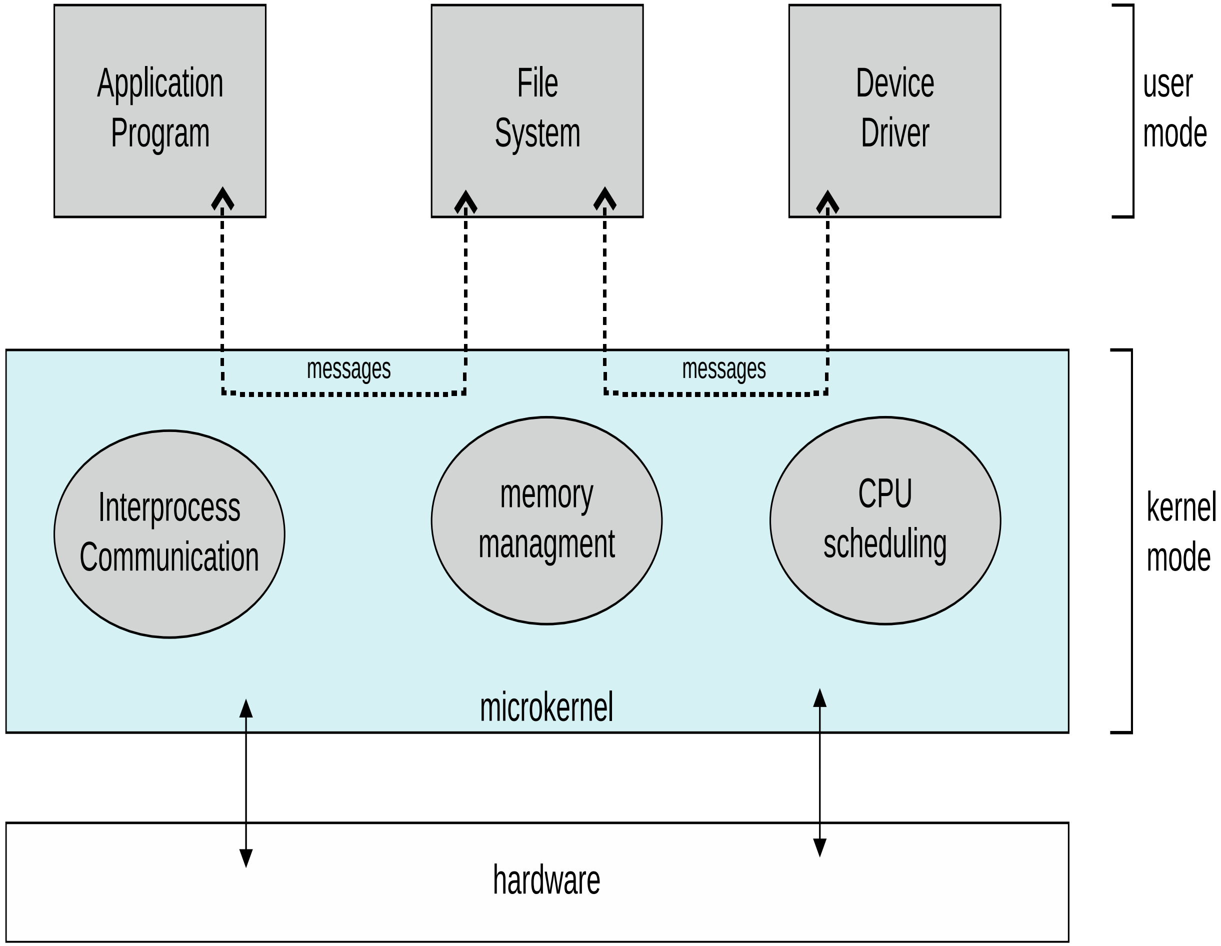
Microkernels

- Moves as much from the kernel into user space
- **Mach** example of **microkernel**
  - Developed in mid 1980's @ Carnegie Mellon university.
  - Maps Unix system calls to messages sent to appropriate user level services
  - Mac OS X kernel (**Darwin**) partly based on Mach
- Micro-kernel provides minimal process and memory management, in addition to a communication facility.
- Communication takes place between user modules using **message passing** via the microkernel.



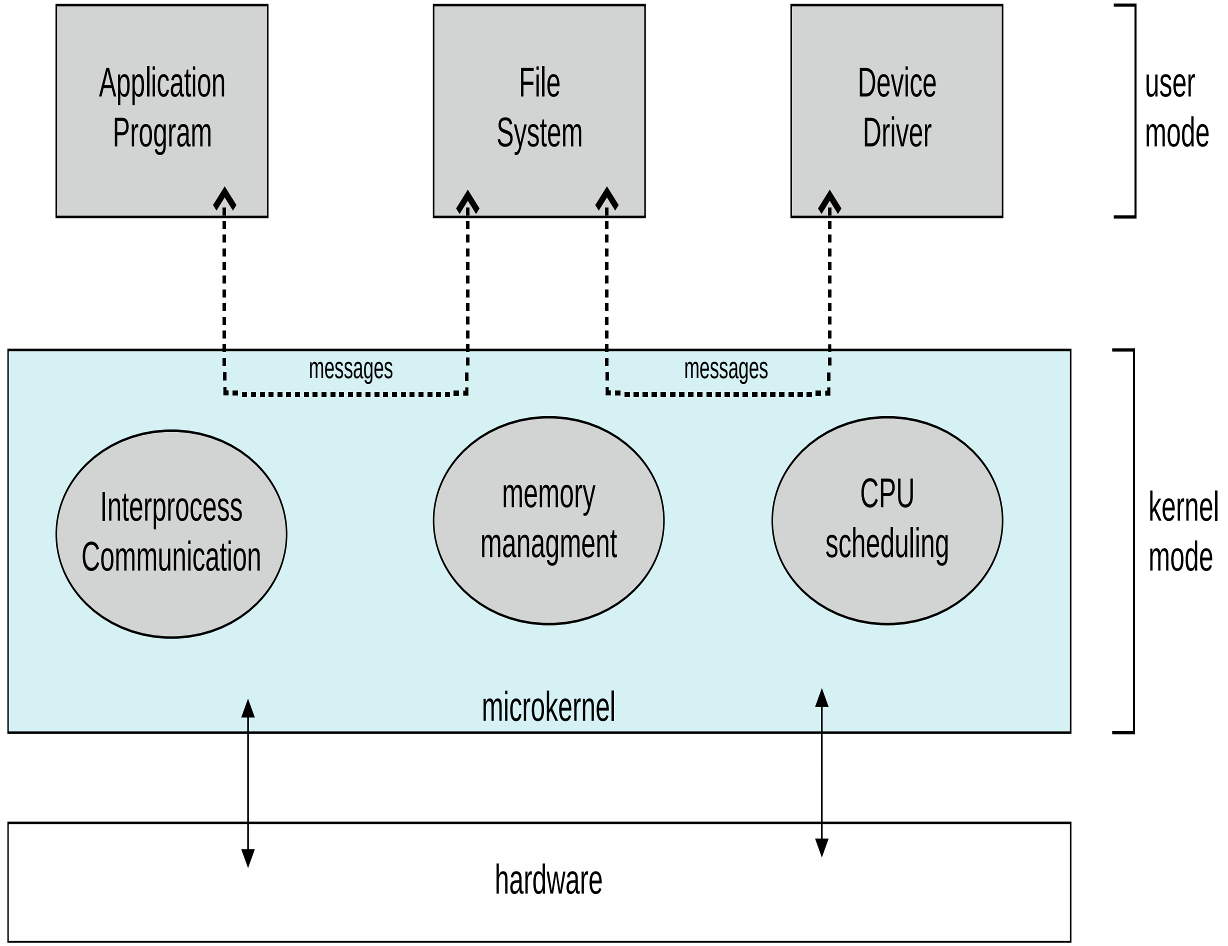
Microkernels – cont.

- Advantages:
  - 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
- Disadvantages:
  - Performance overhead of user space to kernel space, as well as user space to user space communications



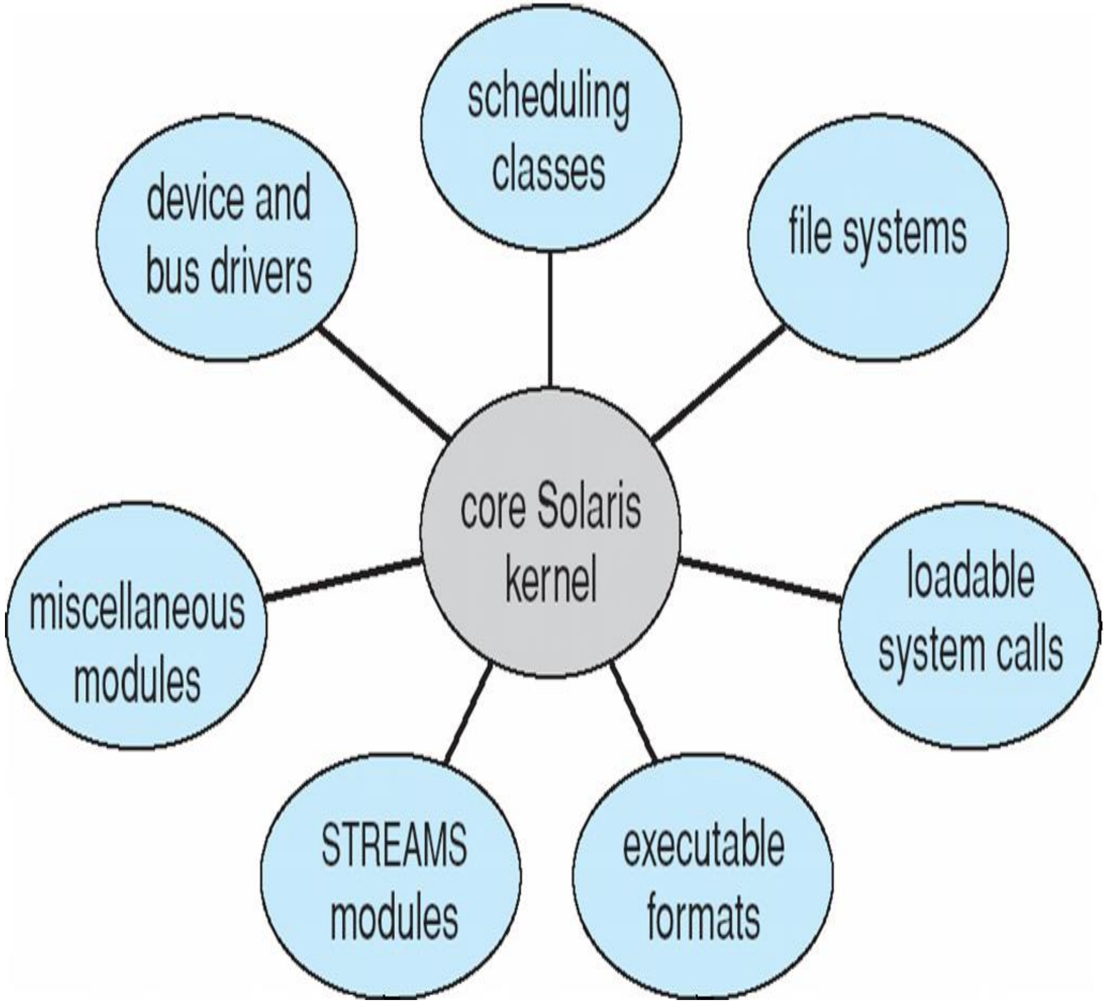
Microkernels – cont.

- Other examples:
  - Tru64 (aka Digital Unix)
  - Minix
  - **QNX Neutrino**,
    - An embedded real-time OS.
    - Kernel only handles process scheduling, memory management, IPC, low level network comm. and H/W interrupts.
    - The Inter-process communication mechanism allows a process waiting for a message to be immediately invoked when a message is sent to it, without invoking the scheduler.
    - IPC messages are sent according the priority of the receiving process
    - Hence tight coupling between the scheduler and IPC.



Subsystems and Loadable Modules

- Many modern operating systems implement **subsystems** and **loadable kernel modules**
  - The kernel has a set of core components/subsystems that are linked (at compile time or load time) to additional services via modules.
  - Subsystems and modules talk to each others over known interfaces
  - Modules may be loaded as needed within the kernel (preferred, as opposed to compile-time linking)
- Overall, similar to layers but with more flexibility
  - Linux, Solaris, Mac OS X, Windows, etc
- The solaris system shown has 7 loadable modules.
- Linux has loadable modules primarily for device drivers and file systems.



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** have kernel components that run in the same address space, **plus modular** for dynamic loading of functionality
  - **Windows** is has subsystems and modules, but it retains some **microkernel-like** behavior since it has different subsystems running as user-mode processes (referred to as personalities). At the same time it provides dynamically loadable kernel modules.